

SCWCD – Study Guide

Exam: CX-310-081

Book: *Head First Servlets & JSP*

Authors: Bryan Basham, Kathy Sierra, Bert Bates

Abridger: Barney Marispini

CHAPTER 1 (Overview)

HTTP

- HTTP stands for HyperText Transfer Protocol.
- HTTP is the protocol clients and servers use to communicate on the web.
- HTTP runs on top of TCP/IP.
 - TCP makes sure that files sent as packets across the wire end up as complete files.
 - IP is the underlying protocol that moves/routes packets from one host to its destination.
- HTTP is a stateless protocol; in other words, it uses stateless connections. As far as the Container is concerned, *each request* is from a *new client*. It doesn't differentiate one client from another.
- HTTP uses the request/response model.
 - The client makes a "request" while the server sends back a "response".
- If the response from the server is an HTML page, then the HTML is "added" to the HTTP response.
- An HTTP request includes the request URL (the resource the client is trying to access), the HTTP method (GET, POST, etc.), and optionally form parameter data (query string).
- An HTTP response includes a status code, the content-type (MIME type), and the actual content of the response (HTML, image, etc.).
- The HTTP header's "content-type" value is known as the MIME type. It tells the browser what TYPE of data is being sent so that it knows how to render it. The MIME type value relates to the HTTP request's "accept" header.
- There are 8 HTTP methods...

HTTP Method	Use
GET	Asks the server to get or return the requested resource.
POST	Asks the server to accept the body info attached to the request, and then get or return the requested resource.
HEAD	Asks for only the header part of whatever a GET would return. It's like a GET with no body in the response. Gives info about the requested URL without actually getting the resource.
TRACE	Asks for a loopback of the request message so that the client can see what's being received on the other end, for testing or troubleshooting.
OPTIONS	Asks for a list of HTTP methods to which the thing at the requested URL can respond.
PUT	Tells the server to put the enclosed info/resource (body) at the requested URL.
DELETE	Tells the server to delete the resource at the requested URL.
CONNECT	Says to connect for the purposes of tunneling. This is the only HTTP method that does not have a corresponding doXXX() method in HttpServlet.

GET & POST

- GET is the simplest HTTP method; it asks the server to "get" a resource and send it back.
 - Use GET to GET things/resources.
 - Hyperlinks always use GET.

- GET is the default for HTML forms.
- GET is idempotent (according to the HTTP spec)...
 - It is not supposed to be used to change anything on the server.
 - It can be executed more than once without any bad side effects.
 - It's safe because it doesn't change anything on the server (unlike POST).
- There are three main limitations...
 - **Size** - The total number of characters that can be sent with a GET is limited.
 - **Security** - Data sent with a GET is insecure in that it is appended to the URL and thus visible to the user. Do not use a GET for sending passwords or sensitive information.
 - **Bookmarking** - GET submissions can be bookmarked which is not always a good thing.
- With POST, you can "request" something and at the same time "send" form data to the server.
 - Use POST to UPDATE or CHANGE something on the server.
 - POST is NOT idempotent (according to the HTTP spec)...
 - Because POST is intended to update or change things on the server, double submissions will typically cause undesirable results or side effects. The data submitted might be destined for a transaction that can't be reversed.
 - There is no limit to the amount of data (parameters) that can be sent to the server.
 - POST requests include form data in the body of the request (sometimes called the "payload").
 - Because GET is the default, POST requests must be explicitly specified in the HTML form tag...
 - `<form action="POST">`
- If by chance you do not implement the appropriate HTTP methods (i.e. doGet(), doPost()) in your servlet, the request will fail. Servlets don't default to GET like HTML forms if not specified.

URL

- URL stands for Uniform Resource Locator.
- The anatomy is as follows...
 - **http://www.domain.com:80/folder/subfolder/file.html?key=value**

Example	Name	Description
http://	Protocol	Tells the server which communications protocol to use.
www.domain.com	Server	<ul style="list-style-type: none"> • The unique name of the server • Maps to an IP address.
:80	Port	<ul style="list-style-type: none"> • Server applications are identified by ports. • Ports are 16-bit numbers that identify specific software programs. • There are 65,536 possible ports. • The TCP port numbers from 0 to 1023 are reserved for well-known services. • 80 is the default port for "web" servers (when not specified). • HTTPS uses port 443.
/folder/subfolder/	Path	<ul style="list-style-type: none"> • The path to the location on the server of the resource being requested. • The FIRST slash is the ROOT of the application.
file.html	Resource	The name of the content being requested.
?key=value	Query String	Optional form data (key/value pairs).

CGI vs. Servlets

- With CGI, servers have to launch a heavy-weight process for each separate request.
- Servlets are more efficient in that they stay loaded while client requests are handled as separate threads of a single running servlet. There's also no overhead of starting the JVM or loading the class. In addition, servlets can interact with other J2EE clients such as EJB.

- Web servers are good at serving static HTML pages, but if you need dynamically-generated data then you need some other helper application that can work with the server. The non-Java term for these helper applications (typically written in PERL) is CGI (Common Gateway Interface).

CHAPTER 2 (Web App Architecture)

Containers

- Servlets are controlled by the Container.
- When the web server gets a request for a servlet, the web server hands the request to the Container who then passes it to the servlet invoking its methods (doGet(), doPost(), etc.).
- Containers provide...
 - **Communications Support** - Built in communication with the web server.
 - **Lifecycle Management** - Takes care of loading, instantiating, initializing, invoking, and making servlets eligible for garbage collection.
 - **Multithreading Support** - Automatically creates a new Java thread for every servlet request.
 - **Declarative Security** - XML deployment descriptors configure security without the need to hard-code it into your servlets. No need to recompile when security changes.
 - **JSP Support** - Translates JSPs into servlets.
- The Container creates a request and response object that servlets can use to get information about the request and send information (response) to the client.
- J2EE application servers contain both a web container and an EJB container.
- Tomcat is just a web container; not a J2EE application server.
- J2EE 1.4 server includes...
 - Servlet Specification 2.4
 - JSP Specification 2.0
 - EJB Specification 2.1

Servlets

- Servlets have three names...
 - **Full Path Name** - The "fully-qualified" name that includes both the package and class name.
 - **Public URL Name** - A made-up public-facing, "client-known" name.
 - **Deployment Name** - A made-up "secret internal" name known only to the deployer.
 - **<servlet>** - Maps the "deployment name" to the "full path name".
 - **<servlet-name>** - Ties the <servlet> to a particular <servlet-mapping> (never seen by the end user).
 - **<servlet-class>** - The fully-qualified name of the class without the ".class" extension.
 - **<servlet-mapping>** - Maps the "deployment name" to the "public URL name".
 - **<servlet-name>** - Matches the <servlet-name> contained in <servlet>.
 - **<url-pattern>** - A made-up name the client sees and uses (Public URL Name).
- Mapping servlet names improves your application's flexibility and security. It keeps clients from knowing the inner structure of your application and allows you the freedom to restructure without having to change every single reference to the restructured servlet.
- A typical servlet is a class that extends `HttpServlet` and overrides one or more service methods that correspond to HTTP methods invoked by the browser (doGet(), doPost(), etc.).

MVC

- MVC stands for Model-View-Controller.
 - **Model** - Holds the business logic and state. It's the only part that talks to the database.
 - **View** - Responsible for the presentation.
 - **Controller** - Forwards the request to the model and then back to the view.
- With MVC the business logic is not only separate from the presentation, it doesn't even know that there is a presentation.
- MVC takes the business logic out of the servlet, and puts it into a model (POJO).

- Models are a combination of business data and the methods that operate on that data.

CHAPTER 3 (MVC)

Key APIs

- `javax.servlet.http.HttpServlet` extends `javax.servlet.GenericServlet` implements `<<javax.servlet.Servlet>>`
- `<<javax.servlet.http.HttpServletRequest>>` extends `<<javax.servlet.ServletRequest>>`
- `<<javax.servlet.http.HttpServletResponse>>` extends `<<javax.servlet.ServletResponse>>`

[See Tutorial]

CHAPTER 4 (Request & Response)

Servlets Are Controlled by the Container

- The Container...
 - Creates the request and response objects.
 - Finds the correct servlet in web.xml based on the URL (pattern) in the request.
 - Creates or allocates a new thread for the servlet.
 - Calls the servlet's `service()` method, passing the request and response references as arguments.
- The Servlet...
 - Determines which servlet method to call based on the HTTP method (GET, POST, etc.) sent by the client (`<form action="POST">`).
 - Forwards the request and response objects as arguments to the appropriate servlet method (`doGet()`, `doPost()`, etc.) who then uses the request object to write out the response to the client.
 - The response is sent back to the Container.
 - The `service()` method completes.
 - The thread either dies or returns to a Container-managed thread pool.
 - The request and response object references fall out of scope and are then garbage collected.
 - The client gets the response from the Container.
- The Container runs MULTIPLE THREADS to process MULTIPLE REQUESTS to a SINGLE SERVLET. In other words, every request to a servlet runs in a separate thread. Also, there is ever only one instance of a particular servlet class.
- Every client request generates a new pair of request and response objects. There is one thread per request (even if the same client makes multiple requests).

Servlet Lifecycle

- There is one main state - initialized.
 - If it isn't initialized then it's...
 - Being initialized (running its constructor or `init()` method).
 - Being destroyed (running its `destroy()` method).
 - Does not exist.
- The Container...
 - Loads the servlet class.
 - Instantiates the servlet (by running the no-arg constructor).
 - You should NOT write your own constructor; instead, use the compiler-supplied default.
 - The constructor creates an object NOT a Servlet. It isn't until the `init()` method completes that the object actually becomes a servlet. At this stage it is too early to perform servlet-specific things; any attempt will fail.
 - Calls the `init()` method.
 - Called only once in the servlet's life.
 - This is when the object actually becomes a servlet.

- Once a servlet, you have access to...
 - ServletConfig (Configuration of the individual servlet).
 - One per servlet.
 - Used to pass deploy-time information (database name, EJB name, etc.).
 - Used to access the ServletContext.
 - Parameters are configured in web.xml.
 - ServletContext (think instead as “AppContext”)
 - One per web app (not per servlet).
 - Used to access web app parameters (also configured in web.xml.).
 - Think as a bulletin-board for messages (attributes) that the entire app shares.
 - Used to get server info (Container name, version, API supported, etc.).
- Always completes before the first call to the service() method.
- Used to initialize the servlet before handling any client requests.
- Only override when you have initialization code like...
 - Getting a database connection.
 - Registering yourself with other objects.
 - Etc.
- Calls the service() method.
 - Handles client requests by calling the appropriate HTTP method (doGet(), doPost(), etc.).
 - Each request runs a separate thread (called in its own stack).
 - Most of the servlet’s life is spent running a service() method for a client request.
 - You should NOT override the service() method...
 - Let the HttpServlet implementation worry about calling the right HTTP method.
 - Instead ALWAYS override whichever HTTP method(s) you support...
 - doGet(), doPost(), doHead(), doOptions(), doPut(), doTrace(), and doDelete().
- Calls the destroy() method.
 - Called only once in the servlet’s life, this is the servlet’s chance to clean up before it’s made ready for garbage collection.
- HttpServlet extends GenericServlet – an ABSTRACT class that implements most of the basic servlet methods. GenericServlet implements the Servlet interface.
- Servlet classes (except those related to JSPs) are in one of two packages...
 - javax.servlet.*
 - javax.servlet.http.*

Request

- The doGet() and doPost() methods both take as arguments...
 - HttpServletRequest
 - HttpServletResponse
- Common request methods...
 - String parameterValue = request.**getParameter**(“parameterName”)
 - Gets a specific parameter from the request.
 - String[] parameterValues = request.**getParameterValues**(“parameterName”)
 - Gets multiple values for a single parameter in the request.
 - Used for checkboxes, multiple-select list boxes, etc.
 - String client = request.**getHeader**(“User-Agent”)
 - Gets the specified request header value such as the client’s platform and browser info.
 - Cookie[] cookies = request.**getCookies**()
 - Gets the cookies associated with the request.
 - HttpSession session = request.**getSession**()
 - Gets the session associated with this client.
 - String httpMethod = request.**getMethod**()
 - Gets the HTTP method of the request.
 - InputStream input = request.**getInputStream**()

- Gets an input stream (raw bytes) from the request.
- Use, for example, when you want to strip out all header information and then write the payload (body) to a file on the server.
- The differences between...
 - **getServerPort()** – *The port to which the ORIGINAL request was sent.* All requests are sent to a single port (where the server is listening).
 - **getLocalPort()** – *The port on which the request ENDED UP.* The server finds a different local port for each thread so that the app can handle multiple clients at the same time.
 - **getRemotePort()** – *Means get the CLIENT'S PORT since the client is REMOTE to the server, who is actually the one asking the question. Remember, if you're a servlet, remote means client.*

Response

- The response is what goes back to the client. The thing the browser gets, parses, and renders for the user. Typically, you use the response object to get an output stream (usually a `Writer`) and then use that stream to write content for the user (HTML).
- Most of the time, you use the response just to *send data* back to the client.
- Common response methods...
 - `response.setContentType("text/html")`
 - Tells the browser what to EXPECT so it can handle the response accordingly.
 - Always set the content type (MIME type) BEFORE writing to the output stream.
 - Common content types include...
 - `text/html`
 - `application/pdf`
 - `image/jpeg`
 - `PrintWriter out = response.getWriter()`
 - Used for printing TEXT data to a character stream.
 - While you CAN write character data to an `OutputStream`, the `PrintWriter` is the stream designed for character data.
 - The `PrintWriter` returned by this method actually *decorates* the `ServletOutputStream` returned by the `getOutputStream()` method.
 - `ServletOutputStream out = response.getOutputStream()`
 - Used for writing BYTES to the output stream. Use for anything other than character data.
- Remember...
 - `PrintWriter`
 - `getWriter()` gets a `PrintWriter`.
 - `println()` to a `PrintWriter`.
 - Used for character data (**text**).
 - `ServletOutputStream`
 - `getOutputStream()` gets a `ServletOutputStream`.
 - `write()` to a `ServletOutputStream`.
 - Used for **binary** data (can also do text).
- Both the `setHeader()` and `addHeader()` methods will add a header *if it does not already exist*; the difference is that the `setHeader()` will *override the previous value* for an existing header while `addHeader()` will *add another value* to that header.
- You can also use the response to set headers, send errors, and add cookies.
- Typically you'll use JSP to send HTML responses (as opposed to using a `PrintWriter`).
- When sending binary data to the client use the `ServletOutputStream`.

Redirect vs. Request Dispatch

- Redirect
 - A servlet redirect makes the *browser* do the work (redirect = client).
 - It changes the URL in the address bar (pops the browser).
 - When a servlet does a redirect, it's like asking the client to call someone else instead; which is why the URL in the address bar changes.

- Relative URLs come in two flavors...
 - With a starting forward slash (i.e. `response.sendRedirect("/foo/stuff.html")`).
 - The forward slash means relative to the root of the web app (context path).
 - Without a starting forward slash (i.e. `response.sendRedirect("foo/stuff.html")`).
 - Because the Container remembers the original request URL, it prepends that to the path specified in the `sendRedirect()` method.
- You CAN'T do a `sendRedirect()` AFTER writing to the response.
 - It will throw an `IllegalStateException` if you try to invoke it after the "response has already been committed" (sent to the browser).
- Remember that `sendRedirect()` takes a String argument that is a URL, not an actual URL object.
- Request Dispatch
 - A request dispatch does the work on the *server* side (request dispatch = server).
 - The URL in the address bar does NOT change.
 - When a servlet does a request dispatch, it's like asking a co-worker to take over working with a client. The client does not care or even know someone else took over.

CHAPTER 5 (Attributes & Listeners)

ServletConfig - Init Parameters

- In order to avoid hard coding your servlets, you can pass init parameters directly from the deployment descriptor (`web.xml`) to each of your servlets. Each servlet can contain its own set(s) of init parameters.
 - Deployment Descriptor


```
<web-app...>
  <servlet>
    <servlet-name>initParameterServlet</servlet-name>
    <servlet-class>com.example.InitParameterServlet</servlet-class>
    <init-parameter>
      <param-name>adminEmail</param-name>
      <param-value>admin@domain.com</param-value>
    </init-parameter>
    <init-parameter>
      <param-name>salesEmail</param-name>
      <param-value>sales@domain.com</param-value>
    </init-parameter>
  </servlet>
</web-app>
```
 - Servlet


```
Enumeration en = getServletConfig().getInitParameterNames();
while (en.hasMoreElements) {
  out.println(getServletConfig().getInitParameter(en.nextElement()));
}
out.println(getServletConfig().getInitParameter("adminEmail"));
```
- When the Container initializes a servlet, it makes a unique `ServletConfig` for each servlet.
- The Container reads the servlet init parameters from `web.xml` and gives them to the `ServletConfig`, then passes the `ServletConfig` to the servlet's `init()` method.
- Remember that you can't use the `ServletConfig` until the servlet is *initialized*.
 - You can't call it from the constructor (at this state, it's still just an object, NOT a servlet)!!!
 - Inside the `init()` method is the soonest the `ServletConfig` can be accessed.
 - The `ServletConfig` contains the init parameters.
- There are TWO `init()` methods, one that takes a `ServletConfig` argument and one that does not.
 - The no-arg version is the one that is typically overridden.
 - If you override the other, be sure to call...


```
super.init(ServletConfig);
```

- The servlet init parameters are read only once - when the Container initializes the servlet.
 - When the Container makes a servlet, it reads the deployment descriptor (web.xml) and creates the name/value pairs for the ServletConfig.
 - Changes to the init parameters will not take effect until the web app is re-deployed.
- If the values of your init parameters are going to change frequently, you're better off having your servlet methods get the values from a file or database.
- The ServletConfig's main job is to give you init parameters.
 - It can also give you a ServletContext (although it is typically retrieved in another way).
- The **ServletConfig** is for **servlet** configuration.
 - There is ONE per servlet!!!
- When not specified as context or servlet, assume "init parameter" is "**servlet** init parameter".

ServletContext - Init Parameters

- Context init parameters work just like servlet (ServletConfig) init parameters, except context parameters are available to the entire web app, not just single servlets.
 - Deployment Descriptor


```
<web-app...>
  <servlet>
    <servlet-name>initParameterServlet</servlet-name>
    <servlet-class>com.example.InitParameterServlet</servlet-class>
  </servlet>
  <context-param>
    <param-name>adminEmail</param-name>
    <param-value>admin@domain.com</param-value>
  </context-param>
  <context-param>
    <param-name>salesEmail</param-name>
    <param-value>sales@domain.com</param-value>
  </context-param>
</web-app>
```
 - Servlet


```
Enumeration en = getServletContext().getInitParameterNames();
while (en.hasMoreElements) {
  out.println(getServletContext().getInitParameter(en.nextElement()));
}
out.println(getServletContext ().getInitParameter("adminEmail"));
```
- Things to remember about Context init parameters...
 - They are declared *outside* <servlet> elements.
 - They do not use the word "init" anywhere (unlike servlet init parameters).
 - They use <context-param> NOT <init-param>.
 - They are accessed in virtually the same way...
 - Servlet


```
getServletContext().getInitParameter("parameterName");
```
 - Context


```
getServletContext ().getInitParameter("parameterName");
```
- There is only ONE ServletContext per web app and all parts of the web app share it.
- Web app initialization...
 - Container reads web.xml and creates name/value pairs for each <context-param>.
 - Container creates a new instance of ServletContext.
 - Container gives the ServletContext a reference to each name/value pair of the context init parameters.
 - Every servlet and JSP deployed as part of a single web app has access to that same ServletContext.
- There is one ServletContext per web app in a single JVM.

- However, in a clustered environment (distributed across multiple servers), your web app could actually have *more than one* ServletContext.
- In order for changes to context and servlet init parameters to be noticed by the web app, the web app must be re-deployed.
- There is no **setInitParameter()** method in either ServletContext or ServletConfig; therefore, it is NOT possible to set values at runtime.
 - Think of init parameters as *deploy-time constants*.
 - You can get them at runtime, but you can't set them.
- Common methods of the Servlet**Context** interface...
 - getInitParameter(String)
 - This returns a String not an object (cast to String not necessary).
 - getInitParameterNames()
 - getAttribute(String)
 - This returns an Object so don't forget to cast (unless you want an Object)!!!
 - getAttributeNames()
 - setAttribute(String)
 - removeAttribute(String)
 - getRequestDispatcher(String)
- There are two ways to get the ServletContext...
 - this.getServletContext().getInitParameter();
 - This one is inherited from GenericServlet.
 - getServletConfig().getServletContext().getInitParameter();
 - This approach is not common, but legal. Use when your servlet does not extend HttpServlet or GenericServlet.

ServletContextListener (Application Scope)

- This is a class (not a servlet or JSP) that...
 - Implements javax.servlet.ServletContextListener
 - Listens for two key events in the ServletContext's life...
 - Initialization
 - Gets notified when the app is initialized (deployed).
 - Examples of use...
 - Get the context init parameters from the ServletContext.
 - Use the init parameter look up name to make a database connection.
 - Store the database connection as an attribute the whole app can access.
 - Destruction
 - Gets notified when the context is destroyed (undeployed or goes down).
 - Examples of use...
 - Close a database connection.
 - Use when you need to run code BEFORE the rest of the app has a chance to service a client.
 - Similar to a main() method in a standalone Java application.
 - ServletContextListener has two methods to implement...
 - context**Initialized**(ServletContext**Event** event)
 - context**Destroyed**(ServletContext**Event** event)
 - The ServletContextEvent object is what the ServletContextListener uses to get a reference to the ServletContext. Once it has the reference it can get and set attributes accordingly.
 - event.getServletContext()
 - To tell the Container about your ServletContextListener, add it to the web.xml file...
 - Deployment Descriptor


```
<web-app...>
  <context-param>
    <param-name>parameterName</param-name>
    <param-value>parameterValue</param-value>
  </context-param>
```

```

<listener>
  <listener-class>com.example.MyServletContextListener</listener-class>
</listener>
</web-app>

```

- Listener
 - ServletContext servletContext = event.getServletContext();
 - String parameterValue = servletContext.getInitParameter("parameterName");
- The Container knows which *type of listener* by inspecting the class (sees the interface).
 - Besides context events, you can listen for...
 - Context attributes
 - Servlet requests
 - Servlet attributes
 - HTTP sessions
 - Session attributes

The Eight Listeners

Scenario	Listener Interface	Event Type
Use when you want to know if an attribute in a web app context has been added, removed, or replaced.	javax.servlet. ServletContextAttributeListener <ul style="list-style-type: none"> • attributeAdded • attributeRemoved • attributeReplaced 	ServletContextAttributeEvent
Use when you want to know how many concurrent users there are. In other words, you want to track the active sessions.	javax.servlet.http. HttpSessionListener <ul style="list-style-type: none"> • sessionCreated • sessionDestroyed 	HttpSessionEvent
Use when you want to know each time a request comes in, so that you can log it.	javax.servlet. ServletRequestListener <ul style="list-style-type: none"> • requestInitialized • requestDestroyed 	ServletRequestEvent
Use when you want to know when a request attribute has been added, removed, or replaced.	javax.servlet. ServletRequestAttributeListener <ul style="list-style-type: none"> • attributeAdded • attributeRemoved • attributeReplaced <p>Note: The getValue() method of the ServletRequestAttributeEvent parameter returns the OLD value of the attribute if the attribute was replaced.</p>	ServletRequestAttributeEvent
Use when you have an attribute class (a class for an object that will be stored as an attribute) and you want objects of this type to be notified when they are bound to or removed from a session. In other words, when the attribute <i>itself</i> needs to know when it has been added to or removed from the session. This is	javax.servlet.http. HttpSessionBindingListener <ul style="list-style-type: none"> • valueBound • valueUnbound 	HttpSessionBindingEvent

Scenario	Listener Interface	Event Type
<i>the only listener that is NOT registered with the Container through the deployment descriptor.</i>		
Use when you want to know when a session attribute has been added, removed, or replaced.	javax.servlet.http. HttpSessionAttributeListener <ul style="list-style-type: none"> • attributeAdded • attributeRemoved • attributeReplaced 	HttpSessionBindingEvent (Inconsistent naming)
Use when you want to know if a context has been created or destroyed.	javax.servlet. ServletContextListener <ul style="list-style-type: none"> • contextInitialized • contextDestroyed 	ServletContextEvent
Use when you have an attribute class, and you want objects of this type to be notified when the session to which they're bound is MIGRATING to and from another JVM.	javax.servlet.http. HttpSessionActivationListener <ul style="list-style-type: none"> • sessionDidActivate • sessionWillPassivate 	HttpSessionEvent

Attributes

- An attribute is an object set (referred to as a *bound*) into one of three other servlet API objects - ServletContext, HttpServletRequest (or ServletRequest), HttpSession. Think of it as simply a name/value (where name is a String and value is an Object) pair in a map instance variable.
- An attribute is like an object pinned to a bulletin board. Somebody stuck it on the board so that others can get it.
- All three interfaces (ServletContext, ServletRequest, and HttpSession) have the same method signatures for accessing attributes...
 - Object getAttribute(String attributeName)
 - setAttribute(String attributeName, Object attributeValue)
 - removeAttribute(String attributeName)
 - Enumeration getAttributeNames()
- Difference between attributes and parameters...

	Attributes	Parameters
Types	Application/Context Request Session Note: There is no servlet-specific attribute.	Application/Context Request Servlet Init Parameter Note: There is no such thing as session parameters.
Get Methods	getAttribute(String name) Note: Don't forget to cast the Object returned (unless you want an Object).	getInitParameter(String name)
Set Methods	setAttribute(String name, Object value)	You CANNOT set Application/Context and Servlet init parameters. They're set in the deployment descriptor (web.xml).
Return Types	Object	String

The Three Scopes

- Context (Application)
 - Accessible to everyone in the application (ServletContext).
 - This scope is NOT thread safe!!!
 - Multiple servlets in multiple threads can manipulate the same, shared attributes.
 - Not even synchronizing the service() method will protect the context attributes.
- Session
 - Accessible to only those with access to a specific HttpSession.
- Request
 - Accessible to only those with access to a specific ServletRequest.

Thread Safety

- **The context scope is NOT thread safe!!!**
 - The problem is that multiple servlets in multiple threads can freely access and manipulate the same, shared attributes.
 - Not even synchronizing the service() method will protect the context attributes.
 - While it controls one servlet's access it does NOT BLOCK another servlet's access. In other words, while it will stop THAT servlet from servicing more than one request at a time, it will NOT stop OTHER servlets and JSPs in the same web app from accessing the context.
 - While context scope attributes are not inherently thread safe, there is one approach that works...
 - **Lock the context (not the servlet).**
 - The typical and BEST way to protect the context attribute is to **synchronize ON the context object itself** - instead of trying to lock the servlet by synchronizing its service() method. If everyone accessing the context has to first get the lock on the context object, then you're guaranteed that only one thread at a time can be getting or setting the context attribute. *But, this only works if all of the other code that manipulates the same context attributes ALSO synchronize on the ServletContext.*

```
public void doGet(...) {  
    ...  
    synchronized(getServletContext()) {  
        getServletContext().setAttribute("attributeName", "attributeValue");  
        out.println(getServletContext().getAttribute("attributeName"));  
    }  
    ...  
}
```

Note: Get the lock on the CONTEXT itself. *Do not use synchronized(this).*

- **The session scope is ALSO NOT thread safe!!!**
 - Session scope attributes may appear to be thread safe on the surface since each request originates from a single client (browser). But what happens when the user has more than one browser instance (window) sharing the same session? This is how the Container can get tripped up - same session, different clients (browser windows).
 - Session scope attributes can be made thread safe using the same technique applied to context attributes. **Protect them by synchronizing on the HttpSession!**
 - Lock the HttpSession.

```
public void doGet(...) {  
    ...  
    HttpSession session = request.getSession();  
    synchronized(session) {  
        session.setAttribute("attributeName", "attributeValue");  
        out.println(session.getAttribute("attributeName"));  
    }  
    ...  
}
```

Note: Get the lock on the SESSION itself. *Do not use synchronized(this).*

- While thread safe, the SingleThreadModel is deprecated and not a good idea...
 - It's exactly the same as synchronizing the service() method.
 - It demolishes concurrency without actually protecting the attributes.
 - It offers nothing but poor performance.
 - Only works if your web app consists of a single servlet!!!
 - If implemented, the Container may create one instance for each request.
 - Unlike normal servlets where the container will create no more than one instance per JVM.
- **Only request and local variables are thread safe (that includes method parameters)!!!**
 - *Not even servlet member instance variables are thread safe.*

Request Attributes and Request Dispatching

- Use request attributes when you want some other part of the application to take over all or part of the request.
- The RequestDispatcher interface has only two methods...
 - forward(ServletRequest, ServletResponse)
 - This method *permanently* transfers control.
 - include(ServletRequest, ServletResponse)
 - This method *temporarily* transfers control.
- There are two ways to get the RequestDispatcher.
 - From the ServletRequest...
 - RequestDispatcher dispatcher = request.getRequestDispatcher("result.jsp");
 - From the ServletContext...
 - RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/result.jsp");
 - **Note:** When getting the RequestDispatcher from the ServletContext, you must ALWAYS start the path with a FORWARD SLASH - it cannot be relative!!!
- You cannot forward the request if you've already committed the response (sent it to the client).
 - Any attempts to do so will render an IllegalStateException.
 - The following code will NOT work...


```
public void doGet(...) {
    ...
    OutputStream outputStream = response.getOutputStream();
    while (...) {
        outputStream.write(...);
    }
    outputStream.flush(); // BAD! Commits the response before forwarding!!!
    RequestDispatcher dispatcher = request.getRequestDispatcher("result.jsp");
    dispatcher.forward(request, response);
    outputStream.close();
    ...
}
```

CHAPTER 6 (Session Management)

Sessions

- An HttpSession object can hold conversational state across *multiple requests* from the *same client*. In other words, *it persists for an entire session with a specific client*.
- The Container uses session IDs to remember clients...
 - On the client's first request...
 - The Container generates a unique session ID.
 - Then gives it back to the client along with the response.
 - The client then sends back the session ID with each subsequent request.
 - With each request, the Container...

- Sees the ID.
- Finds the matching session.
- And then associates that session with the request.
- Sessions are most typically accessed through the request object; however, it is also possible to access a session object through one of the session listeners (HttpSessionEvent or HttpSessionBindingEvent)...
 - Request


```
HttpSession session = request.getSession();
```
 - Session Listener


```
public void sessionCreated(HttpSessionEvent event) {
    HttpSession session = event.getSession();
}
```
- The getSession() method returns a session regardless of whether there's a pre-existing session.
 - The only way to tell if that session is new is to ask the session...
 - request.getSession().isNew();
 - **Note:** This method will ONLY return true if the client has *not yet responded* with the session ID. If by chance the browser has cookies disabled, then this method will ALWAYS return true (since the client will never respond with the session ID).
- There is an overloaded getSession(boolean) method for accessing only a previously created session. This method will return either null or a pre-existing HttpSession. It will NOT create a new session like its counterpart, getSession().
 - Pass false to get a pre-existing session...
 - request.getSession(false);
 - **Note:** Passing true will act exactly the same as the convenience method, getSession().
- Session objects take up resources so don't keep sessions longer than necessary.
- In addition to the get, set, and remove attribute methods, here are a few other key methods of HttpSession...

	Function	Use
getCreationTime()	Returns the time the session was created.	Use to find out how old the session is. You might want to restrict certain sessions to a fixed length of time. Like, for example, only providing ten minutes to complete a form.
getLastAccessedTime()	Returns the last time the Container got a request with this session ID (in milliseconds).	Use to find out when the client last accessed this session. You might, for example, decide to invalidate the session or send an email inviting the user to come back after a certain amount of time.
getMaxInactiveInterval()	Returns the maximum time (in seconds) that is allowed between client requests for this session.	Use to find out how long this session can be inactive and still be alive. You could use this to judge how much more time an inactive client has before the session will be invalidated.
setMaxInactiveInterval()	Specifies the maximum time (in seconds) that you want to allow between client requests for this session. Note that passing an argument of zero, as in <code>setMaxInactiveInterval(0)</code> , has the same effect as the <code>invalidate()</code> method. It	Use to cause the session to be destroyed after a certain amount of time has passed without the client making any requests for this session. This is one way to reduce the amount of stale sessions sitting on the server.

	immediately ends that particular session.	
invalidate()	Ends the session. This includes unbinding all session attributes currently stored in that particular session. <i>Any attempt to call a session method on an already invalidated session will result in an IllegalStateException runtime exception.</i>	Use to kill a session if the client has been inactive or if you KNOW the session is over. The session instance itself might be recycled by the Container, but we don't care. Invalidate means the session ID no longer exists, and the attributes are removed from the session object.

- There are three ways a session can die...
 - It times out.
 - You call the invalidate() method on the session object.
 - The application goes down (crashes or is undeployed).
- Configuring the timeout in web.xml has virtually the same effect as calling the setMaxInactiveInterval() method on every session that's created. The main difference is that the deployment descriptor sets the timeout for *EVERY session* while the setMaxInactiveInterval() method sets the timeout only for *that PARTICULAR session*. In addition, the deployment descriptor takes its value in minutes while the setMaxInactiveInterval() method takes its argument in seconds.
 - Deployment Descriptor

```
<web-app...>
  <session-config>
    <session-timeout>20</session-timeout>
  </session-config>
</web-app>
```

Note: A value of -1 will cause the session to never expire.
 - Session Method

```
session.setMaxInactiveInterval(20*60); // Expires after 20 minutes
```

Session Migration

- In a clustered environment, the Container might do *load balancing* by taking client requests and sending them out to different JVMs (potentially on different physical boxes). In other words, the app is in multiple places. That means each time the same client makes a request, the request could end up going to a *different* instance of the same servlet in a *different* JVM.
 - So what happens to things like ServletContext, ServletConfig, and HttpSession objects?
 - There is only one ServletContext *per JVM*.
 - There is only one ServletConfig *per servlet, per JVM*.
 - Remember that a ServletConfig is stored in a Servlet that is stored in a ServletContext.
 - **Only HttpSession objects (and their attributes) move from one JVM to another - everything else is DUPLICATED across multiple JVMs.**
 - There is only ONE HttpSession object for a given session ID *per web app*, regardless of how many JVMs the app is distributed across.
 - Sessions live in only one place at any given moment. The same session ID for a given web app will never appear in two different JVMs at the same time.
 - How an app server vendor handles clustering and web app distribution varies with each vendor.
 - There's *no guarantee* in the J2EE spec that a vendor has to support distributed apps.
 - If, however, the vendor does support distributed apps, the J2EE spec only *requires* that Containers migrate sessions (and their attributes) across multiple JVMs.
 - While containers are *only required* to migrate *Serializable attributes*, they are *not required* to use Serialization.
 - To be safe, ALWAYS make your attribute class types Serializable.
 - If you choose not to make them Serializable...
 - Have your attribute object classes implement HttpSessionActivationListener and use the activation/passivation callbacks to work around it.

Cookies

- Cookies are what the Container and client use to send session IDs back and forth to each other.
- The Container does virtually all the cookie work for you...
 - Once you tell the Container you want to start a session (conversation)...
 - The Container...
 - Generates the session ID.
 - Creates a new Cookie object.
 - Puts the session ID into the cookie.
 - Sets the cookie as part of the response.
 - Sends the response back to the client.
 - On subsequent requests, the Container...
 - Gets the session ID from the cookie in the request.
 - Matches the session ID with an existing session.
 - And then associates that session with the current request.
- Sending the session cookie in the response is exactly the same as getting the session ID from the request. The Container does all the work for you. It uses the following logic...

```
if (The request includes a session ID cookie) {  
    Find the existing session matching that ID  
} else {  
    Create a new session  
}
```
- You can use cookies to exchange name/value String pairs between the server and the client.
- The server sends the cookie to the client, and the client sends it back with each subsequent request.
 - The server first sends this in the HTTP response header...
 - Set-Cookie: username=JohnSmith
 - The client returns (sends) this with each HTTP request header...
 - Cookie: username=JohnSmith
- Session cookies vanish when the client's browser quits, but you can tell a cookie to persist on the client even after the browser shuts down.
- While you *can* get cookie-related headers out of the HTTP request and response, DON'T!!!
 - Everything you need to do with cookies has been encapsulated into three Servlet API classes...
 - HttpServletRequest
 - getCookies()
 - HttpServletResponse
 - addCookie()
 - Cookie
 - getDomain() and setDomain(String)
 - getMaxAge() and setMaxAge(int)
 - getPath() and setPath(String)
 - getValue() and setValue(String)
 - getSecure() and setSecure(boolean)
 - String getName()
- Working with Cookie objects...
 - Creating a new Cookie...

```
Cookie cookie = new Cookie("cookieName", "cookieValue");
```

 - Note that there is only ONE constructor and that it takes a name/value pair.
 - Setting how long a cookie will live on the client...

```
cookie.setMaxAge(30*60); // Expires after 30 minutes
```

 - Note that the method argument is specified in seconds. However, setting the max age to -1 will make the cookie immediately disappear when the browser exits. This is what the "JSESSIONID" cookie is set to.
 - Sending the cookie to the client...

- ```

 response.addCookie(cookie);

```
- Getting the cookie(s) from the client request...
 

```

Cookie[] cookies = request.getCookies();
for (int i = 0; i < cookies.length; i++) {
 Cookie cookie = cookies[i];
 if (cookie.getName().equals("cookieName")) {
 String cookieValue = cookie.getValue();
 out.println(cookieValue);
 break;
 }
}

```

    - **Note:** *There is no getCookie(String) method.* You have to get an ARRAY of cookies and then loop over it to find the one you want.
  - **Don't confuse Cookies with HTTP headers...**
    - When you add a header to a response, you pass the name and value **Strings** as arguments...
 

```

response.addHeader("attributeName", "attributeValue");

```
    - But when you add a Cookie to the response, you pass a Cookie **object**. You set the name and value in the Cookie constructor...
 

```

Cookie cookie = new Cookie("cookieName", "cookieValue");
response.addCookie(cookie);

```
    - Also note that there is both a setHeader() and addHeader() method, but only a setCookie() method. *There is no addCookie() method.*

## URL Rewriting

- URL rewriting enables clients and Containers to have conversations (sessions) when the client has disabled cookies.
- URL rewriting simply appends the session ID to the URL.
  - URL + ;jsessionid=12345
    - **Note:** The semicolon separator is vendor-specific.
- URL rewriting kicks in ONLY if cookies fail, and ONLY if you tell the response to ENCODE the URL. The Container will always attempt to use cookies first (except for the first response) and then URL rewriting. Once again, **URL rewriting will not work if the URLs are not encoded.**
  - response.encodeURL("/file.jsp")
- On the first response to the client the Container doesn't yet know if the client will accept cookies so the **Container actually uses BOTH cookies and URL rewriting with that first response.**
  - The Container will know on the second request if the client accepts cookies because there will be a session ID cookie in the request header.
- There is a special URL encoding method for redirecting the request to a different URL that will also pass along the session ID.
  - response.encodeRedirectURL("/file.jsp")
- Remember that URL encoding is handled by the RESPONSE object (not the Request, etc.).
- There are only two places a "jsessionid" belongs...
  - Inside a cookie header
    - Cookie: JSESSIONID=12345
  - Appended to the end of a URL as "extra info"
    - http://www.domain.com:80/folder/subfolder/file.jsp;jsessionid=12345
  - **Note:** You should NEVER use "jsessionid" yourself. Do NOT create your own custom "jsessionid" header or use "jsessionid" as a request parameter.
- There's no way to get automatic URL rewriting with your static pages, so if you depend on sessions, you must use dynamically generated pages (JSP, servlet, etc.).

## CHAPTER 7 (JSP)

## JSPs Are Servlets

- The Container takes your JSP file (example.jsp)...
  - TRANSLATES it into a Java servlet SOURCE file (Example\_jsp.java).
    - This is where JSP syntax errors are caught.
  - COMPILES that into a Java servlet CLASS file (Example\_jsp.class).
    - This is where Java language/syntax errors are caught.
  - From this point, treats it like any other servlet...
    - LOADS the newly generated servlet class.
    - INSTANTIATES the servlet.
      - Container calls the init() method who in turn calls the jsplnit() method.
    - CREATES A NEW THREAD for each client request.
      - Container calls the service() method who in turn calls the \_jspService() method.
  - The translation and compilation steps happen ONLY ONCE in the JSP's life. Once it's translated and compiled, it's just like any other servlet. And just like any other servlet, once that servlet has been loaded and initialized, the only thing that happens at *request time* is creation or allocation of a thread for the service method.

## Scriptlets

- *Scriptlets* allow you to write plain old Java code in the JSPs.
- Simply include your Java code within `<%...%>` tags.
  - Remember to end each line with a semicolon as you would normal Java code.
- The Container puts all scriptlet code into the service() method when translated from JSP to servlet.
  - That means variables declared in scriptlets are always LOCAL variables.

## Directives

- A *directive* is a way for you to give special instructions to the Container at *PAGE TRANSLATION* time.
- Directives are signified using `<%@...%>`.
- Directives come in three flavors...
  - Page
    - Defines *page-specific* properties...
      - Character Encoding
      - Content Type
      - Session Object (implicit or not)
    - There are 13 page attributes...
      - Only the following six are on the exam...
        - import
          - Defines the Java import statements that'll be added to the generated servlet.
          - The following imports are implicit (included by default)...
            - java.lang
            - javax.servlet
            - javax.servlet.http
            - javax.servlet.jsp
        - isThreadSafe
          - Defines whether the generated servlet needs to implement SingleThreadModel.
          - Only set to "false" if you want the generated servlet to use SingleThreadModel.
          - The default value is "true" (Does NOT implement SingleThreadModel).
        - contentType
          - Defines the MIME type (and optional character encoding) for the JSP response.
        - isELIgnored
          - Defines whether the EL expressions are ignored when this page is *translated*.
        - isErrorPage
          - Defines whether the current page represents another JSP's error page.
          - The default value is "false".

- If set to "true", then the page has access to the implicit exception object; otherwise the JSP does NOT have access to that object.
- `errorPage`
  - Defines the URL to the resource to which uncaught Throwables should be sent.
  - The JSP page to which all the URLs point must have `isErrorPage="true"`.
- The remaining seven are NOT on the exam...
  - `language`
    - Defines the scripting language used in scriptlets, expressions, and declarations.
    - Right now, "java" is the only possible value (and is set by default).
  - `extends`
    - Defines the superclass of the class this JSP will become.
    - This overrides the class hierarchy provided by the Container.
    - This is rarely used.
  - `session`
    - Defines whether the page will have an implicit "session" object.
    - The default is "true".
  - `buffer`
    - Defines how buffering is handled by the implicit "out" object (reference to the `JspWriter`).
  - `autoFlush`
    - Defines whether the buffered output is flushed automatically.
    - The default value is "true".
  - `info`
    - Defines a String that gets put into the translated page, just so you can get it using the generated servlet's inherited `getServletInfo()` method.
  - `pageEncoding`
    - Defines the character encoding for the JSP.
    - The default is "ISO-8859-1" (unless the `contentType` attribute already defines a character encoding or the page uses XML Document syntax).
- `Include`
  - Defines text and code that gets added into the current page at *TRANSLATION* time.
- `Taglib`
  - Defines tag libraries available to the JSP.
- Import examples...
  - `<%@ page import="com.example.*" %>`
    - Notice that there is no semicolon at the end of the statement.
  - `<%@ page import="com.example.*, java.util.*" %>`
    - Use a comma to separate packages.
    - Quotes go around the entire list of packages.
    - There is no semicolon at the end of the statement.

## Expressions

- *Expressions* are shorthand that *automatically PRINT* whatever is between its tags.
- Expressions are signified using `<%=...%>`.
- Never end an expression with a semicolon!!!
- Expressions become the ARGUMENT to an `out.print()` method. In other words, the Container takes *everything* typed between the opening and closing expression tags and uses it as the argument passed to an `out.print()` method.
- Do NOT use methods with void return types within expressions.
  - You will get an error because the Container knows there won't be anything to return.
- Expressions cannot be used to declare variables.
  - Don't forget that expressions are ARGUMENTS to `out.print()` methods.
- Expressions are moved into the `service()` method when translated from JSP to servlet.

## Declarations

- Declarations are for *DECLARING MEMBERS* of the generated servlet class.
  - That means both variables and methods are added *OUTSIDE* the service method.
  - Use for declaring both static and instance variables and methods.
    - While it is possible to declare inner classes, it is seldom used for that.
- Declarations are signified using `<%!...%>`.

## Comments

- HTML Comments
  - Signified with `<!--...-->`.
  - The Container passes these comments straight on to the client, where the browser interprets them as HTML comments. The commented out blocks still display in the source code.
- JSP Comments
  - Signified with `<%--...--%>`.
  - The Container completely strips out anything between these comment tags when translating the JSP to a servlet. Nothing is shown in the generated source code.

## Implicit Objects

- The following are implicit objects...
  - out (JspWriter)
  - request (HttpServletRequest)
    - CANNOT be accessed from a directive or declaration.
  - response (HttpServletResponse)
  - session (HttpSession)
  - application (ServletContext)
    - Note that there is no such thing as *context* scope. The *application* scope is what would be considered context scope if there were such a scope.
      - context = application
  - config (ServletConfig)
  - exception (JspException)
    - Only available on designated "error" pages.
  - pageContext (PageContext)
    - Encapsulates other implicit objects.
      - Every other implicit object can be accessed through this object.
      - You can use the pageContext to get attributes from any scope.
        - The methods that work with other scopes take an int (constant) argument to indicate the scope.
          - APPLICATION\_SCOPE
          - SESSION\_SCOPE
          - REQUEST\_SCOPE
          - PAGE\_SCOPE
        - When searching for attributes using the findAttribute() method, the search will begin with the most restrictive and work its way up until it finds a match. The order of most restrictive to least restrictive is as follows...
          - Page
          - Request
          - Session
          - Application
      - Stores page-scoped attributes.
    - page (**Object**)
      - Typically only used for custom tag development.

## Lifecycle Methods

- The three key methods of a JSP...

- `jspInit()`
  - This method is called from the servlet's `init()` method.
  - This method CAN be overridden.
- `jspDestroy()`
  - This method is called from the servlet's `destroy()` method.
  - This method CAN be overridden.
- `_jspService()`
  - This method is called from the servlet's `service()` method which means that it runs in a SEPARATE THREAD for EACH REQUEST.
  - This method **CANNOT** be overridden.
  - Note that this method begins with an underscore - unlike the others. Think of the underscore as indicating something you cannot override.

## Initializing JSPs

- You configure servlet init params for your JSP virtually the same way you configure them for a normal servlet. The only difference is that you have to add a `<jsp-file>` element within the `<servlet>` tag.

```
<web-app...>
 <servlet>
 <servlet-name>Example</servlet-name>
 <jsp-file>/example.jsp</jsp-file> ← In place of the <servlet-class> element!
 <init-param>
 <param-name>parameterName</param-name>
 <param-value>parameterValue</param-value>
 </init-param>
 </servlet>
</web-app>
```

- Overriding the `jspInit()` method...
  - Because the `jspInit()` method is called after the `init()` method runs, the `jspInit()` method has access to the `ServletConfig` and `ServletContext` objects. That means that you CAN call `getServletConfig()` and `getServletContext()` from within the `jspInit()` method.

```
<%!
 public void jspInit() {
 ServletConfig config = getServletConfig();
 String parameterValue = config.getInitParameter("parameterName ");
 ServletContext context = getServletContext();
 context.setAttribute("parameterName", parameterValue);
 }
%>
```

## EL (Expression Language)

- The purpose of the EL is to offer a simpler way to *invoke* Java code that exists *outside* the JSP - typically in a `JavaBean` or `TagHandler` class.
- EL expressions are ALWAYS enclosed in curly braces, and prefixed with a dollar (\$) sign.
  - The following two statements are equivalent...
    - `#{applicationScope.attributeName}`
    - `<%= application.getAttribute("attributeName") %>`
- To enforce the use of EL over scripting elements (scriptlets, expressions, or declarations), simply add the `<scripting-invalid>` tag to the deployment descriptor.

```
<web-app...>
 <jsp-config>
 <jsp-property-group>
 <url-pattern>*.jsp</url-pattern>
 <scripting-invalid>true</scripting-invalid>
 </jsp-property-group>
```

```
</jsp-config>
</web-app>
```

- **Note:** The following page directive attribute does NOT work. It was considered and then *removed* from the final JSP 2.0 spec.  

```
<%@ page isScriptingEnabled="false" %>
```
- The EL is enabled by default.
  - There are two ways to disable it...
    - Using `<el-ignored>` in the deployment descriptor...

```
<web-app...>
 <jsp-config>
 <jsp-property-group>
 <url-pattern>*.jsp</url-pattern>
 <el-ignored>true</el-ignored>
 </jsp-property-group>
 </jsp-config>
</web-app>
```
    - Using the `isELIgnored` page directive attribute...

```
<%@ page isELIgnored="true" %>
```

      - **Note:** The page directive takes priority over the deployment descriptor. Also be aware of the naming inconsistency.

## CHAPTER 8 (Scriptless JSP)

### Standard Actions

- There are only three STANDARD actions...
  - `<jsp:useBean>`
  - `<jsp:getProperty>`
  - `<jsp:setProperty>`
- Standard actions can be used to create scriptless JSPs...
  - First tell the Container which JavaBean you're going to use (this declares AND initializes)...

```
<jsp:useBean id="person" class="com.example.Person" scope="request" />
```

    - **id** - Declares the identifier for the bean object (reference variable name).
    - **class** - Declares the fully qualified class type for the bean object.
    - **scope** - Identifies the attribute scope for this bean object.
  - Then access the property of the bean you want to display...

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

    - **name** - Identifies the actual bean object. The name must match the "id" specified in its corresponding `<jsp:useBean>` tag.
    - **property** - Identifies the property name (name = getName()).
  - You can also set values if needed...

```
<jsp:setProperty name="person" property="name" value="John" />
```

    - **name** - Identifies the actual bean object. The name must match the "id" specified in its corresponding `<jsp:useBean>` tag.
    - **property** - Identifies the property name (name = setName()).
    - **value** - Specifies the value to set for the property (setName("John")).
- By default, if the `<jsp:useBean>` standard action does not find the JavaBean in the specified scope, it will CREATE a new one.
  - To **CONDITIONALLY** set properties **ONLY** if a new bean is created, simply place the `<jsp:setProperty>` tag(s) into the *body* of the `<jsp:useBean>` tag.

```
<jsp:useBean id="person" class="com.example.Person" scope="request">
 <jsp:setProperty name="person" property="name" value="John" />
</jsp:useBean>
```
- By convention, ALL JavaBeans...
  - **MUST** provide a public, no-arg constructor.

- MUST provide getters and setters for properties (member instance variables).
  - For example, the "name" property must have a corresponding...
    - **getName()**
      - **Note:** Booleans can optionally be accessed using "is" instead of "get" (**isProperty()**).
    - **setName()**
  - In addition, the setter argument and getter return type must be identical.
    - `String getName()`
    - `setName(String name)`
- For use with JSPs, the property type SHOULD be a String or primitive.
  - While still a legal JavaBean if not followed, scripting may be required to access those properties (standard actions will not be enough).
- Polymorphic behavior can be achieved by using the "type" attribute of the <jsp:useBean> tag...
 

```
<jsp:useBean id="person" type="com.example.Person" class="com.example.Employee" />
```

  - **type** - Declares the *reference variable* type (Interface, Abstract Class, Concrete Class).
  - **class** - Declares the *actual object variable* type (Concrete Class).
  - The generated servlet translates the action to...
 

```
Person person = new Employee() ;
```
  - Important rules...
    - If type is used without class, the bean MUST already exist or it will throw an `InstantiationException` exception.
      - In general, you will never use type without class, UNLESS you are certain that the bean is already stored as an attribute (with the right scope and id).
    - If class is used (with or without type) the class must NOT be abstract (or an interface), and must have a public no-arg constructor.
- If not specified, the "scope" attribute defaults to "page" (the most narrow scope).
- You can send a request parameter straight into a bean, without scripting or without going through a servlet, just by using the "param" attribute. This attribute lets you set the value of a bean property to the value of a request parameter - just by naming the request parameter.
  - request.jsp
 

```
<html>
 <form action="result.jsp">
 Name: <input type="text" name="username" />
 <input type="submit" />
 </form>
</html>
```
  - response.jsp
 

```
<jsp:useBean id="person" type="com.example.Person" class="com.example.Employee">
 <jsp:setProperty name="person" property="name" param="username" />
</jsp:useBean>
```

    - **Note:** The param value "username" comes from the *name* attribute of the form's input field.
- If the request parameter name matches the bean property name, you don't need to specify a value in the <jsp:setProperty> tag for that property.
  - request.jsp
 

```
<html>
 <form action="result.jsp">
 Name: <input type="text" name="name" />
 <input type="submit" />
 </form>
</html>
```
  - response.jsp
 

```
<jsp:useBean id="person" type="com.example.Person" class="com.example.Employee">
 <jsp:setProperty name="person" property="name" />
</jsp:useBean>
```

    - **Note:** There was no need to specify the param because the parameter name for the field matches the bean property (name).

- It is also possible to tell the Container to iterate through the all request parameters to match each bean property.
  - request.jsp
 

```
<html>
 <form action="result.jsp">
 Name: <input type="text" name="name" />
 Employee ID: <input type="text" name="employeeID" />
 <input type="submit" />
 </form>
</html>
```
  - response.jsp
 

```
<jsp:useBean id="person" type="com.example.Person" class="com.example.Employee">
 <jsp:setProperty name="person" property="*" />
</jsp:useBean>
```

    - **Note:** The asterisk (\*) tells the Container that all the request parameters have matching/corresponding bean properties - so get/set them accordingly.

## Standard Action Review

- The <jsp:useBean> standard action defines a variable that holds a reference to either an *existing* bean attribute or, if the bean doesn't already exist, a *new* bean.
- The <jsp:useBean> MUST have an "id" attribute which declares the variable name that'll be used in this JSP to refer to the bean.
- If you don't include a "scope" attribute with <jsp:useBean>, the scope defaults to PAGE scope.
- The "class" attribute is OPTIONAL, and it declares the class type that will be used if a new bean is created. The type must be public, non-abstract (concrete), and have a public no-arg constructor.
- If you put a "type" attribute in <jsp:useBean>, it must be a type to which the bean can be cast.
- If you have a "type" attribute but do NOT have a "class" attribute, the bean must already exist, since you haven't specified the class type that should be instantiated for the new bean.
- The <jsp:useBean> tag can have a body, and anything in the body runs ONLY if a NEW bean is created as a result of <jsp:useBean> (which means that no bean with that "id" was found in the specified (or default) scope).
- The main purpose of the body of <jsp:useBean> is to set the NEW bean's properties, using <jsp:setProperty>.
- <jsp:setProperty> must have a "name" attribute (which will match the "id" from <jsp:useBean>), and a "property" attribute. The "property" attribute must be either an actual property name or the wildcard "\*".
- If you don't include a "value" attribute, the Container will set the property value only if there's a request parameter with a name that MATCHES the property name. If you use the wildcard (\*) for the "property" attribute, the Container will set the value of ALL properties that have a matching request parameter name (other properties won't be affected).
- If the request parameter name is different from the property name, but you want to set the value of the property equal to the request parameter value, you can use the "param" attribute in the <jsp:setProperty> tag.
- If you specify a "type" attribute in <jsp:useBean>, you can set properties in <jsp:setProperty> ONLY on properties of the "type", but NOT on properties that exist only in the actual "class" type (in other words, polymorphism and normal Java type rules apply).
- Property values can be Strings or primitives, and the <jsp:setProperty> standard action will do the conversion automatically.

## Expression Language

- Bean tags convert PRIMITIVE properties automatically.
  - JavaBean properties *can be anything*, but if they're Strings or primitives, all the coercing is done for you. You don't have to do the parsing and conversion yourself.
  - Automatic String-to-primitive conversion does NOT work if you use scripting!!! It fails even if an expression is INSIDE the <jsp:setProperty> tag.

- These work...
  - `<jsp:setProperty name="person" property="*" />`
  - `<jsp:setProperty name="person" property="employeeID" />`
  - `<jsp:setProperty name="person" property="employeeID" value="123" />`
  - `<jsp:setProperty name="person" property="employeeID" param="payrollID" />`
- This does NOT work...
  - `<jsp:setProperty name="person" property="employeeID" value="<%= request.getParameter("employeeID") %>" />`
- While bean tags convert Strings and primitives automatically, they do not handle composite objects out of the box. You have to use the JSP EL for that.
- The EL makes it easy to print nested properties (properties of properties) - composite objects.
  - My dog's name is **`#{person.dog.name}`**
    - **Note:** *Person* has a *Dog* that has a *name*.
- The EL doesn't always follow Java's language/syntax rules so think of the EL as a way of accessing Java objects *without using Java*.
- EL expressions are ALWAYS within curly braces, and prefixed with the dollar sign.
  - `#{firstThing.secondThing}`
    - The first named variable in the expression is either an *implicit object* or an *attribute*.
      - EL Implicit Objects
        - `pageScope` (Map of page-scoped attributes)
        - `requestScope` (Map of request-scoped **attributes**)
        - `sessionScope` (Map of session-scoped attributes)
        - `applicationScope` (Map of application-scoped attributes)
        - `param` (Map of request **parameters**)
        - `paramValues`
        - `header` (Map of request headers)
        - `headerValues`
        - `cookie` (Map of cookies)
        - `initParam` (Map of the **CONTEXT init parameters**, NOT servlet init parameters)
        - `pageContext`
          - **Note:** With the exception of `pageContext`, the EL implicit objects are all different than those available to JSP scripting. Also be aware that `pageContext` is the only one that is NOT a Map. It's an actual reference to the `pageContext` OBJECT. By the way, the `pageContext` is a `JavaBean`.
      - Attribute
        - In page scope
        - In request scope
        - In session scope
        - In application scope
          - **Note:** If the first thing in the EL expression is an attribute, it can be the name of an attribute stored in any of the four available scopes.
- When the variable is on the left side of the dot, it's either a map (something with keys) or a bean (something with properties). This is true regardless of whether the variable is an implicit object or an attribute.
- The variable on the right side of the dot is ALWAYS either a Map key or a bean property.
  - These variables must follow normal Java naming conventions...
    - Must start with a letter, `_`, or `$`.
    - The first letter must be lowercased.
    - Numbers can be used AFTER the first character.
    - It cannot be a Java keyword.
- The dot operator works only when the thing on the right is a bean property or map key for the thing on the left. It does NOT work if the thing on the right is an array or a list.
- The bracket `[ ]` operator works the same as the dot convention, but also allows you to work with things like arrays and lists.

- `${firstThing[secondThing]}`
  - If the expression has a variable followed by a bracket [], the left-hand variable can be a map, bean, list, or array.
  - If the thing inside the brackets is a String literal (meaning in quotes), it can be a map key, or a bean property, or an INDEX into a list or array.
- For arrays and lists, a String index is coerced to an int. Remember the EL is not Java (syntax).
  - The following are equivalent...
    - `${myArray[0]}`
    - `${myArray["0"]}`
      - **Note:** `${myArray["zero"]}` would not work because "zero" cannot be coerced to an int.
- For beans and maps you can use either the dot or bracket operators.
  - The following are equivalent...
    - `${myMap.keyName}`
    - `${myMap["keyName"]}`
- If there are no quotes inside the brackets, the Container evaluates what's inside the brackets by searching for an attribute bound under that name, and substitutes the value of the attribute. If there is an implicit object with the same name, the implicit object will always be used.

```
Map map = new HashMap();
map.put("mapKey", "mapValue");
request.setAttribute("map", map);
request.setAttribute("requestKey", "mapKey");
${map[requestKey]}
${map["mapKey"]}
```

  - **Note:** The "requestKey" in the request scope's value is "mapKey" (the actual key of the map); therefore, the EL evaluates the `${map[requestKey]}` to `${map["mapKey"]}`. The above code would have NOT worked had "requestKey" been a String literal.
- You can use nested expressions inside brackets. Like the use of nested parenthesis in Java, the expression is evaluated from the inside out. The following for example is legal...
  - `${map[keyArray[0]]}`
- With beans and maps, you can use the dot operator, but only if the thing you type after the dot is a legal Java identifier. In other words, if you wouldn't use it for a variable name in your Java code, DON'T put it after a dot.
- The requestScope is NOT the request object. The implicit requestScope is just a MAP of the request scope attributes, NOT the request scope itself.
  - Use requestScope to get request ATTRIBUTES, not request PROPERTIES/METHODS. For request properties, you need to go through pageContext.
    - `${pageContext.request.method}`
      - **Note:** This is the same as `<%= request.getMethod() %>`. The pageContext returns the actual request scope object. That object contains a property called "method".
- Don't confuse the Map scope objects with the objects to which the attributes are bound.
- The EL **initParam** is **NOT** for params configured using **<init-param>**!
  - Servlet init params are configured using `<init-param>` while context params use `<context-param>` but *the EL implicit "initParam" is for CONTEXT params.*
- EL functions are a simple way to call static methods in POJOs.
  - Here are the steps...
    - Write a Java class with a public static method.
      - The method must be PUBLIC and STATIC.
      - It should (but isn't required) have a non-void return type.
    - Write a TLD file.
      - The TLD provides a mapping between the Java class that *defines* the function and the JSP that *calls* the function. The method name used in the EL doesn't have to be the same as the actual method name in the Java class.
      - Put the TLD file somewhere under WEB-INF, and make sure that the taglib directive in the JSP includes a URI attribute that matches the `<uri>` element in the TLD.

- Put a taglib directive in your JSP.
  - This allows you to define the namespace. It's kind of like giving all your functions fully qualified names.
- Use the EL to invoke the function.
  - Simply call the function from an expression using the `${prefix:methodName()}` syntax.
- While regular scriptlet expressions MUST return something, EL functions DO NOT have to return anything (although they should).
- Yes, EL functions can accept arguments! *Just remember to specify the fully qualified class name (unless it's a primitive) for each argument in the TLD.*
- The METHOD name does NOT have to match the FUNCTION name defined in the TLD. What you use in the EL to invoke the function must match the <name> element in the TLD's <function> declaration.
  - Also note that everything in the <function> tag has the word "function" in its tag EXCEPT for the <name> tag. There is NO <function-name> ONLY <name>.
- EL operators...
  - Arithmetic
    - + (Addition)
    - - (Subtraction)
    - \* (Multiplication)
    - / and **div** (Division)
      - With EL you CAN divide by zero (you get infinity NOT an error).
    - % and **mod** (Remainder)
      - You CANNOT use the remainder operator against zero (you'll get an exception).
  - Logical
    - && and **and** (And)
    - || and **or** (Or)
    - ! and **not** (Not)
  - Relational
    - == and **eq** (Equals)
      - There is NO single equals (=) in EL.
    - != and **ne** (Not Equals)
    - < and **lt** (Less Than)
    - > and **gt** (Greater Than)
    - <= and **le** (Less Than Or Equal To)
    - >= and **ge** (Greater Than Or Equal To)
  - Other
    - **true** (Boolean Literal)
    - **false** (Boolean Literal)
    - **null** (Null)
    - **instanceof** (Reserved future keyword)
    - **empty** (Checks for null, empty Strings, or empty collections)
- EL is null-friendly. It handles the unknown or null values so that the page still displays, even if it can't find an attribute/property/key with the name in the expression.
  - In arithmetic, EL treats the null value as "zero".
  - In logical expressions, EL treats the null value as "false".

## Expression Language Review

- EL expressions are always within curly braces, and prefixed with a dollar sign as in `${expression}`.
- The first named variable in the expression is either an implicit object or an attribute in one of the four scopes (page, request, session, or application).
- The dot operator lets you access values by using a map key or a bean property name, for example `${foo.bar}` gives you the value of bar, where bar is the name of the map key into the map foo, or bar is the property of bean foo. *Whatever comes to the right of the dot operator must follow normal Java*

*naming rules for identities!* In other words, it must start with a letter, underscore, or dollar sign, and can include numbers after the first character.

- You can NEVER put anything to the right of the dot that wouldn't be legal as a Java identifier as in `#{foo.1}`.
- The `[]` operator is more powerful than the dot, because it lets you access arrays and lists, and you can put other expressions including named variables within the brackets, and you can nest them to any level you can stand.
- For example, if `musicList` is an `ArrayList`, you can access the first value in the list by saying `#{musicList[0]}` or `#{musicList["0"]}`. EL doesn't care if you put quotes around the list index.
- If what's inside the brackets is not in quotes, the Container EVALUATES it. If it is in quotes, and it's not an index into an array or List, the Container sees it as the literal name of a property or key.
- All but one of the EL implicit objects are maps. From the map implicit objects you can get attributes from any of the four scopes, request parameter values, header values, cookie values, and context init parameters. The non-map (JavaBean) implicit object is `pageContext`, which is a reference to the `PageContext` object.
- *Don't confuse the implicit EL scope objects (maps of attributes) with the objects to which the attributes are bound.* In other words, don't confuse the `requestScope` implicit object with the actual JSP implicit request object. The only way to access the request object is by going through the `pageContext` implicit object.
- EL functions allow you to call a public static method in a POJO. The function name does not have to match the actual method name. For example, `#{foo:staticMethod()}` does not necessarily mean that there must be a method in the POJO with that exact same name (although there can be).
- The function name (`staticMethodAlias()`) is mapped to a real static method (`staticMethod()`) using a TLD file. Declare a function using the `<function>` element, including the `<name>` of the function (`staticMethodAlias()`), the fully qualified `<function-class>`, and the `<function-signature>` which includes the return type as well as the method name and argument list.
- To use a function in a JSP, you must declare the namespace using a `taglib` directive. Put a prefix attribute in the `taglib` directive to tell the Container the TLD in which the function you're calling can be found.
  - `<%@ taglib prefix="my" uri="/WEB-INF/foo.tld" %>`

### **Include Directive vs. `<jsp:include>` Standard Action**

- The include *DIRECTIVE* tells the Container to *COPY* everything in the *INCLUDED* file and *PASTE* it into the target file. It is exactly the same as if you had done this *manually except it happens at TRANSLATION time*.
  - The include directive inserts the *SOURCE* of the included file at translation time (not runtime). In other words, the source of the included file becomes *PART* of the page with the including directive.
- While the `<jsp:include>` standard action *APPEARS* to do the same thing as the include directive, they are actually quite different.
  - The `<jsp:include>` inserts the *RESPONSE* of the included file at *RUNTIME* (not translation time).
  - The key to the `<jsp:include>` is that the Container is creating a `RequestDispatcher` from the `page` attribute and applying the `include()` method. The dispatched/included JSP executes against the same request and response objects, within the same thread.
- The attribute names are different for the *include directive* and `<jsp:include>` *standard action*.
  - `<%@ include file="header.jsp" %>`
  - `<jsp:include page="header.jsp" />`
    - In order to keep the attributes straight, make the clear association that at **translation** time the Container cares about **files** while at **runtime** the Container cares about **pages**.
      - Translation = Files
      - Runtime = Pages
- An included page *CANNOT* change the response status code or set headers. Attempting to do so will *NOT* throw an error but just fail silently.
- Both includes are position-sensitive in the JSP.

- **Note:** The include directive is the ONLY directive whose position/placement actually matters.
- Do not put opening and closing HTML and BODY tags within your reusable pieces.
  - Design and write your layout template chunks (includes) assuming they will be included in some OTHER page.
- The **.jspx** extension is a convention for JSP segments (they used to be known as "fragments").
  - A segment or fragment is a JSP page that is not intended to stand alone but in fact is meant to be included as part of another page.
- Parameters CAN be passed to `<jsp:include>` standard actions (NOT include directives).
  - Template
 

```
<jsp:include page="header.jspf">
 <jsp:param name="subTitle" value="SubTitle" />
</jsp:include>
```
  - Include
 

```
#{param.subTitle}
```

## Include Review

- You can build a page with reusable components using one of the two include mechanisms - the include *directive* or the `<jsp:include>` *standard action*.
- The include directive does the include at TRANSLATION time, only ONCE. So the include directive is considered the appropriate mechanism for including content that isn't likely to change after deployment.
- The include directive essentially copies everything from within the included file and then pastes it into the page with the include. The Container combines all the included files and compiles just ONE file for the generated servlet. At runtime, the page with the include runs exactly as though you had typed all the source into ONE file yourself.
- The `<jsp:include>` standard action includes the RESPONSE of the included page into the original page at RUNTIME. So the include standard action is considered appropriate for including content that may be updated after deployment, while the include directive is not.
- EITHER mechanism can include dynamic elements as well as static HTML pages.
- The include directive is the only position-sensitive directive; the included content is inserted into the page at the exact location of the directive.
- The attributes for the include directive and include standard action are inconsistently named - the directive uses "file" as the attribute while the standard action uses a "page" attribute.
- In your reusable components, be sure to strip out the opening and closing HTML tags. Otherwise, the generated output will have nested opening and closing HTML tags, which not all browsers can handle. Design and construct your reusable pieces knowing that they'll be included/inserted into something else.
- You can customize an included file by setting (or replacing) a request parameter using the `<jsp:param>` standard action inside the body of a `<jsp:include>` tag.
- The `<jsp:param>` can also be used inside the `<jsp:forward>` tag as well.
- The ONLY places where a `<jsp:param>` makes sense are within a `<jsp:include>` or `<jsp:forward>` standard action.
- If the param name used in `<jsp:param>` already has a value as a request parameter, the new value will OVERWRITE the previous one. Otherwise, a new request parameter is added to the request.
- The included resource has some limitations; it cannot change the response status code or set headers.
- The `<jsp:forward>` standard action forwards the request (just like using the RequestDispatcher) to another resource from the same web app.
- When a forward happens, the response buffer is cleared first! The resource to which the request was forwarded gets to start with a clean output. So anything written to the response before the forward will be thrown away.
- If you commit the response before the forward (by calling `out.flush()`, for example), the client will be sent whatever was flushed, but that's it. The forward won't happen, and the rest of the original page won't be processed.

## The <jsp:forward> Standard Action

- You CAN forward from one JSP to another JSP, servlet, or other resource.
- With <jsp:forward>, the buffer is cleared BEFORE the forward. In other words, the resource to which the request is forwarded starts with a clean response buffer - anything written to the response before the forward happens is thrown out.
  - NOTHING you write BEFORE the forward will appear if the forward happens.
- **Never do a flush-and-forward.**
  - If you commit the response BEFORE the forward, by doing something like **out.flush()**, an `IllegalStateException` will be thrown and fail silently (nobody will see it or even know it happened). The user will see what was already committed and the forward will NOT happen as intended.

## CHAPTER 9 (JSTL)

### Installing JSTL 1.1

- The JSTL 1.1 is NOT part of the JSP 2.0 specification. Having access to the Servlet and JSP APIs doesn't mean you have access to JSTL.
- Before you can use JSTL, you need to have a copy of the "jstl.jar" file in your WEB-INF/lib directory.

### <c:forEach>

- The <c:forEach> tag allows you to iterate over collections (arrays, collections, maps, and comma-delimited Strings).
- The following example loops through the entire collection (the "movieList" attribute) and *prints each element* in a new row within the containing table.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
...
<table>
 <c:forEach var="movie" items="{movieList}" varStatus="counter">
 <tr>
 <td>
 Number: ${counter.count}

 Movie: ${movie}
 </td>
 </tr>
 </c:forEach>
</table>
```

- The <c:forEach> tag contains the following attributes...
  - **var** = The variable that holds EACH ELEMENT in the collection. Its value changes with each iteration. The scope of the variable is confined to the tag (it cannot be used outside the tag).
  - **items** = The actual thing to loop over (array, collection, map, or comma-delimited String).
  - **varStatus** = An optional attribute that creates a variable to hold the count of the current iteration (like the "i" in a for loop). The variable holds an instance of `servlet.jsp.jstl.core.LoopTagStatus`.
- <c:forEach> tags can be nested for more complex structures.

### <c:if>

- The <c:if> tag allows you do conditional logic (it's an "if" statement WITHOUT the "else" block).
- The following example only displays the included section IF the user is a member...

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
...
<c:if test="userType eq 'member'">
 <jsp:include page="membersOnly.jsp" />
</c:if>
```

- **Note:** You can use either single or double quotes in tags and EL (i.e. 'member').

## <c:choose>, <c:when>, and <c:otherwise>

- Use <c:when> and <c:otherwise> tags wrapped inside the <c:choose> tag when you need an if/else construct.
- Only one of the statements below will execute...

```
<c:choose>
 <c:when test="{userType eq 'executive'}">
 Executive-specific statement
 </c:when>
 <c:when test="{userType eq 'manager'}">
 Manager-specific statement
 </c:when>
 <c:otherwise>
 Employee-specific statement (Everyone else)
 </c:otherwise>
</c:choose>
```
- **Note:** The <c:otherwise> tag will ONLY run if NONE of the conditions above equate to true. Also, be aware that the <c:choose> tag is NOT required to have an <c:otherwise> tag.

## <c:set>

- The <c:set> tag is like <jsp:setProperty> but much better in that not only can you set the property of a bean, but you can also, for example, add a new Map entry or create new request-scoped attributes.
- <c:set> comes in two flavors (both with and without a body)...
  - **var** - Used for setting ATTRIBUTE variables.
  - **target** - Used for setting BEAN PROPERTIES or MAP VALUES.
- Below are two examples using the var attribute tag with and without body elements...
  - Without body...

```
<c:set var="userType" scope="session" value="" />
```

    - **var** - This attribute is required. If there's NOT a session-scoped attribute named "userType", then this tag creates one (*ONLY if the value attribute is NOT null*).
    - **scope** - The scope attribute is optional.
    - **value** - This attribute is required. It can evaluate to any type of object (not just String). *Be aware that if the value attribute evaluates to null, the variable will be REMOVED.*
  - With body...

```
<c:set var="userType" scope="session">
 Executive, Manager, Employee
</c:set>
```
- The <c:set> tag if used with the target attribute can ONLY be used for Bean properties and Map values. *It CANNOT be used with lists or arrays.*
- Below are two examples using the target attribute with and without body elements...
  - Without body...

```
<c:set target="{petMap}" property="dogName" value="Rover" />
```

    - **target** - This must NOT be null.
    - **property** - The name of the Bean property or Map key.
    - **value** - The value to set.
  - With body...

```
<c:set target="{person}" property="name">
 {foo.name}
</c:set>
```

    - **Note:** The body can be a String or expression. Also, the "target" must evaluate to OBJECT. You don't type in the String "id" name of the bean or Map attribute.
- The target attribute does NOT work the same as var.
  - With the target attribute, you do NOT type in the String literal that represents the name under which the attribute was bound to the page, scope, etc. No, the target attribute needs a value

that resolves to the REAL THING. That means an EL expression, scripting expression, or `<jsp:attribute>`.

- Key points and gotchas with `<c:set>`...
  - You can NEVER have BOTH the "var" and "target" attributes in a `<c:set>`.
  - "Scope" is optional, but if you don't use it, the default is page scope.
  - If the "value" is null, the attribute named by "var" will be removed.
  - If the attribute named by "var" does not exist, it'll be created, but only if "value" is not null.
  - If the "target" expression is null, the Container throws an exception.
  - The "target" is for putting in an expression that resolves to the REAL Object. If you put in a String literal that represents the "id" name of the bean or Map, it won't work, In other words, "target" is NOT for the *attribute NAME* of the bean or Map - it's for the *ACTUAL attribute OBJECT*.
- Whether you use or don't use the body, it is exactly the same (there's no difference).
- If you don't specify the optional "scope", then the Container will search all the scopes in order from smallest to biggest (page, request, session, application (context)). If you use the "var" version without specifying a scope, the Container will search EACH scope. If it does not find the named object, then it will create one in the PAGE scope.
- Remember, `<c:set>` does a remove when you pass in a null value.

### `<c:remove>`

- The `<c:remove>` tag removes attributes from any scope.

```
<c:set var="color" scope="request" value="blue">
 BEFORE: color: ${color}
 <c:remove var="color" scope="request" />
 AFTER: color: ${color}
</c:set>
```

  - **var** - This attribute MUST be a STRING LITERAL. It cannot be an expression.
  - **scope** - This is optional and like always defaults to page scope if not specified.
  - **Note:** The before prints "blue" while the after prints nothing (empty String, NOT null).

### `<c:import>`

- The `<c:import>` tag is the third way to import content.
  - `<%@ include file="header.jsp" %>`
    - **Static:** Adds the content from the value of the **file** attribute to the current page at **translation** time. It CANNOT include anything outside of the Container.
  - `<jsp:include page="header.jsp" />`
    - **Dynamic:** Adds the content from the value of the **page** attribute to the current page at **request** time. It CANNOT include anything outside of the Container.
  - `<c:import url="http://www.domain.com/includes/header.jsp" />`
    - **Dynamic:** Adds the content from the value of the **URL** attribute to the current page, at **request** time. It works a lot like the `<jsp:include>` standard action, but is more powerful and flexible. Unlike the other include mechanisms, this tag CAN include content from OUTSIDE of the Container.
- All three includes use different attribute names...
  - **Directive:** Originally intended for static layout templates so it deals with **files**.
  - **<jsp:include>**: Intended more for dynamic content so it deals with **pages**.
  - **<c:import>**: Deals with **URLs** so that you can go outside the Container.

### `<c:url>`

- The `<c:url>` tag AUTOMATICALLY handles URL rewriting. In other words, it will automatically append the jsessionid if the Container deems it necessary.
- The `<c:url>` tag will also do URL encoding (fix the URL by adding %20, +, etc. where necessary) ONLY if the query string is built using `<c:param>` tags.

```
<c:url var="employeeProfile" value="/profile.jsp">
 <c:param name="firstName" value="${firstName}" />
```

```
<c:param name="lastName" value="{lastName}" />
</c:url>
```

**Note:** The params are appended to the URL as a query string using URL encoding.

### <c:catch>

- The JSTL also provides a mechanism for try-catch constructs...

```
<c:catch var="myException">
 <%
 int divideByZero = (10 / 0);
 out.println("This line will NOT print because of the exception thrown above.");
 %>
</c:catch>
<c:if test="{myException != null}">
 Exception: {myException.message}
</c:if>
```

- **catch** - Use this tag as a consolidated *try-catch* construct. It serves as both the *try* and the *catch*; therefore, when an exception occurs, like in a Java *try* block, *the execution stops and leaves the entire block skipping the remaining statements*. In short, the <c:catch> tag is more like a *try* block than a *catch* block. Be aware that the var defined in the <c:catch> tag is NOT AVAILABLE INSIDE the <c:catch> block. It actually creates a page-scoped attribute named, "myException", that can be accessed BELOW the <c:catch> block (if an exception is thrown).
- **var** - This attribute is optional, but must be used if you wish to access the actual exception thrown. *Remember ONLY officially designated error pages get the implicit exception object.*

### The "Core" Library

- Even though there are five JSTL libraries, only the "core" library is covered on the exam.
  - General
    - <c:out>
    - <c:set>
    - <c:remove>
    - <c:catch>
  - Conditional
    - <c:if>
    - <c:choose>
    - <c:when>
    - <c:otherwise>
  - URL
    - <c:import>
    - <c:url>
    - <c:redirect>
    - <c:param>
  - Iteration
    - <c:forEach>
    - <c:forEachToken>
      - Works like StringTokenizer in that it allows you to iterate over tokens, such as a comma-separated list. Simply pass it the delimiter.

### Error Pages

- To designate a particular page as an error page (so that it has access to the implicit "exception" object), simply add the following directive...
  - <%@ page isErrorPage="true" %>
  - **Note:** ONLY pages with this explicit directive will have access to the "exception" object.
- To have pages forward to that particular error page when exceptions are encountered, simply add the following directive...
  - <%@ page errorPage="/errorPage.jsp" %>

- Using the <error-page> tag in the deployment descriptor (web.xml), you can declare error pages for the entire web app specific to both exception types and HTTP error codes.
  - The Container uses the deployment descriptor's <error-page> configuration by default, but will FAVOR the explicit "errorPage" directive if specified.
- Here's how to declare a catch-all error page in the deployment descriptor based on exception...
 

```
<error-page>
 <exception-type>java.lang.Throwable</exception-type>
 <location>/errorPage.jsp</location>
</error-page>
```

  - **Note:** Like in the try-catch blocks of Java, the Container will always choose the MOST SPECIFIC exception if multiple related exception-error-pages are provided. Also be aware that declaring exception/error pages in the deployment descriptor does NOT give them access to the implicit "exception" object. ONLY pages with the "isErrorPage" directive set to "true" have access to that object.
- Here's how to declare an error page based on an HTTP status code...
 

```
<error-page>
 <error-code>404</error-code>
 <location>/pageNotFound.jsp</location>
</error-page>
```

## TLD

- Everything you need to know in order to use a custom tag is in the TLD.
- The TLD describes two main things...
  - Custom tags
  - EL functions
- TLD example...
 

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..." version="2.0">
 <tlib-version>0.9</tlib-version>
 <short-name>RandomTags</short-name>
 <function>
 <name>rollDice</name>
 <function-class>com.example.DiceRoller</function-class>
 <function-signature>int rollDice()</function-signature>
 </function>
 <uri>randomThings</uri>
</tag>
 <description>Random Advice</description>
 <name>advice</name>
 <tag-class>com.example.AdvisorTagHandler</tag-class>
 <body-content>empty</body-content>
 <attribute>
 <name>user</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
</taglib>
```

  - **<tlib-version>** - Mandatory - Declares the version of the tag library.
  - **<short-name>** - Mandatory - Mainly used for tools to use.
  - **<uri>** - The unique name used in the taglib directive.
    - <%@ taglib prefix="my" uri="randomThings" %>
  - **<description>** - Optional - Describes what the tag does.
  - **<name>** - Required - The actual name of the tag.
    - <my:advice>

- **<tag-class>** - Required - What the Container invokes when the tag is used.
- **<body-content>** - Required - Defines whether the tag accepts content in its body. There are actually four possible values...
  - **empty** - The tag must NOT have a body (the <jsp:attribute> tag does NOT count as a body).
  - **scriptless** - The tag must NOT have scripting elements (scriptlets, scripting expressions, and declarations - in short, anything with <%...%> syntax), but CAN have template text, EL, standard actions, and other custom tags.
  - **tagdependent** - The tag body is treated as plain text, so the EL is NOT evaluated and tags/actions are not triggered.
  - **JSP** - The tag body can have anything that can go inside a JSP.
- **<attribute>** - There will be one for each attribute.
- **<name>** - The name of the attribute.
- **<required>** - Specifies whether the attribute is required. If not specified the default is false.
- **<rtexprvalue>** - Specifies whether the attribute's value can be a runtime expression (not just a String literal). If not specified the default is false.
- There are three kinds of expressions that can be used to pass runtime values to tags...
  - EL Expressions
 

```
<my:advice user="{username}" />
```
  - Scripting Expressions
 

```
<my:advice user='<%= request.getAttribute("username") %>' />
```
  - <jsp:attribute> Standard Actions
 

```
<my:advice>
 <jsp:attribute name="user">{username}</jsp:attribute>
</my:advice>
```

    - **Note:** The <jsp:attribute> lets you put attributes in the BODY of a tag, even when the tag body is explicitly declared "empty" in the TLD. It is simply an ALTERNATE way to define attributes in a tag. If used, there must be only ONE <jsp:attribute> for EACH attribute in the enclosing tag. The "name" attribute of the <jsp:attribute> tag must match the corresponding attribute name of the outer tag for which it is representing.
- *Think of the TLD as the API for custom tags.*
- The taglib **URI is just a name** NOT a location.
  - While it may often look like a path or URL to an actual resource, it is not. It is simply a name!
  - The name in the taglib directive **MUST** match the one in the TLD's <uri> tag.
- Before JSP 2.0, the developer **HAD** to specify a mapping between the <uri> in the TLD and the actual location of the TLD file. The deployment descriptor (web.xml) had to tell the Container where the TLD file with a matching <uri> was located. This is no longer the case, *now the Container automatically builds a map between TLD files and <uri> names*, so that when a JSP invokes a tag, the Container knows exactly where to find the TLD that describes the tag.
  - If you **DO** specify an **EXPLICIT** <taglib-location> in the deployment descriptor, a JSP 2.0 Container will use it; if not, it will **DEFAULT** to searching WEB-INF on its own to find TLDs.
 

```
<web-app ...>
 <jsp-config>
 <taglib>
 <taglib-uri>randomThings</taglib-uri>
 <taglib-location>/WEB-INF/myFunctions.tld</taglib-location>
 </taglib>
 </jsp-config>
</web-app>
```
- The Container will search the following places to find TLD files...
  - Directly inside WEB-INF (at the top level) or one of its sub-directories.
  - Inside the META-INF directory or one of its sub-directories of a JAR file inside the WEB-INF/lib directory
- When using taglibs...
  - Make sure taglib URI names are unique.
  - Do NOT use a prefix that's on a reserved list...

- jsp:
- jsp:include:
- java:
- javax:
- servlet:
- sun:
- sunw:

## CHAPTER 10 (Custom Tags)

### Tag Files

- With tag files, you can invoke reusable content using a custom tag instead of the generic `<jsp:include>` or `<c:import>`. Think of tag files as a kind of "tag handler lite", because they let you create custom tags, without having to write a complicated Java tag handler class.
  - Tag files are really just glorified includes.
- The simplest way to make and use a tag file...
  - Name the include file with the ".tag" or ".tagx" extension.
    - "header.jsp" changes to "header.tag"
  - Place the tag file into the "tags" directory under in the WEB-INF folder.
    - /WEB-INF/tags/header.tag
  - Include the tag file using the taglib directive with its "tagdir" attribute.
 

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<myTags:header />
```

    - **Note:** Use the "tagdir" attribute instead of the "uri" attribute used for tag libraries. Notice that the name of the tag is simply the file name (header.tag).
- Unlike the `<jsp:include>` and `<c:import>` tags, you DON'T pass request parameters, you pass TAG ATTRIBUTES to tag files.
  - Passing information to the included file...
 

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<myTags:header subTitle="Never try, never fail." />
```
  - Using the information passed to the included file...
 

```
<%@ attribute name="subTitle" required="true" rtexprvalue="true" %>
My Take On Life - ${subTitle}
```

    - **Note:** In order for an attribute to be used, it must be declared so that page designers can know what can and can't be passed. This is similar to what is done in the TLD for tag libraries. In fact, think of the attribute directive like the `<attribute>` sub-element of `<tag>` used in TLDs. It serves the same exact purpose.
- All tag attributes have TAG scope. Once the tag is closed, the tag attributes go out of scope.
  - `<jsp:include>` and `<jsp:param>` values are passed as request parameters which means that to the web-app it looks as though it came in as a form submission which means that other servlets and JSPs have access to that data, not just the included file it was originally intended for.
- Tag attributes can also be passed through the body of the tag file include. This is useful and practical when passing large amounts of data, like paragraph text.
  - **Tag File (Include)...**

```
<%@ attribute name="fontColor" required="true" %>
<%@ tag body-content="tagdependent" %>

 <jsp:doBody />

```

    - **Note:** This accepts two different attributes. The one named, "fontColor", and the UNNAMED one passed in through the BODY of the tag file call. The `<jsp:doBody>` tag simply includes everything passed to it from the calling JSP. Also, notice the use of a tag directive in place of what would normally have been a page directive. Tag directives are just like page directives but are used exclusively in tag files. They have many of the same attributes, plus

an important one not found in the page directive - body-content. The body-content directive specifies what can be passed in the body of the tag. The default is "scriptless", you can also use "empty" (nothing in the tag body) and "tagdependent" (treats the body as plain text). *Tag file BODIES are NEVER allowed to have scripting, but you CAN do scripting in the actual tag file!*

- **Calling JSP...**

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
```

```
<myTags:header fontColor="red">
```

Long paragraph text.

```
</myTags:header>
```

- **Note:** This JSP is passing two different attribute values. The one named, "fontColor", and the UNNAMED one contained in the body of the opening and closing tags.
- A tag file must have a TLD if it's deployed in a JAR file; however, if it is put directly into the "WEB-INF/tags" folder or one of its sub-directories, it does not need one.
- The Container will search the following places to find tag files...
  - Directly inside WEB-INF/tags or one of its sub-directories.
  - Inside the META-INF/tags directory or one of its sub-directories of a JAR file inside the WEB-INF/lib directory.
  - **Note:** If the tag file is deployed in a JAR, there MUST be a TLD for the tag file.
- Tag files have access to the implicit request and response objects since they are included directly into the JSP. Also, tag files have access to the JspContext NOT the ServletContext.
- If tag files are deployed in a JAR, they MUST have a TLD that describes their location. But it doesn't describe attribute, body-content, etc. The TLD entries for a tag file describe ONLY the location of the actual tag file.

```
<taglib...>
 <tlib-version>1.0</tlib-version>
 <uri>myTagLibrary</uri>
 <tag-file>
 <name>header</name>
 <path>/META-INF/tags/header.tag</path>
 </tag-file>
</taglib>
```

- Custom tags MUST have a TLD, but tag files can declare attributes and body-content directly inside the tag file, and need TLDs ONLY if the tag file is in a JAR.

## Tag Handlers

- Tag **files** implement the tag functionality with another **page** (using JSP).
  - Use when doing an include that doesn't require custom Java code.
- Tag **handlers** implement the tag functionality with a special Java **class**.
  - Use when you need to write Java code.
- Tag handlers come in two flavors: **simple** and **classic**.
- Simple tag handlers and tag files were added in JSP 2.0.
- The tag handler API has **five interfaces** and **three support classes**. There's virtually no reason to implement the interfaces directly, so you'll probably **always extend a support class**.

## Simple Tag Handlers

- Below are the steps for making a **simple tag handler**...
  - Write a **class** that extends SimpleTagSupport.
 

```
public class SimpleTagTest extends SimpleTagSupport {
 ...
}
```
  - Implement the **doTag()** method of that class.
 

```
public void doTag() throws JspException, IOException {
 getJspContext().getOut().print("This is only a test.");
}
```

- **Note:** If the handler needed to process the body of the tag (by printing it directly to the response), the following line of code would be required in place of the one shown above (the null argument means the output goes to the *response* rather than some OTHER *writer* you pass in)...
  - `getJspBody().invoke(null);`
- Create a **TLD** for the tag.
 

```
<taglib...>
 <tlib-version>1.0</tlib-version>
 <uri>mySimpleTags</uri>
 <tag>
 <description>Custom simple tag example.</description>
 <name>simpleTagTest</name>
 <tag-class>com.example.SimpleTagTest</tag-class>
 <body-content>empty</body-content>
 </tag>
</taglib>
```
- **Deploy** the tag handler and TLD.
  - Put the TLD in the WEB-INF folder.
  - Put the tag handler inside WEB-INF/classes (with proper package structure).
- Write a **JSP** that uses the custom tag.
 

```
<%@ taglib prefix="myTags" uri="mySimpleTags" %>
Simple Tag Handler: <myTags:simpleTagTest />
```
- SimpleTagSupport implements the methods of SimpleTag (SimpleTag extends the JspTag interface). All of the methods of SimpleTag are implemented with the exception of doTag().
- When a JSP invokes a tag, a new instance of the tag handler class is instantiated, two or more methods are called on the handler, and when the doTag() method completes, the handler object goes away. In other words, HANDLER OBJECTS ARE NOT REUSED!
- Simple tag handler lifecycle...
  - The Container...
    - **Loads** the tag handler class.
    - **Instantiates** the class by calling its *no-arg constructor*.
    - Calls the **setJspContext(JspTag)** method.
      - This gives the handler a reference to a PageContext (subclass of JspContext).
    - If the tag is nested (invoked from within another tag), calls the **setParent(JspTag)** method.
      - A nested tag can communicate with the other tags in which it's nested.
    - If the tag has attributes, calls the **attribute setters**.
      - Uses the JavaBean naming convention.
    - If the tag is NOT declared to have a <body-content> of empty AND the tag has a body, calls the **setJspBody(JspFragment)** method.
      - If the tag has a body, the body comes in through this method, as an instance of JspFragment.
    - Calls the **doTag()** method.
      - This method is always overridden. It is where the actual code goes that powers the tag.
- Tags can use attributes that do not exist until the tag handler executes. In the code below, the tag handler actually sets the attribute and THEN invokes the body; thus the JSP can use the "message" attribute when it would have otherwise returned "null" outside of the tag's context.
  - JSP
 

```
<myTags:test>
 Message: #{message}
</myTags:test>
```
  - Tag Handler
 

```
public void doTag() throws JspException, IOException {
 getJspContext().setAttribute("message", "Freaking Idiots!!!");
 getJspBody().invoke(null);
}
```

- To use attributes with custom tags...
  - Declare each attribute in the TLD.
  - Provide getter and setter methods in the tag handler for each attribute.
    - Use JavaBean-style syntax...
      - private String attribute;
      - public String **getAttribute**();
      - public void **setAttribute**(String attribute);
- SimpleTag handlers are NEVER reused! Each tag handler instance takes care of a single invocation. A SimpleTag handler will always be initialized before any of its methods are called.
- Attribute methods of a SimpleTag handler do NOT have to only evaluate to Strings or primitives. Any object type can be used; however, if anything OTHER than a String or primitive is used, you MUST declare in the TLD that the SimpleTag handler accepts runtime values. In other words, <rtexprvalue> MUST be set to "true".
- The setJspBody() method is ONLY invoked if the following is true...
  - The tag is NOT declared in the TLD to have an empty body.
  - The tag is invoked with a body.
  - **Note:** This means that even if the tag is declared to have a non-empty body, the setJspBody() method will not be called if the tag is invoked in either of these two ways...
    - With an empty tag...
      - <foo:bar />
    - With no body...
      - <foo:bar></foo:bar>

## JspFragment

- A JspFragment is an object that represents JSP code. Its sole purpose in life is to be invoked. It's something that is meant to *run* and generate *output*.
- The *body of a tag* that invokes a simple tag handler is ENCAPSULATED in the JspFragment object, then *sent to the tag handler* in the setJspBody() method.
- JspFragments must NOT contain any scripting elements (scriptlets, declarations, or expressions).
  - JspFragments CAN contain template text, standard and custom actions, and EL expressions.
- Since JspFragments are objects, they can be passed around to other helper objects. Those helper objects, in turn, can get information from the JspFragment by invoking the getJspContext() method. Once you have the context you can ask for attributes.
- Most of the time, JspFragments will simply be used to output the body of the tag to the response.
- While you can *write* the body of the tag to something, you can't directly *get* the body.
  - If you *do* want access to the body, you can use the argument to the invoke() method to pass in a java.io.Writer, then use methods on that Writer to process the contents of the tag body.
  - The invoke() method takes a java.io.Writer. *If you want the body to be written to the response output, pass "null" to the invoke method.*

## SkipPageException

- SkipPageException sends everything BEFORE the exception to the response (the first part of the page) but NOT anything AFTER the exception was thrown (the last part of the page that would otherwise fail due to the problem encountered).
- In the example below, the JSP page will print everything up until the custom tag call at which time a SkipPageException will be thrown. As a result, the "After: SkipPageException" line will NOT print to the response. Execution will stop at that point.
  - Tag Handler
 

```
public void doTag() throws JspException, IOException {
 getJspContext().getOut().println("Message from within doTag()");
 getJspContext().getOut().println("About to throw SkipPageException");
 if (thingsGoToHell) {
 throw new SkipPageException();
 }
}
```

- JSP
  - <%@ taglib prefix="myTags" uri="mySimpleTags" %>
  - Before:** SkipPageException
  - <myTags:simpleTagTest />**
  - After:** SkipPageException
- SkipPageException stops ONLY the page that directly invoked the tag.
  - If the page that invokes the tag was included from some other page, ONLY the page that invokes the tag stops processing! The original page that did the include keeps going after the SkipPageException.

## Simple Tag Handler Review

- Tag files implement tag functionality using a *page*, while tag handlers implement tag functionality using a Java tag-handler *class*.
- To make a simple tag handler, extend SimpleTagSupport (which implements the SimpleTag interface).
- To deploy a Simple tag handler, you must create a TLD that describes the tag using the same <tag> element used by JSTL and other custom tag libraries.
- To use a Simple tag with a body, make sure the TLD <tag> for this tag does NOT declare <body-content> empty. Then call getJspBody().invoke() to cause the body to be processed.
- The SimpleTagSupport class includes implementation methods for everything in the SimpleTag interface, plus three convenience methods including getJspBody(), which you can use to get access to the contents of the body of the tag.
- The Simple tag lifecycle...
  - **Simple tags are never reused by the Container**, so each time a tag is invoked, the tag handler is instantiated, and its setJspContent() method is invoked.
  - If the tag is called from within another tag, the setParent() method is called.
  - If the tag is invoked with attributes, a bean-style setter method is invoked for each attribute.
  - If the tag is invoked with a body (assuming its TLD does NOT declare it to have an empty body), the setJspBody() method is invoked.
  - Finally, the doTag() method is invoked, and when it completes, the tag handler instance is destroyed.
- The setJspBody() method will be invoked ONLY if the tag is actually called with a body. If the tag is invoked without a body, either with an empty tag <myTags:simpleTagTest /> or with nothing between the opening and closing tags <myTags:simpleTagTest></ myTags:simpleTagTest>, the setJspBody() method will NOT be called. Remember, if the tag has a body, the TLD must reflect that, and the <body-content> must NOT have a value of "empty".
- The Simple tag's doTag() method can set an attribute used by the body of the tag, by calling getJspContext().setAttribute() followed by getJspBody().invoke().
- **The doTag() method declares a JspException and an IOException**, so you can write to the JspWriter without wrapping it in a try/catch.
- You can iterate over the body of a Simple tag by invoking the body (getJspBody().invoke()) in a loop.
- The getJspBody() method returns a JspFragment, which has two methods: invoke(java.io.Writer), and getJspContext() that returns a JspContext the tag handler can use to get access to the PageContext API (to get access to implicit variables and scoped attributes).
- Passing "null" to the invoke() method writes the evaluated body to the response output, but you can pass another Writer in if you want direct access to the body contents.
- Throw a SkipPageException if you want the current page to stop processing. If the page that invoked the tag was included from another page, the including page keeps going even though the included page stops processing from the moment the exception is thrown.

## Classic Tag Handlers

- The JSP and TLD are exactly the same with Classic tag handlers as they are with Simple tag handlers. The difference is in the actual tag handler itself.
 

```
public class ClassicTagHandler extends TagSupport {
```

```

JspWriter out;
public int doStartTag() throws JspException {
 out = pageContext.getOut();
 try {
 out.println("Inside doStartTag() method.");
 } catch (IOException ioe) {
 throw new JspException(ioe);
 }
 return SKIP_BODY;
}
public int doEndTag() throws JspException {
 try {
 out.println("Inside doEndTag() method.");
 } catch (IOException ioe) {
 throw new JspException("ioe");
 }
 return EVAL_PAGE;
}
}
}

```

- **TagSupport** - By extending TagSupport, we're implementing both Tag and Iteration Tag.
- **doStartTag()** - This overridden method is called BEFORE the body is evaluated.
- **doEndTag()** - This overridden method is called AFTER the body is evaluated.
- **pageContext.getOut()** - Classic tags inherit a pageContext member *variable* from TagSupport (in contrast to the getJspContext() *method* of SimpleTag).
- **IOException** - Notice that both methods declare/throw JspException, but NOT IOException, thus we have to wrap the IOException in a JspException object. *Note that SimpleTag's doTag() method declares both IOException AND JspException.*
- **SKIP\_BODY** - We have to return an int to tell the Container what to do next. In this case, we are telling the Container NOT to evaluate the body if there is one; instead, go straight to the doEndTag() method.
- **EVAL\_BODY** - This tells the Container to evaluate the rest of the page (as opposed to SKIP\_PAGE, which would be just like throwing a SkipPageException from a SimpleTag handler).
- The way Classic tag handlers evaluate tag bodies is very different from that of Simple tag handlers. Remember, SimpleTag bodies are evaluated when (and if) you want by calling invoke() on the JspFragment that encapsulates the body. But in Classic tags, the body is evaluated in between the doStartTag() and doEndTag() methods! Both of the examples below have the exact same behavior. *Notice that with the Classic tag handler, the body is ONLY evaluated IF the doStartTag() method returns the EVAL\_BODY\_INCLUDE constant.*

- **Simple Tag Handler**

```

public class SimpleTagHandler extends SimpleTagSupport {
 public void doTag() throws JspException, IOException {
 getJspContext().getOut().println("BEFORE body.");
 getJspBody().invoke(null);
 getJspContext().getOut().println("AFTER body.");
 }
}

```

- **Classic Tag Handler**

```

public class ClassicTagHandler extends TagSupport {
 JspWriter out;
 public int doStartTag() throws JspException {
 out = pageContext.getOut();
 try {
 out.println("BEFORE body .");
 } catch (IOException ioe) {
 throw new JspException(ioe);
 }
 }
}

```

```

 }
 return EVAL_BODY_INCLUDE;
}
public int doEndTag() throws JspException {
 try {
 out.println("AFTER body.");
 } catch (IOException ioe) {
 throw new JspException(ioe);
 }
 return EVAL_PAGE;
}
}
}

```

- The Classic tag handler lifecycle...
  - The Container...
    - **Loads** the tag handler class.
    - **Instantiates** the class by calling its *no-arg constructor*.
      - This happens the first time the tag is invoked, but the Container MAY (depending on the circumstances) re-use the Classic tag object after this.
    - Calls the **setPageContext(PageContext)** method.
      - This gives the handler a reference to a PageContext.
    - If the tag is nested (invoked from within another tag), calls the **setParent(Tag)** method.
      - A nested tag can communicate with the other tags in which it's nested.
    - If the tag has attributes, calls the **attribute setters**.
      - Uses the JavaBean naming convention.
    - Calls the doStartTag() method.
    - If the tag is NOT declared to have an empty body, AND the tag is NOT invoked with an empty body, AND the doStartTag() method returns EVAL\_BODY\_INCLUDE, **the body is evaluated**.
      - The body is evaluated BETWEEN the doStartTag() and the doEndTag() methods.
    - **If the body content was evaluated, call the doAfterBody() method.**
      - The doAfterBody() method lets you do things AFTER the body runs, and unlike the other methods *it can be invoked more than once*.
    - Calls the **doEndTag()** method.
      - This is always called ONCE, either after doStartTag() or after doAfterBody().
- Possible return values when you extend TagSupport...
  - doStartTag()
    - SKIP\_BODY
    - EVAL\_BODY\_INCLUDE
  - doAfterBody()
    - SKIP\_BODY
    - EVAL\_BODY\_AGAIN
  - doEndTag()
    - SKIP\_PAGE
    - EVAL\_PAGE
- Take special note that the constants used as return values for doStartTag() and doEndTag() return value constants that are inconsistently named.
- Returning SKIP\_PAGE from doEndTag() is exactly like throwing a SkipPageException from a Simple tag.
- When you write a tag handler that extends TagSupport, you get all the lifecycle methods from the Tag interface, plus the one method from IterationTag - doAfterBody().
  - Without doAfterBody(), you can't iterate over the body because doStartTag() is too early and doEndTag() is too late.
  - With doAfterBody(), your return value tells the Container whether it should repeat the body again (EVAL\_BODY\_AGAIN) or call the doEndTag() method (SKIP\_BODY).

- The TagSupport class assumes your tag doesn't have a body, or that if the body IS evaluated, that the body should be evaluated only ONCE. It also assumes that you always want the rest of the page to be evaluated.
- Default return values when you don't override the TagSupport method implementation...
  - doStartTag()
    - SKIP\_BODY
    - EVAL\_BODY\_INCLUDE
  - doAfterBody()
    - SKIP\_BODY
    - EVAL\_BODY\_AGAIN
  - doEndTag()
    - SKIP\_PAGE
    - EVAL\_PAGE
- *The doStartTag() and doEndTag() methods ALWAYS run exactly ONCE.*
  - These two methods are ALWAYS called ONCE, regardless of anything else that happens. But the doAfterBody() method can run from zero to many times, depending on the return value of doStartTag() and previous doAfterBody() calls.
- You MUST override doStartTag() if you want the tag body to be evaluated.
  - *The default value from doStartTag() is SKIP\_BODY, so if you want the body of your tag evaluated, and you extend TagSupport, you MUST override doStartTag() if for no other reason than to return EVAL\_BODY\_INCLUDE.*
  - With doAfterBody(), it should be obvious that if you want to iterate over the body, you have to override that method as well, since its return value is SKIP\_BODY.
- If you're implementing TagSupport, and you want to set values the body can use, then you MUST set those attribute values in doStartTag(). You can't wait until doAfterBody(), because by the time you get to doAfterBody(), the body has already been processed once.
- **Unlike Simple tag handlers, Classic tag handlers can be pooled and reused by the Container.**
  - Be very careful about instance variables, you should reset them with each new tag invocation (which means in the doStartTag() method). Otherwise, the code will only work correctly with the first invocation; subsequent invocations will be incorrect.
  - Don't rely on the release() method to reset your instance variables as this is only called when the Container wants to dispose of the tag handler.
- Extending BodyTagSupport gives you two more lifecycle methods from the BodyTag interface - setBodyContent() and doInitBody(). You can use these to do something with the actual CONTENTS of the body of the tag used to invoke the handler.
- When extending BodyTagSupport, the doStartTag() also gets a new return value possibility - EVAL\_BODY\_BUFFERED. In fact, the new value becomes the default (instead of SKIP\_BODY).
- If you do NOT extend BodyTagSupport or implement BodyTag, then you must NOT return EVAL\_BODY\_BUFFERED from the doStartTag() method.
- If the TLD for a tag declares an empty body, doStartTag() MUST return SKIP\_BODY.
- Lifecycle return values for Classic tag methods...

		<b>BodyTagSupport</b>	<b>TagSupport</b>
<b>doStartTag()</b>	<i>Possible</i> return values	SKIP_BODY EVAL_BODY_INCLUDE EVAL_BODY_BUFFERED	SKIP_BODY EVAL_BODY_INCLUDE
	<i>Default</i> return value from the implementation class	EVAL_BODY_BUFFERED	SKIP_BODY
	Number of times it can be called per JSP tag invocation	Exactly once	Exactly once
<b>doAfterBody()</b>	<i>Possible</i> return values	SKIP_BODY EVAL_BODY_AGAIN	SKIP_BODY EVAL_BODY_AGAIN

	<i>Default</i> return value from the implementation class	SKIP_BODY	SKIP_BODY
	Number of times it can be called per JSP tag invocation	Zero to many	Zero to many
<b>doEndTag()</b>	<i>Possible</i> return values	SKIP_PAGE EVAL_PAGE	SKIP_PAGE EVAL_PAGE
	<i>Default</i> return value from the implementation class	EVAL_PAGE	EVAL_PAGE
	Number of times it can be called per JSP tag invocation	Exactly once	Exactly once
<b>doInitBody()</b> <b>setBodyContent()</b>	Circumstances under which they can be called, and number of times per tag invocation.	Exactly once, and <b>ONLY</b> if <b>doStartTag()</b> returns <b>EVAL_BODY_BUFFERED</b>	<b>Never</b>

### Nested/Cooperating Tags

- Both SimpleTag and Tag have a getParent() method.
  - Tag's getParent() returns a Tag (don't forget to cast).
    - OuterTag parent = (OuterTag) getParent();
  - SimpleTag's getParent() returns an instance of JspTag (once again requires cast).
    - OuterTag parent = (OuterTag) getParent();
    - MyClassicParent parent = (MyClassicParent) getParent();
- The getParent() method will return either another tag (on which you can call its getParent()) or null.
- Simple tags can access BOTH Classic and Simple parents, but Classic tags can ONLY access Classic tag parents.
- You can walk up, but you can't walk down...
  - There's a getParent() method, but no getChild() method.
- Getting information from the child to the parent...
  - Since there's no automatic mechanism for the parent to find out about its child tags, you simply have to use the same design approach to get info to the parent FROM the child as you do to get info from the parent TO the child. You get a reference to the parent tag, and call methods. Only instead of getters, this time you call some kind of *set* or *add* method.
    - Parent Tag Handler...

```

public class ParentTag extends TagSupport {
 private List children;
 public void addChild(String child) {
 children.add(child);
 }
 public int doStartTag() throws JspException {
 children = new ArrayList();
 return EVAL_BODY_INCLUDE;
 }
 public int doEndTag() throws JspException {
 ...
 }
}

```

- **Note:** The addChild() method is NOT an attribute setter method. It's just a helper method. It's called in between the doStartTag() and doEndTag() methods. Notice that the List is initialized and reset in the doStartTag() method. This is because the Container may choose to reuse the tag handler. Also, if you do not return EVAL\_BODY\_INCLUDE, the child tags will never be processed (the nested tags will be skipped).
- Child Tag Handler...
 

```
public class ChildTag extends TagSupport {
 private String value;
 public void setValue(String value) {
 this.value = value;
 }
 public int doStartTag() throws JspException {
 return EVAL_BODY_INCLUDE;
 }
 public int doEndTag() throws JspException {
 ParentTag parent = (ParentTag) getParent();
 parent.addChild(value);
 return EVAL_PAGE;
 }
}
```

  - **Note:** The "value" member has a corresponding attribute in the TLD. To add the child to the parent, simply call its addChild() method to add it to the parent's list of children.
- There is another mechanism for skipping nesting levels and going straight to a grandparent or great grandparent - findAncestorWithClass() (this is a STATIC method). The Container will walk the tag nesting hierarchy until it finds a tag that's an instance of the class specified in the method call. It returns the FIRST one it finds. If you need the second one, then use it to find the first occurrence, then use the method again to find the next (and so forth).
  - SomeOuterTag ancestor = (SomeOuterTag) findAncestorWithClass(this, SomeOuterTag.class);
    - **this** - Represents the starting tag.
    - **SomeOuterTag.class** - This is the class (ancestor) you're looking for.

	Simple Tags	Classic Tags
Tag interfaces	SimpleTag (extends JspTag)	Tag (extends JspTag) IterationTag (extends Tag) BodyTag (extends IterationTag)
Support implementation classes	SimpleTagSupport (implements SimpleTag)	TagSupport (implements IterationTag) BodyTagSupport (extends TagSupport, implements BodyTag)
Key lifecycle methods that YOU might implement	doTag()	doStartTag() doEndTag() doAfterBody()  And for BodyTag... - doInitBody() and setBodyContent()
How you write to the response output	getJspContext().getOut().println()  (No try/catch needed because SimpleTag methods declare IOException)	pageContext.getOut().println()  (Wrapped in a try/catch because Classic tag methods do NOT declare the IOException)
How you access implicit variables and scoped attributes from a support implementation	With the getJspContext() method that returns a JspContext (which is usually a PageContext)	With the pageContext implicit variable, NOT a method like it is with SimpleTag
How you cause the body to be processed	getJspBody().invoke(null)	Return EVAL_BODY_INCLUDE from doStartTag(), or

		EVAL_BODY_BUFFERED if the class implements BodyTag
How you cause the current page evaluation to STOP	Throw a SkipPageException	Return SKIP_PAGE from doEndTag()

- Remember, a tag handler class is NOT a servlet or a JSP, so it DOES NOT have automatic access to a bunch of implicit objects. It does; however, get a reference to a PageContext, and with it can get to all kinds of things it might need.
  - While Simple tags get a reference to a JspContext and Classic tags get a reference to a PageContext, the Simple tag's JspContext is usually a PageContext instance. So if your Simple tag handler needs access to PageContext-specific methods or fields, you'll have to cast it from a JspContext to a PageContext.
- The one-arg getAttribute(String) method in JspContext is for page scope ONLY. It does NOT search all the scopes for the correct match. DO NOT confuse this with the findAttribute() method that searches ALL the scopes (page, request, session, application) looking for the FIRST match.

## CHAPTER 11 (Web App Deployment)

### Web App Structure

- webapps**
  - myWebApp**
    - myPage.jsp** - Static content and JSPs can be at the web app root level OR in a subdirectory, including under WEB-INF, although that affects their accessibility.
    - WEB-INF**
      - web.xml** - The deployment descriptor MUST be named, "web.xml" and it MUST be immediately inside the "WEB-INF" directory (NOT inside a subdirectory).
      - classes**
        - (package structure)** - The package structure for ALL class files (servlets, listeners, helpers, beans, tag handlers, etc.) must be immediately under "WEB-INF/classes".
          - MyTagHandler.class
      - lib**
        - (JAR file)**
          - META-INF** - "META-INF" must be immediately inside the JAR file. TLDs in a JAR file MUST be somewhere inside "META-INF". TLD files NOT in a JAR must be somewhere under "WEB-INF".
            - tlds**
              - myTags.tld
            - (package structure)** - The package structure for classes in a JAR must be IMMEDIATELY inside the JAR, and the JAR must be inside "WEB-INF/lib".
              - MyHandler.class
        - Tags** - Tag files (.tag or .tagx) MUST be inside "WEB-INF/tags" or in a subdirectory.
          - include.tag**

### WAR Files

- WAR (Web ARchive) files are JAR (Java ARchive) files with a ".war" extension.
  - They're basically zipped/compressed files.
  - WARs contains everything UNDER the web app context directory (not including myWebApp).
  - When WARs are deployed, depending on the Container, the context or web app name is the name of the WAR file (this is the case with Tomcat at least).
- The ONLY significant difference between a WAR and an exploded directory structure is that the WAR requires a "META-INF" directory (a WAR is a JAR) at the same level as "WEB-INF".
  - This means that WARs CAN declare library dependencies (unlike exploded web apps).

- The META-INF/MANIFEST.MF file gives you a DEPLOY-TIME check for whether the Container can find the packages and classes your web app depends on. You don't have to wait until request-time to blow up; you can know immediately.

## File Accessibility

- Direct access to files can be prevented by placing them under the WEB-INF directory or, if deployed as a WAR file, under the META-INF directory. Either location will prevent them from being accessed directly. Any other location is fair game.
- If the server gets a client request for anything under WEB-INF or META-INF, the Container MUST respond with a 404 NOT FOUND error.
- The Container automatically puts JAR files onto its classpath, so classes for servlets, listeners, beans, etc. are available EXACTLY as they would have been had they been originally placed directly in the WEB-INF/classes directory (keeping their correct package structure, of course). Keep in mind, that the Container will always look for classes in the WEB-INF/classes directory BEFORE it looks inside JAR files in WEB-INF/lib. **Classes come before JARs!**
- If your web app code needs direct access to a resource (text file, JPEG, etc.) that's inside a JAR, you need to use the getResource() or getResourceAsStream() methods of the classloader. This is just plain old J2SE, not specific to servlets.
  - This is different from the getResource() and getResourceAsStream() methods from the ServletContext API. The difference is that the methods inside ServletContext work ONLY for resources WITHIN the web app that are NOT deployed within a JAR file.

## Servlet Mappings

- Every servlet mapping has two parts...
  - <servlet> - Defines a servlet name and class.
  - <servlet-mapping> - Defines the URL pattern that maps to a servlet name defined somewhere else in the deployment descriptor.
- The URL patterns you put into a servlet mapping can be completely made-up. It's just a logical name you want to give clients. Remember, URL patterns don't map to anything other than the <servlet-name> defined in the <servlet> element.
- The <servlet-name> elements are the key to servlet mappings - they match a request <url-pattern> to an actual servlet class.
  - Clients request servlets by <url-pattern>, NOT by <servlet-name> or <servlet-class>.
- Three types of <url-pattern> elements...
  - Exact Match...
    - <url-pattern>/someDirectory/someFile.do</url-pattern>
      - MUST begin with a slash (/).
      - CAN have an OPTIONAL extension.
  - Directory Match...
    - <url-pattern>/someDirectory/\*</url-pattern>
      - MUST begin with a slash (/).
      - CAN be a virtual OR real directory.
      - MUST end with a slash/asterisk (/).
  - Extension Match...
    - <url-pattern>\*.do</url-pattern>
      - MUST begin with an asterisk (\*) - NEVER with a slash (/).
      - MUST have a dot extension (.do, .jsp, etc.).
- Key rules about servlet mappings...
  - The Container always looks for matches in the following order (most restrictive to broadest)...
    - Exact
    - Directory
    - Extension
  - If the request matches more than one directory <url-pattern>, the Container will choose the longest, most SPECIFIC match.

## Welcome Files

- Welcome pages are configured in the deployment descriptor (web.xml). The Container iterates through the list choosing the first match it finds.

```
<web-app ...>
 <welcome-file-list>
 <welcome-file>index.html</welcome-file>
 <welcome-file>default.html</welcome-file>
 <welcome-file>home.html</welcome-file>
 </welcome-file-list>
</web-app>
```
- Multiple welcome files go in a single deployment descriptor element, `<welcome-file-list>`.
- *The files in the `<welcome-file>` element do NOT start with a slash.*
- When a client requests a directory with no specified resource, the Container iterates the `<welcome-file-list>` to find a matching file IN THAT PARTICULAR directory. If it does not find one, the behavior is vendor-specific.
  - Tomcat, for example, shows a directory listing.
  - Another Container might show a "404 Not Found" error.

## Error Pages

- There are three different types of error pages...
  - Catchall exception error page...

```
<error-page>
 <exception-type>java.lang.Throwable</exception-type>
 <location>/errorPage.jsp</location>
</error-page>
```

    - **Note:** You can override this in individual JSPs by adding a page directive with an "errorPage" attribute.
  - Explicit exceptions...

```
<error-page>
 <exception-type>java.lang.ArithmeticException</exception-type>
 <location>/arithmeticErrorPage.jsp</location>
</error-page>
```

    - **Note:** The Container will choose the most specific exception before defaulting to the catchall error page (if provided).
  - HTTP status code error page...

```
<error-page>
 <error-code>404</error-code>
 <location>/pageNotFoundErrorPage.jsp</location>
</error-page>
```

    - **Note:** This is only called when a 404 HTTP status code is encountered. Also be aware that you CANNOT use `<error-code>` and `<exception-type>` together (in the same `<error-page>` element). It's one or the other (by exception type or error code).
- ANYTHING that extends Throwable can be declared in the `<exception-type>` element - that includes `java.lang.Error`, runtime exceptions, and any checked exceptions (so long as they are on the Container's classpath, of course).
- You MUST always use the fully qualified class name in the `<exception-type>`.
- You CAN generate error codes using `response.sendError()`. There is no difference between Container-generated and programmer-generated HTTP errors.

## Servlet Initialization

- If you want servlets to be loaded at deploy time (or at server restart time) rather than on first request, use the `<load-on-startup>` element in the deployment descriptor. Any value greater than zero will work.

- If you have multiple servlets that you want preloaded, and you want to control the order in which they're initialized, use the value of `<load-on-startup>` to determine the order.
  - If two servlets have the same `<load-on-startup>` value, then the order in which the servlets are declared in the deployment descriptor will determine which gets preloaded first.
- Any number greater than zero means "initialize this servlet at deployment or server startup time, rather than waiting for the first request."
 

```
<web-app ...>
 <servlet>
 <servlet-name>ServletOne</servlet-name>
 <servlet-class>com.example.ServletOne</servlet-class>
 <load-on-startup>1</load-on-startup>
 </servlet>
</web-app>
```
- Values greater than one do not affect the *number of servlet instances*.
  - For example, `<load-on-startup>4</load-on-startup>` does NOT mean "load four instances of the servlet". It means that this servlet should be loaded only AFTER servlets with a `<load-on-startup>` number less than four are loaded.

## JSP Documents

- JSP documents are just normal JSP pages written with XML syntax.
- In addition to the diagram below, the main differences between JSP documents and JSP pages, besides the XML syntax, is that JSP documents are enclosed within a `<jsp:root>` tag, and that JSP documents do not specify taglib directives using the `<jsp:directive>` tag, but instead have them declared from directly within the `<jsp:root>` tag as attributes.

	JSP Page Syntax	JSP Document Syntax
<b>Directives</b> (Except Taglib)	<code>&lt;%@ page import="java.util.*" %&gt;</code>	<code>&lt;jsp:directive.page import="java.util.*" /&gt;</code>
<b>Declarations</b>	<code>&lt;%!   int x = 3; %&gt;</code>	<code>&lt;jsp:declaration&gt;   int x = 3; &lt;/jsp:declaration&gt;</code>
<b>Scriptlets</b>	<code>&lt;%   list.add("value"); %&gt;</code>	<code>&lt;jsp:scriptlet&gt;   list.add("value"); &lt;/jsp:scriptlet&gt;</code>
<b>Text</b>	Sample Text	<code>&lt;jsp:text&gt;   Sample Text &lt;/jsp:text&gt;</code>
<b>Scripting Expressions</b>	<code>&lt;%=   map.get("key") %&gt;</code>	<code>&lt;jsp:expression&gt;   map.get("key") &lt;/jsp:expression&gt;</code>

## EJB-Related Tags

- A LOCAL bean means that the client and the bean MUST be running in the SAME JVM.
- A REMOTE bean means that the client and the bean CAN be running in DIFFERENT JVMs (possibly on different machines as well).
- Declaring EJBs...
  - Local Bean References...
 

```
<ejb-local-ref>
 <ejb-ref-name>ejb/Customer</ejb-ref-name>
 <ejb-ref-type>Entity</ejb-ref-type>
 <local-home>com.example.ejb.CustomerHome</local-home>
 <local>com.example.ejb.Customer</local>
</ejb-local-ref>
```

- **Note:** The `<ejb-ref-name>` element contains the JNDI lookup name used in the code to find the actual EJB. The `<local-home>` and `<local>` elements **MUST** both contain fully qualified names.
- Remote Bean References...
  - `<ejb-ref>`

```

 <ejb-ref-name>ejb/LocalCustomer</ejb-ref-name>
 <ejb-ref-type>Entity</ejb-ref-type>
 <home>com.example.ejb.CustomerHome</home>
 <remote>com.example.ejb.Customer</remote>
 </ejb-ref>

```
- Notice that the LOCAL and REMOTE tags are inconsistent...
  - `<ejb-local-ref>` vs. `<ejb-ref>`
    - Notice that there is **NO** `<ejb-remote-ref>`. This is because at the time `<ejb-ref>` was first defined, there was no such thing as "local" EJBs. Since ALL EJBs were "remote", there was no need to differentiate between local and remote, so there was no need to put "remote" into the name of the tag.
  - `<local-home>` vs. `<home>`
    - Notice that there is **NO** `<remote-home>`. Once again for the same reason as the other inconsistency. There were **NO** local EJBs at the time `<home>` was defined. Therefore, there was no need to differentiate.
- Declaring JNDI environment entries...
  - `<env-entry>`

```

 <env-entry-name>rates/discountRate</env-entry-name>
 <env-entry-type>java.lang.Integer</env-entry-type>
 <env-entry-value>10</env-entry-value>
 </env-entry>

```

    - **<env-entry-name>** - This is the lookup name to use in the code.
    - **<env-entry-type>** - This tag's value can be any type that takes a single String as a constructor parameter (or a single Character if it's `java.lang.Character`). In this particular case, the Integer wrapper's constructor does accept a single String argument. The `<env-entry-type>` tag is **NOT** just limited to wrappers. It can be **ANYTHING** that accepts a single String argument in its constructor (or a single Character for a `Character` type).
    - **<env-entry-value>** - This tag's value will be passed in as a String (or a single Character if the `<env-entry-type>` is `java.lang.Character`).

## MIME Mappings

- Mappings between extensions and MIME types can be configured in the deployment descriptor.
  - `<mime-mapping>`

```

 <extension>mpg</extension>
 <mime-type>video/mpeg</mime-type>
 </mime-mapping>

```

    - **Note:** *DO NOT include the dot (.) in the `<extension>` tag.* It's **ONLY** the character extension (mpg). Also, do not confuse the `<extension>` tag with `<file-type>` or `<content-type>`. Remember, `<extension>` goes with `<mime-type>`.

## CHAPTER 12 (Web App Security)

### Servlet Security

- Servlet security helps foil...
  - Impersonators - Someone who pretends to be someone else.
  - Upgraders - Someone who gets into special areas for upgrades or discounts.
  - Eavesdroppers - Someone who listens in and can scare off customers.
- Servlet security boils down to four main concepts...
  - Authentication - Foils Impersonators.

- Authorization - Foils Upgraders.
- Confidentiality - Foils Eavesdroppers.
- Data Integrity - Foils Eavesdroppers.
- How the Container does authentication and authorization...
  - When the Container receives a request with a username and password, it checks the URL in the security table.
  - If it finds the URL in the security table (and sees that it's constrained), it checks the username and password information to make sure they match.
  - If the username and password are OK, the Container checks to see if the user has been assigned the correct "role" to access this resource (i.e. authorization). If so, the resource is returned to the client. If not, the Container returns a 401-error page.
- Some reasons why security constraints should be handled declaratively...
  - Supports the idea of component-based development.
  - Reduces ongoing maintenance when your application grows.
  - Allows application developers to reuse servlets without access to the source code.
  - Allows you to use servlets you've already written in more flexible ways.
  - Often maps naturally to the existing job roles in a company's IT department.
- 95% of the security work you'll do in servlets will be declarative. Programmatic security just isn't used very much.
- The table below illustrates the key items in servlet security. Authorization is the most time-consuming to implement and authentication is next. From the servlet perspective, confidentiality and data integrity are pretty easy to set up.

Security Concept	Responsibility	Complexity Level	Effort Level	Exam Importance
Authentication	Admin	Medium	High	Medium
Authorization	Deployer (Mostly)	High	High	High
Confidentiality	Deployer	Low	Low	Low
Data Integrity	Deployer	Low	Low	Low

## Authorization

- Authorization is the most important and complex of the vendor-neutral security concepts.
- Authorization...
  - Step 1: Defining Roles...
    - The most common form of authorization in servlets is for the Container to determine whether a specific servlet and the invoking HTTP request method can be called by a user who has been assigned a certain security "role". This is achieved by mapping roles in the vendor-specific "users" file to roles established in the deployment descriptor. *In other words, the deployer creates <role-name> elements in the deployment descriptor so that the Container can map roles to users.*
      - Vendor-Specific Descriptor (users.xml)
 

```
<tomcat-users>
 <role rolename="Admin" />
 <role rolename="Member" />
 <role rolename="Guest" />
 <user username="Peter" password="admin" roles="Admin, Member, Guest" />
 <user username="Paul" password="member" roles="Member, Guest" />
 <user username="Mary" password="guest" roles="Guest" />
</tomcat-users>
```
      - Deployment Descriptor (web.xml)
 

```
<web-app ...>
 <security-role>
 <role-name>Admin</role-name>
 <role-name>Member</role-name>
 <role-name>Guest</role-name>
```

```

</security-role>
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
</web-app>

```

- Step 2: Defining Resource/Method Constraints...
  - Declaratively specify that a given resource/method combination is accessible only by users in certain roles. Most of the security work done is with <security-constraint> elements in the deployment descriptor.
    - Deployment Descriptor (web.xml)
 

```

<web-app ...>
 <security-constraint>
 <web-resource-collection>
 <web-resource-name>Users</web-resource-name>
 <url-pattern>/Example/AddUser/*</url-pattern>
 <url-pattern>/Example/ViewUser/*</url-pattern>
 <http-method>GET</http-method>
 <http-method>POST</http-method>
 </web-resource-collection>
 <auth-constraint>
 <role-name>Admin</role-name>
 <role-name>Member</role-name>
 </auth-constraint>
 </security-constraint>
</web-app>

```

      - **web-resource-name** - Mandatory name used by tools.
      - **url-pattern** - Required element that defines the resources to be *constrained*.
      - **http-method** - Describes which HTTP methods are *constrained* (restricted) for the resources defined by the URL.
      - **auth-constraint** - This optional element lists which roles CAN invoke the constrained HTTP methods. In other words, it says WHO is allowed to do a GET and POST on the specified URL patterns.
  - The purpose of the <web-resource-collection> element is to tell the Container which resources and HTTP method combinations should be *constrained*.
  - <web-resource-collection> Key Points...
    - It has two primary sub-elements...
      - <url-pattern> (one or more)
      - <http-method> (optional, zero or more)
    - The URL patterns and HTTP methods work together to define resource requests that are *constrained* (restricted).
    - A <web-resource-name> element is MANDATORY (even though you probably won't use it for anything yourself. It is intended for tools).
    - A <description> element is OPTIONAL.
    - The <url-pattern> element uses servlet standard naming and mapping rules.
    - You MUST SPECIFY AT LEAST ONE <url-pattern>, but you CAN have many.
    - Valid methods for the <http-method> element are GET, POST, PUT, TRACE, DELETE, HEAD, and OPTIONS.
    - If NO HTTP methods are specified then ALL methods will be constrained (allowed).
    - If you DO specify an <http-method>, then ONLY THOSE METHODS specified will be constrained (allowed). In other words, once you specify even a single <http-method>, you automatically enable (disallow) any HTTP methods which you have NOT specified.
    - You can have more than one <web-resource-collection> element in the same <security-constraint>.
    - The <auth-constraint> element applies to ALL <web-resource-collection> elements in the <security-constraint>.

- Constraints are not at the RESOURCE level; they are at the HTTP REQUEST level.
  - Resources are NOT constrained. It's the combination of a RESOURCE AND HTTP METHOD that are constrained.
  - A resource is always constrained on an HTTP method by an HTTP method basis, although you CAN configure the <web-resource-collection> in such a way that ALL methods are constrained (allowed), simply by NOT putting in any <http-method> elements.
  - The <auth-constraint> element does NOT define which roles are allowed to access the resources from the <web-resource-collection>. Instead, it defines which roles are allowed to make the *constrained request*.
    - Don't think of it as "Bob is a Member, so Bob can access the AddUser servlet". Instead, say "Bob is a Member, so Bob can make a GET or POST request on the AddUser servlet".
- If you specify an <http-method> element, all the other HTTP methods you do NOT specify are unconstrained (not allowed).
  - The web server's job is to SERVE, so the default assumption is that you want the HTTP methods to be UNconstrained unless you explicitly say (using <http-method>) that you want a method to be constrained (for the resources that match the <url-pattern>).
    - If you put in ONLY an <http-method>GET</http-method> in the security constraint, then POST, TRACES, PUT, etc. are NOT constrained (allowed). That means anybody, regardless of security role (or even regardless of whether the client is authenticated), can invoke those HTTP methods. This is ONLY if you have specified at least one <http-method> element. If you do NOT specify any <http-method>, then you're constraining (allowing) ALL HTTP methods (You'll probably never want to do that since the whole point of a security constraint is to constrain specific HTTP requests on a particular set of resources).
    - Of course, HTTP methods won't work in a servlet unless you've overridden the appropriate doXXX() methods.
  - Any HTTP methods supported by your servlet (because you overrode the matching service method) will be allowed UNLESS you do one of two things...
    - Do not specify ANY <http-method> elements in the <security-constraint>, which means that ALL methods are constrained.
    - Explicitly list the method using the <http-method> element.
- Even though it's got *constraint* in its name, the <auth-constraint> sub-element of <security-constraint> specifies which roles are ALLOWED to access the web resources specified by the <web-resource-collection>.
  - <role-name> Rules...
    - Within an <auth-constraint> element, the <role-name> element is OPTIONAL.
    - If <role-name> elements exist, they tell the Container which roles are ALLOWED.
    - If an <auth-constraint> element exists with NO <role-name> elements, then NO USERS ARE ALLOWED.
    - If <role-name>\*</role-name>, then ALL users are ALLOWED.
    - If NO <role-name> exists, then ALL users are ALLOWED.
    - If <auth-constraint />, then ALL users are ALLOWED.
      - No <auth-constraint> is the opposite of an EMPTY <auth-constraint />.
        - If you don't specify WHICH roles are constrained, then NO roles are constrained. But once you DO put in an <auth-constraint>, then ONLY the roles explicitly stated are allowed access (unless you specify the wildcard asterisk).
        - If you don't want ANY role to have access, you MUST put in the <auth-constraint />, but just leave it empty. This tells the Container, "I'm explicitly stating the roles allowed and, by the way, there aren't any!"
    - Role names are CASE-SENSITIVE.
  - <auth-constraint> Rules...
    - Within a <security-constraint> element, the <auth-constraint> element is OPTIONAL.
    - If an <auth-constraint> exists, the Container MUST perform authentication for the associated URLs.
    - If an <auth-constraint> does NOT exist, the Container MUST allow unauthenticated access for these URLs.

- For readability, you can add a <description> inside <auth-constraint>.
- When two different non-empty <auth-constraint> elements apply to the same constrained resource, access is granted to the *union of all roles* from both of the <auth-constraint> elements.
  - Rules...
    - When combining individual role names, ALL of the role names listed will be allowed.
    - A role name of "\*" combines with anything else to allow access to EVERYBODY.
    - An EMPTY <auth-constraint /> TAG combines with ANYTHING else to allow access to NOBODY. *In other words, an empty <auth-constraint /> is always the final word.*
    - If one of the <security-constraint> elements has NO <auth-constraint> ELEMENT, it combines with anything else to allow access to EVERYBODY.

Security Constraint - A	Security Constraint - B	Access
<auth-constraint> <role-name>Guest</role-name> </auth-constraint>	<auth-constraint> <role-name>Admin</role-name> </auth-constraint>	Guest Admin
<auth-constraint> <role-name>Guest</role-name> </auth-constraint>	<auth-constraint> <role-name>*</role-name> </auth-constraint>	Everybody
<b>EMPTY &lt;auth-constraint /&gt; TAG</b>	<auth-constraint> <role-name>Admin</role-name> </auth-constraint>	<b>Nobody</b>
<b>NO &lt;auth-constraint&gt; TAG</b>	<auth-constraint> <role-name>Admin</role-name> </auth-constraint>	<b>Everybody</b>
<b>Note:</b> This is the same as "*" .		

- While it may seem strange to ever want to use an empty <auth-constraint /> tag to deny access to everybody, there are times when it's useful. The access denied is only to clients OUTSIDE the web app; resources INSIDE the web app are free to access at will. Think of constrained resources as private methods; they're for internal use only.
- In HttpServletRequest, there are three methods associated with programmatic security...
  - **getUserPrincipal()** - Mainly used with EJBs (not covered on exam).
  - **getRemoteUser()** - Used to check authentication status (not commonly used; nothing more to know about it for the exam).
  - **isUserInRole()** - Instead of just authorizing at the HTTP *method* level, allows you to authorize *portions* of a method. This gives you a way to customize how a service method behaves based on the user's role. If you're in this service method, then the user made it through the declarative authorization, but now you want to do something in the method conditionally, based on whether the user is in a particular role.
    - How it works...
      - Before the isUserInRole() method is called, the user needs to be authenticated. If the method is called on a user that has NOT been authenticated, the Container will ALWAYS return false.
      - The Container takes the isUserInRole() method's argument and compares it to the roles defined for the user in this request.
      - If the user is mapped to this role, the Container returns true.
- The Container will ALWAYS use a <security-role-ref> mapping even IF the programmatic name matches a "real" <security-role> name.
  - When the Container hits an argument to "isUserInRole()", it looks FIRST for a matching <security-role-ref>. If it finds one, that's what it uses, even when the hard-coded name really DOES match a <security-role> name. *The <security-role-ref> ALWAYS has precedence over the <security-role> elements.*
  - The <security-role-ref> element maps programmatic (hard-coded) role names to declarative <security-role> elements. In the following example, if the <security-role-ref> didn't exist, the code would fail because there is no, <security-role> named "Manager".
    - Deployment Descriptor...

```

<web-app ...>
 <servlet>
 <security-role-ref>
 <role-name>Manager</role-name>
 <role-link>Admin</role-link>
 </security-role-ref>
 </servlet>
 <security-role>
 <role-name>Admin</role-name>
 <role-name>Member</role-name>
 <role-name>Guest</role-name>
 </security-role>
</web-app>

```

- Servlet Method...

```

public void methodName() {
 if (request.isUserInRole()) {
 ...
 } else {
 ...
 }
}

```

- Authorization operates completely independent of the transport layer. It happens within the Container once authentication has occurred. Authentication can affect the transport layer based on how the <auth-method> element is set.

## Authentication

- Users can't be authorized until they're authenticated.
- For a J2EE Container, authentication comes down to this...
  - Ask for a username and password, then verify that they match.
- The servlet specification doesn't talk about HOW a Container should implement support for authentication data, including usernames and passwords. But the general idea is that the Container will supply a vendor-specific table containing usernames and their associated passwords and roles.
  - Virtually all vendors provide hooks into company-specific authentication data such as databases, LDAP, etc.
- A **realm** is a place where **authentication** information is stored.
  - Tomcat uses "tomcat-users.xml".
    - It's located under tomcat's "conf" directory, NOT within "webapps".
    - This file applies to ALL applications deployed under "webapps".
    - It's commonly known as the *memory realm* because Tomcat reads this file into memory at startup time.
    - It's great for testing but NOT recommended for production.
    - You can't modify it without having to restart Tomcat.
    - Your app server will use something different... but SOMEHOW it will let you map users to passwords and roles.
- The control for authentication is located in some sort of data structure like the following. In Tomcat, you can use an XML file called, "tomcat-users.xml" that holds name-password-role sets that the Container uses at authentication time.
 

```

<tomcat-users>
 <role rolename="Guest" />
 <role rolename="Member" />
 <user name="John" password="John Boy" roles="Member, Guest" />
</tomcat-users>

```
- For authentication, you also need to add instructions to the deployment descriptor so that the Container will know to ask for a username and password. There are several different types of methods. The following is one of the simplest...

```

<web-app ...>
 <login-config>
 <auth-method>BASIC</auth-method>
 </login-config>
</web-app>

```

- The first time an unauthorized user asks for a constrained resource, the Container will automatically start the authentication process.
- There are four types of authentication the Container can provide...
  - **BASIC** - Transmits the login information in an encoded (NOT encrypted) form. Since the encoding scheme (base64) is very well known, BASIC provides VERY weak security.
  - **DIGEST** - Transmits the login information in a more secure way, but because the encryption mechanism isn't widely used, J2EE containers aren't required to support it.
  - **CLIENT-CERT** - Transmits login information in an extremely secure form, using Public Key Certifications (PKC). The downside is that clients are required to have a certificate BEFORE they can log into your system. It's fairly rare for consumers to have a certificate, so CLIENT-CERT authentication is used mainly for B2B scenarios.
  - **FORM** - This type of authentication lets you create your own custom HTML login form. Of all four types, this is the LEAST SECURE. The username and password are sent back in the HTTP request with NO encryption.
- BASIC, DIGEST, and CLIENT-CERT all use the browser's standard pop-up form (dialog box) for inputting usernames and passwords. FORM is the only one that allows you to customize it.
- Except for FORM, once you've declared the <login-config> element in the deployment descriptor, implementing authentication is done (assuming you've already configured the username/password/role information into your server).

- **BASIC**

```

<web-app ...>
 <login-config>
 <auth-method>BASIC</auth-method>
 </login-config>
</web-app>

```

- **DIGEST**

```

<web-app ...>
 <login-config>
 <auth-method>DIGEST</auth-method>
 </login-config>
</web-app>

```

- **CLIENT-CERT**

```

<web-app ...>
 <login-config>
 <auth-method>CLIENT-CERT</auth-method>
 </login-config>
</web-app>

```

- **FORM**

```

<web-app ...>
 <login-config>
 <auth-method>FORM</auth-method>
 <form-login-config>
 <form-login-page>/login.html</form-login-page>
 <form-error-page>/error.html</form-error-page>
 </form-login-config>
 </login-config>
</web-app>

```

- FORM-based authentication requires three things...
  - A <login-config> declaration
  - An HTML login form

- There are three entries that are key to communicating with the Container...
 

```
<form method="POST" action="j_security_check">
 <input type="text" name="j_username" />
 <input type="password" name="j_password" />
 <input type="submit" name="login" />
</form>
```
- An HTML error page

Type	Spec	Data Integrity	Comments
BASIC	HTTP	Weak (Base64)	HTTP standard All browsers support it
DIGEST	HTTP	Stronger (But not SSL)	Optional for both HTTP and J2EE Containers
FORM	J2EE	Very Weak (No encryption)	Allows custom login screen
CLIENT-CERT	J2EE	Strong (Public Key)	Strong, but users must have certificates

- Data integrity and confidentiality refer to the degree to which an eavesdropper can steal or tamper with a user's information.
  - Data *integrity* means that data that arrives is the same as the data that was sent.
  - Data *confidentiality* means that nobody else can see the data sent during transmission.
- When you tell a J2EE Container that you want to implement data confidentiality and/or integrity, the J2EE spec guarantees that the data to be transmitted will travel over a "**protected transport layer connection**".
  - Containers are NOT REQUIRED to use any SPECIFIC protocol to handle secure transmission, but in practice they nearly all use **HTTPS over SSL**.
- The following example demonstrates how to implement data confidentiality and integrity sparingly and declaratively. The three sub-elements together read: Only **Members** can make POST requests to resources found in the "**EditUser**" directory, and make sure the transmission is secure.

```
<web-app ...>
 <security-constraint>
 <web-resource-collection>
 <web-resource-name>Users</web-resource-name>
 <url-pattern>/Example/EditUser/*</url-pattern>
 <http-method>POST</http-method>
 </web-resource-collection>
 <auth-constraint>
 <role-name>Member</role-name>
 </auth-constraint>
 <user-data-constraint>
 <transport-guarantee>CONFIDENTIAL</transport-guarantee>
 </user-data-constraint>
 </security-constraint>
</web-app>
```

- Legal values for <transport-guarantee> include...
  - **NONE** - This is the default. It offers NO data protection. This is what you get when you do not specify any <transport-guarantee>.
  - **INTEGRAL** - Means that the data MUST NOT be changed along the way.
  - **CONFIDENTIAL** - Means that the data MUST NOT be seen by anybody along the way.
- Although not guaranteed by the spec, in practice virtually every Container uses SSL for guaranteed transport, *which means that both INTEGRAL and CONFIDENTIAL do the SAME thing*. Either one gives you both confidentiality and integrity. Since you can have only one <user-data-constraint> per <security-constraint>, some people recommend you use CONFIDENTIAL, but again, it will probably never matter in practice, unless you move to a new (and unusual) Container that doesn't use SSL.

- Remember that in the deployment descriptor, the <security-constraint> is about what happens AFTER the request.
- When a request comes in, the Container looks FIRST at the <transport-guarantee>, and if there IS one, the Container tries to deal with that issue first by asking, "Is this request over a secure connection?" If not, the Container doesn't even bother to look at authentication/authorization info; it just tells the client "come back when you're secure, then we'll talk..."
- Typically 301 redirects are used to send clients to different URLs. But with transport security, instead of redirecting the client to a different URL (resource), the Container redirects the client to the SAME resource but with a DIFFERENT protocol (HTTPS).
- To make sure the user's login info is submitted to the server securely, put a transport guarantee on EVERY constrained resource that could trigger the login process!
  - Remember, when you're using declarative authentication, the client NEVER makes a direct request for the login. The client triggers the login/authentication process by requesting a constrained resource. So, if you want to make sure that your client's login data comes back to the server over a secure connection, you need to put a <transport-guarantee> on EVERY constrained resource that could trigger the login form on the client! That way, the Container will get the request for the constrained resource, but BEFORE telling the browser to get the client's login data.
- While NOT guaranteed by the spec, most Containers use HTTPS over SSL (secure sockets). Using it won't necessarily be automatic. You may need to configure SSL on the Container. For production, you will also need a PKC (Public Key Certificate) from an "official" source such as VeriSign.

## CHAPTER 13 (Filters & Wrappers)

### Filters

- Under the hood, filters use the Intercepting Filter pattern.
- Filters allow you to intercept the request and response without the servlet even knowing.
- Filters are Java components, very similar to servlets, that allow you to intercept and process requests BEFORE they are sent to the servlet, or to process responses AFTER the servlet has completed, but BEFORE the response goes back to the client.
- The only way a filter can be invoked is through a declaration in the deployment descriptor.
- The Container decides when to invoke your filters based on declarations in the deployment descriptor. It's the deployer, NOT the programmer, who maps which filters will be called for which request URL patterns.
- There is only ONE filter interface, Filter...
  - There's no such thing as RequestFilter or ResponseFilter, just Filter.
- Common uses for filters...
  - Request
    - Security checks
    - Reformat request headers or bodies
    - Audit or log requests
  - Response
    - Compress the response stream
    - Append or alter the response stream
    - Create a different response altogether
- Because filters are designed to be totally self-contained, they can be chained together to run one after the other.
- There are three ways in which filters are like servlets...
  - The Container knows their API.
  - The Container manages their lifecycle.
  - They're declared in the deployment descriptor.
- A simple filter example...
 

```
public class ExampleFilter implements Filter {
```

```

private FilterConfig config;
public void init(FilterConfig config) throws ServletException {
 this.config = config;
}
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
 throws ServletException, IOException {
 HttpServletRequest httpRequest = (HttpServletRequest) request;
 ...
 chain.doFilter(request, response);
}
public void destroy() {
 ...
}
}

```

- **Filter** - Every filter must implement the Filter interface.
- **init()** - Most of the time you'll just save the config object.
- **doFilter()** - This is where the real work happens (liken it unto doGet() or doPost()). Notice that this method doesn't take HTTP request or response objects, it takes regular ServletRequest and ServletResponse objects. Be sure to cast BEFORE using these objects.
- **chain.doFilter()** - This is how the next filter or servlet in line gets called.
- **destroy()** - While typically left empty (like with servlets), this is where all the clean up goes.
- The filter's lifecycles...
  - **init()**
    - Called when the Container instantiates the filter. This is where the set-up tasks are performed. Most of the time, it will just save the FilterConfig object for later use.
  - **doFilter()**
    - This method is called every time the Container determines that the filter should be applied to the current request. This is where all the work happens. It takes three arguments...
      - ServletRequest (NOT HttpServletRequest)
      - ServletResponse (NOT HttpServletResponse)
      - FilterChain (Controls the ORDER or FLOW)
  - **destroy()**
    - Called when the Container decides to remove the filter from service. This is where clean-up work is performed.
- The servlet spec does NOT dictate HOW the Container is to handle filter chaining; however, it may help to think of filter chaining as pushing and popping off a stack...

		Servlet A		
	Filter #7	Filter #7	Filter #7	
Filter #3	Filter #3	Filter #3	Filter #3	Filter #3
<b>The Stack</b>	<b>The Stack</b>	<b>The Stack</b>	<b>The Stack</b>	<b>The Stack</b>
<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
The Container calls Filter #3's doFilter() method, which runs until it encounters its chain.doFilter() call.	The Container <i>pushes</i> Filter #7 onto the stack where it executes until it reaches its chain.doFilter() call.	The Container <i>pushes</i> Servlet A's service() method onto the stack where it executes until completion, and is then <i>popped</i> off the stack.	The Container returns control to Filter #7, where its doFilter() method completes (point after the doFilter() call) and is then <i>popped</i> off.	The Container returns control to Filter #3, where its doFilter() method completes, and is <i>popped</i> off. THEN the Container completes the response.

- There are three steps involved with configuring filters in the deployment descriptor...

- **Declaring a filter**

```

<web-app...>
 <filter>
 <filter-name>ExampleFilter</filter-name>
 <filter-class>com.example.ExampleFilter</filter-class>
 <init-param>
 <param-name>userLogFile</param-name>
 <param-value>userLog.txt</param-value>
 </init-param>
 </filter>
</web-app>

```

- **<filter-name>** - This is mandatory.
- **<filter-class>** - This is mandatory.
- **<init-param>** - This is optional and there can be as many as you want.

- **Declaring a filter mapping (to a servlet name or URL pattern)**

```

<web-app...>
 <filter-mapping>
 <filter-name>ExampleFilter</filter-name>
 <servlet-name>ExampleServlet</servlet-name>
 (or)
 <url-pattern>*.jsp</url-pattern>
 </filter-mapping>
</web-app>

```

- **<filter-name>** - This is mandatory and is used to link to the correct <filter> element.
- **<servlet-name>** - Either the <servlet-name> or the <url-pattern> element is mandatory (NOT both). This element defines which SINGLE web app RESOURCE will use this filter.
- **<url-pattern>** - Either the <servlet-name> or the <url-pattern> element is mandatory (NOT both). This element defines which web app RESOURCES will use the filter.

- **Ordering the filters**

- The Container uses the following rules when more than one filter is mapped to a given resource...
  - **ALL filters with matching URL patterns are located first.**
    - This is NOT the same as the URL mapping rules the Container uses to choose the "winner" when a client makes a request for a resource, because ALL filters that match will be placed in the chain! Filters with matching URL patterns are placed in the chain in the ORDER in which they are DECLARED in the deployment descriptor.
    - Once all filters with matching URLs are placed in the chain, **then the Container does the same thing with filters that have a matching <servlet-name>** in the deployment descriptor.

- As of version 2.4, filters can now be applied to request dispatchers (forward, include, request dispatch, error handler) as well.

- **Declaring a filter mapping for request-dispatched web resources**

```

<web-app...>
 <filter-mapping>
 <filter-name>ExampleFilter</filter-name>
 <url-pattern>*.jsp</url-pattern>
 <dispatcher>REQUEST</dispatcher>
 (and/or)
 <dispatcher>INCLUDE</dispatcher>
 (and/or)
 <dispatcher>FORWARD</dispatcher>
 (and/or)
 <dispatcher>ERROR</dispatcher>
 </filter-mapping>
</web-app>

```

- **Note:** You can have from 0 to 4 <dispatcher> elements.
  - **REQUEST** - Activates the filter for **CLIENT requests**. If no <dispatcher> element is present, REQUEST is the default.
  - **INCLUDE** - Activates the filter for request dispatching from an **include()** call.
  - **FORWARD** - Activates the filter for request dispatching from a **forward()** call.
  - **ERROR** - Activates the filter for resources called by the **error handler**.
- Response filters are a bit tricky to implement. They require wrapping the response object in a custom wrapper. Fortunately, there is a support convenience class provided by the API that implements all of the ServletResponse and HttpServletResponse interface methods for you. You only have to override the ones you care about!
- There are four convenience classes provided by the API for dealing with request and response objects...
  - ServletRequestWrapper
  - HttpServletRequestWrapper
  - ServletResponseWrapper
  - HttpServletResponseWrapper

## Wrappers

- A wrapper wraps the REAL request or response object (for example), and then delegates (passes through) calls to the real thing, while still letting you do the extra things you need for your custom request or response.
- Wrappers don't just provide an *interface implementation*; they actually HOLD a reference to an object of the SAME interface type to which they DELEGATE method calls.
- Wrappers are sometimes referred to as decorators. Decorators add new capabilities to the object to which they hold. Under the hood, existing functionality is method-forwarded to the held or wrapped object.

## CHAPTER 14 (Design Patterns)

### Architecture

- A common architecture is configuring the hardware in layers or "tiers" of functionality.
- Adding more computers to a tier is known as HORIZONTAL scaling.
- Most of the software for a big web application lives in either the "Web Tier" or the "Business Tier".

			<b>Web/Presentation Tier</b>		<b>Business Tier</b>	
Client	Firewall	Load Balancer	Web Server Web Server Web Server	Load Balancer	EJB Server EJB Server	Database Legacy
			This is where the servlets and JSPs live. As a website gets more hits, more servers can be added to handle the load.		This is where the business logic lives. More servers can be added when a website needs to handle more volume.	

### JNDI

- JNDI stands for Java Naming and Directory Interface.
- It's an API to access naming and directory services.
- JNDI gives a network a centralized location to find things.
- Objects are registered with the JNDI so that programs on the network can find them (phone book).
- JNDI makes relocating components on the network easier.

- Once you've relocated a component, all you need to do is tell JNDI the new location.
- Network programs only need to know how to find JNDI. They don't actually care where the objects registered with JNDI are actually located.
- JNDI is the central point-of-contact.

## RMI

- RMI stands for Remote Method Invocation.
- It's a mechanism that greatly simplifies the process of getting objects to communicate across a network.
- RMI makes it possible for an object in one JVM to invoke a method on a *remote object* (an object on a *different* JVM). All of this is transparent to the client. In fact, to the client, the method call appears to be local.
- To use RMI...
  - Create a **proxy** and then *register* that object with a **registry**.
    - Create a **remote interface**. This is where the signature for methods like `getCustomer()` will reside. Both the stub (proxy) and the actual model service (the remote object) will implement this interface.
    - Create the **remote implementation**, in other words, the actual model object that will reside on the model server. This includes code that registers the model with a well-known registry service such as JNDI or the RMI registry.
    - Generate the stub and (possibly) skeleton. RMI provides a compiler called **rmic** that will create the proxies for you.
    - Start/run the model service (which will register itself with the registry and wait for calls from remote clients).
  - Clients wanting to perform method calls will need to...
    - Do a *lookup* on the *registry* to get a *copy* of the **remote proxy**.
    - Method calls are made on that remote proxy (a local object to the client).
      - The remote proxy is called a **stub**. It handles all the communication details like...
        - Sockets
        - I/O streams
        - TCP/IP
        - Serialization of method arguments and return values
        - Exception Handling
  - The proxy on the server side is often referred to as a **skeleton**.
- Common patterns associated with RMI include...
  - Business Delegate
    - Hides the "remoteness" complexity.
  - Service Locator
    - Centralizes and hides the JNDI lookup complexity.
  - Transfer Object
    - Improves performance by minimizing network traffic.
      - Every time a JSP invokes a getter on a stub, it's a network call.
        - Network calls are 1000 times more expensive than local method calls.

## Design Patterns

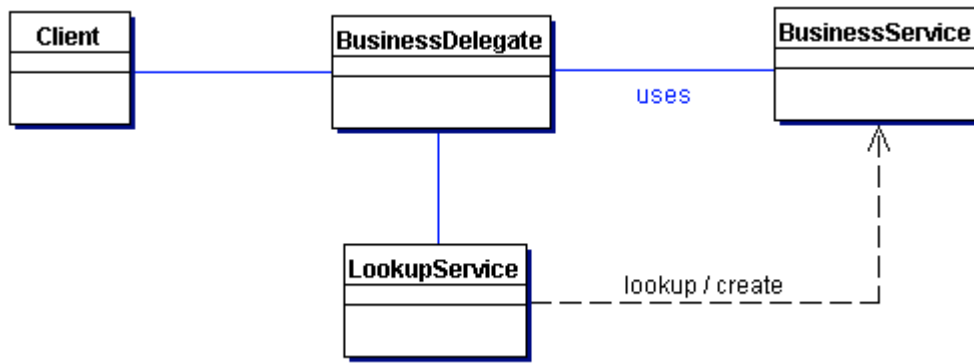
- A software design pattern is "a repeatable solution for a commonly occurring software problem."
- Here are the most important non-functional requirements you're likely to face...
  - Performance
    - Response time
    - Throughput (Number of simultaneous users)
  - Modularity
    - PLUGABILITY of components
    - Ability to mix and match
  - Flexibility

- Ease at which code can be tweaked or modified for MINOR enhancements.
- Maintainability
  - Ease at which SUPPORT changes can be made to existing code.
- Extensibility
  - Ease at which NEW functionality can be incorporated without having to rewrite.
- Common design principles and terminology...
  - Code to interfaces
    - *An interface is a contract between two objects.* When a class implements an interface, it's saying in effect, "My objects can speak your language."
    - Another huge benefit of interfaces is *polymorphism*. Many classes can implement the same interface. The calling object doesn't care who it's talking to as long as the contract is upheld.
  - Separation of concerns and cohesion
    - It's a fact that specialized, single-purpose software components are easier to create, maintain, and reuse. A natural fallout of separating concerns is that cohesion tends to increase. Cohesion means the degree to which a class is designed for one, *cohesive*, task or purpose.
  - Hide Complexity
    - Hiding complexity often goes hand in hand with separating concerns. For instance, if your system needs to communicate with a lookup service, it's best to hide the complexity of that operation in a single component, and allow all the other components that need access to the lookup service to use that specialized component. This approach simplifies all of the system components that are involved.
  - Loose Coupling
    - By their very nature, object-oriented systems involve objects talking to each other. By coding to interfaces, you can reduce the number of things that one class needs to know about another class to communicate with it. The less two classes know about each other, the more *loosely coupled* they are to each other.
  - Remote Proxy
    - Today, when a website grows, the answer is to lash together more servers, as opposed to upgrading a single, huge, monolithic server. The outcome is that Java objects on different machines, in their own separate heaps, have to communicate with each other.
    - Leveraging the power of interfaces, a *remote proxy* is an object local to the "client" object that *pretends* to be a remote object (The proxy is remote in that it is remote from the object it is emulating). The client object communicates with the proxy, and the proxy handles all the networking complexities of communication with the actual "service" object. As far as the client object is concerned, it's talking directly to a local object.
  - Increase declarative control
    - Declarative control over applications is a powerful feature of J2EE Containers. Modifying the deployment descriptor gives the power to change system behaviors without having to modify code.

## Business Delegate

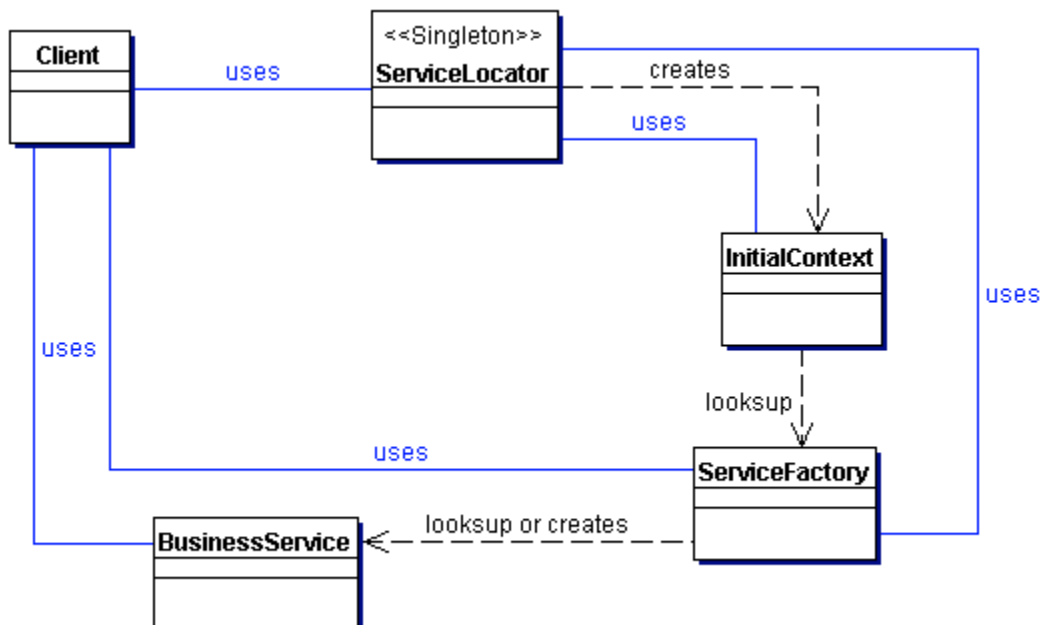
- Use the Business Delegate pattern to shield your web tier controllers from the fact that some of your application's model components are remote.
  - Features
    - Acts as a proxy, implementing the remote service's interface.
    - Initiates communications with a remote service.
    - Handles communication details and exceptions.
    - Receives requests from a controller component.
    - Translates the request and forwards it to the business service (via the stub).
    - Translates the response and returns it to the controller component.
    - By handling the details of remote component lookup and communications, allows controllers to be more cohesive.
  - Principles
    - The Business Delegate is based on...

- Hiding complexity.
- Coding to interfaces.
- Loose coupling.
- Separation of concerns.
- Minimizes the impact on the web tier when changes occur on the business tier.
- Reduces coupling between tiers.
- Adds a layer to the application, which increases complexity.
- Method calls to the Business Delegate should be coarse-grained to reduce network traffic.



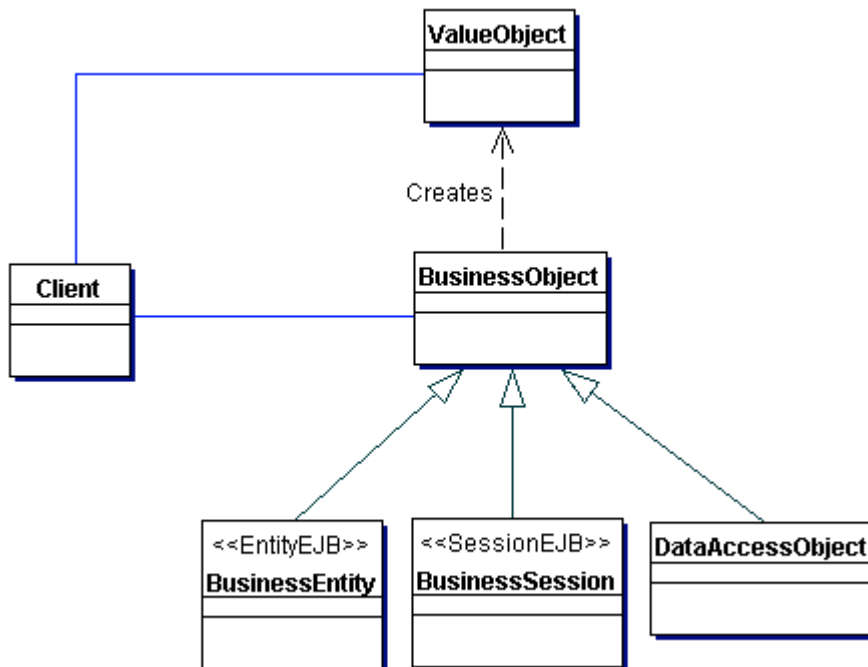
## Service Locator

- Use the Service Locator pattern to perform registry lookups so you can simplify all of the other components (such as Business Delegates) that have to do JNDI (or other registry types) lookups.
  - Features
    - Obtains InitialContext objects.
    - Performs registry lookups.
    - Handles communication details and exceptions.
    - Can improve performance by caching previously obtained references.
    - Works with a variety of registries such as JNDI, RMI, UDDI, and COS naming.
  - Principles
    - The Service Locator is based on...
      - Hiding complexity.
      - Separation of concerns.
    - Minimizes the impact on the web tier when remote components change locations or containers.
    - Reduces coupling between tiers.



## Transfer Object

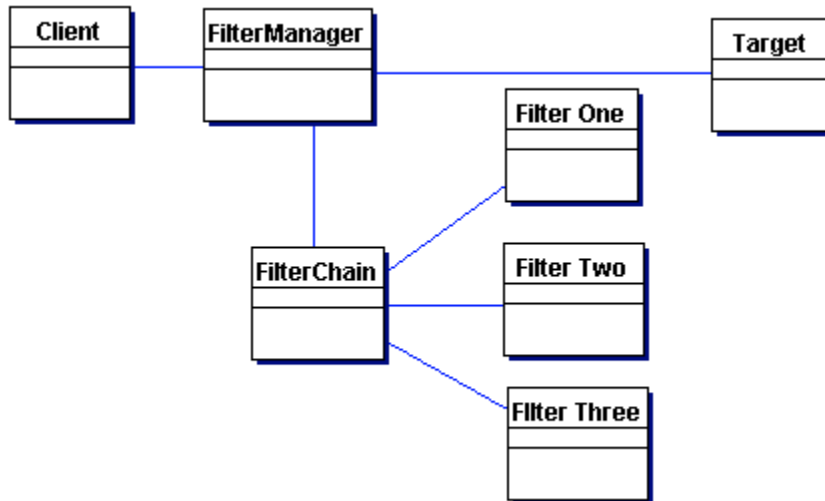
- Use the Transfer Object pattern to minimize network traffic by providing a local representation of a fine-grained remote component (usually an entity).
  - Features
    - Provides a local representation of a remote entity (i.e. an object that maintains some data state).
    - Minimizes network traffic.
    - Can follow JavaBean conventions so that other objects can easily access it.
    - Implemented as a serializable object so that it can move across the network.
    - Typically easily accessible by view components.
  - Principles
    - The Transfer Object is based on...
      - Reducing network traffic.
    - Minimizes the performance impact on the web tier when remote components' data is accessed with fine-grained calls.
    - Reduces coupling between tiers.
    - A drawback is that components accessing the Transfer Object can receive stale (out-of-date) data, because the Transfer Object's data is really representing state that's stored somewhere else.
    - Making updateable Transfer Objects concurrency-safe is typically complex.



## Intercepting Filter

- Use the Intercepting Filter pattern to modify requests being sent to servlets, or to modify responses being sent to users.
  - Features
    - Can intercept and/or modify requests before they reach the servlet.
    - Can intercept and/or modify responses before they are returned to the client.
    - Filters are deployed declaratively using the deployment descriptor.
    - Filters are modular so that they can be executed in chains.
    - Filters have lifecycles managed by the Container.
    - Filter must implement Container callback methods (init(), doFilter(), and destroy()).
  - Principles

- The Intercepting Filter is based on...
  - Cohesion.
  - Loose coupling.
  - Increasing declarative control.
- Declarative control allows Filters to be easily implemented on either a temporary or permanent basis.
- Declarative control allows the sequence of invocation to be easily updated.



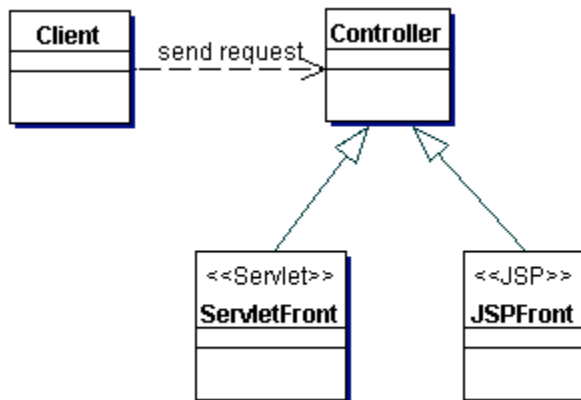
## Model, View, Controller (MVC)

- Use the MVC pattern to create a logical structure that separates the code into three basic types of components (Model, View, and Controller) in your application. This increases the cohesiveness of each component and allows for greater reusability, especially with model components.
  - Features
    - Views can change independently from controllers and models.
    - Model components hide internal details (data structures), from the view and controller components.
    - If the model adheres to a strict contract (interface), then these components can be reused in other application areas such as Swing GUIs or J2ME.
    - Separation of model code from controller code allows for easier migration to using remote business components.
  - Principles
    - MVC is based on...
      - Separation of concerns.
      - Loose coupling.
    - Increases cohesion in individual components.
    - Increases the overall complexity of the application (more to maintain).
    - Minimizes the impact of changes in other tiers of the application.
  - Roles and responsibilities...
    - Model
      - Holds the real business logic and the state. In other words, it knows the rules for getting and updating state.
      - It's the only part of the application that talks to the database.
    - Controller
      - Takes the user input from the request and figures out what it means to the model.
      - Tells the model to update itself, makes the model's state available for the view (the JSP) and forwards to the JSP.
    - View

- Responsible for the presentation. It gets the state of the model from the Controller (although not directly; the Controller puts the model data in a place where the View can find it).

## Front Controller

- Use the Front Controller pattern to gather common, often redundant, request processing code into a single component. This allows the application controller to be more cohesive and less complex.
  - Features
    - Centralizes a web app's initial request handling tasks in a single component.
    - Using the Front Controller with other patterns can provide loose coupling by making presentation tier dispatching declarative.
    - A drawback of Front Controller (on its own, without Struts) is that it's very barebones compared to Struts. To create a reasonable application from scratch using the Front Controller pattern, you would end up rewriting many of the features already found in Struts.
  - Principles
    - The Front Controller is based on...
      - Hiding complexity.
      - Separation of concerns.
      - Loose coupling.
    - Increases cohesion in application controller components.
    - Decreases the overall complexity of the application.
    - Increases the maintainability of the infrastructure code.



- The basic idea of the Front Controller pattern is that a single component, usually a servlet but possibly a JSP, acts as the single control point for the presentation tier of a web application. With the Front Controller, all of the app's requests go through a single controller, which handles dispatching the request to the appropriate places.
- In the real world, it's rare to implement a Front Controller all by itself. Even a really simple implementation usually includes another J2EE pattern called an Application Controller.
- Struts is a popular framework that utilizes a glorified Front Controller with all the bells and whistles.