

SCWCD REVISION NOTES

FREDERIC ESNAULT

CHAPTER 1 : HTTP	4
OVERVIEW :	4
HTTP METHODS :	4
DIFFERENCES BETWEEN GET AND POST	4
CHAPTER 2 : WEB APP ARCHITECTURE	5
CONTAINERS	5
MODEL-VIEW-CONTROLLER (MVC) DESIGN	5
CHAPTER 3 : MINI MVC TUTORIAL	6
APPLICATION CREATION PROCESS	6
CHAPTER 4 : BEING A SERVLET	7
SERVLETS ARE CONTROLLED BY THE CONTAINER	7
SERVLET LIFE	7
METHOD : INIT()	7
METHOD : SERVICE()	8
METHOD : DOGET()/DOPOST()...	8
SERVLETCONFIG	8
SERVLETCONTEXT	9
HTTPSERVLETREQUEST	9
HTTPSERVLETRESPONSE	9
HTTPSERVLETRESPONSE	10
CHAPTER 5 : BEING A WEB APP	11
SERVLET INIT PARAMETERS	11
CONTEXT INIT PARAMETERS	11
SERVLETCONTEXTLISTENER	12
THE EIGHT LISTENERS	13
LISTENERS NOTIFICATION ORDER	13
ATTRIBUTES	14
SYNCHRONIZATION	14
REQUESTDISPATCHER	15
URL MAPPING	16
CHAPTER 6 : SESSION	17
SESSION MANAGEMENT	17
METHODS	17
MANAGING COOKIES	18
SESSION MIGRATION	18
SESSION EVENTS AND LISTENERS	18

CHAPTER 7 : BEING A JSP	20
GENERAL INFORMATION	20
JSP ELEMENTS	20
JSP IMPLICIT OBJECTS	22
GENERATED SERVLET API	22
SCOPES AND ATTRIBUTES	23
OVERRIDING THE JSPINIT() METHOD	23
JSP AND DEPLOYMENT DESCRIPTOR	23
EXPRESSION LANGUAGE	24
CHAPTER 8 : SCRIPTLESS PAGES (EL)	25
STANDARD ACTIONS	25
BEAN DECLARATION	25
BEAN PROPERTY TAGS	26
HIERARCHY NAVIGATION	27
INCLUDE	27
FORWARD	28
INCLUDE AND FORWARD PARAMETERS	28
EXPRESSION LANGUAGE	28
EL SYNTAX	28
EL OPERATORS	29
EL IMPLICIT OBJECTS	30
SCOPE OBJECTS AND [] OPERATORS	30
EL FUNCTIONS	31
EL OPERATORS	32
CHAPTER 9 : JSTL	33
CORE CUSTOM TAGS	33
ERROR PAGES	34
CUSTOM TAG TLD DEFINITION	35
TLD LOADING	36
CHAPTER 10 : CUSTOM TAGS DEVELOPMENT	37
TAG FILES	37
TAG ATTRIBUTES	37
STORAGE	38
TAG HANDLERS	38
SIMPLE TAGS	38
JSP FRAGMENTS	39
CLASSIS TAGS	39
NESTED TAGS	41
DYNAMIC ATTRIBUTES	42
CHAPTER 11 : WEB APP DEPLOYMENT	43
WAR FILES	43
DIRECT ACCESS	43
DEPLOYMENT DESCRIPTOR	43
JSP DOCUMENTS (XML-VIEW)	45
CHAPTER 12 : SECURITY	46

SECURITY CONCEPTS	46
AUTHENTICATION	46
ROLES DEFINITION	46
ENABLING AUTHENTICATION	46
SECURITY CONSTRAINTS	46
PROGRAMMATIC AUTHENTICATION	48
AUTHORIZATION	48
AUTHORIZATION METHOD IMPLEMENTATION.	48
CONFIDENTIALITY	49
AUTHENTICATION AND CONFIDENTIALITY	49
CHAPTER 13 : FILTERS	50
<hr/>	
LIFECYCLE	50
FILTERCHAIN	50
DEPLOYMENT DESCRIPTOR	50
WRAPPER (AKA DECORATOR PATTERN)	51
CHAPTER 14 : DESIGN PATTERNS	52
<hr/>	
BUSINESS DELEGATE	52
SERVICE LOCATOR	52
TRANSFER OBJECT	52
INTERCEPTING FILTER	53
MODEL-VIEW-CONTROLLER	53
FRONT CONTROLLER	54

Chapter 1 : HTTP

Overview :

HTTP stands for HyperText Transport Protocol, and is the network protocol used over the web. It runs over TCP/IP

HTTP uses a request/response model. Client sends an HTTP request, then the server gives back the HTTP response that the browser will display (depending on the content type of the answer)

If the response is an HTML file, then the HTML file is added to the HTTP response body

An HTTP response includes : status code, the content-type (MIME type), and actual content of the response

A MIME type tells the browser what kind of data the response is holding

URL stands for Uniform Resource Locator: starts with a protocol, then a server name, optionally followed by a port number, then the path to the resource followed by the resource name. Parameters may appear at the end, separated from the rest by a ?

HTTP methods

GET	Gets the data identified by an URI
POST	Same, the request's body is passed to the resource specified by the URI
HEAD	Sends back only the header of the response that would be sent by a GET
PUT	Sends to the server data to store at specified URI. Server does not process the data ; it only stores them
DELETE	Delete specified resource
TRACE	When it receives this request, the server sends it back to the client
OPTIONS	Ask server informations about a resource or about its own capabilities
CONNECT	Ask for a connection (tunnelling)

Differences between GET and POST

POST has a body.

GET parameters are passed on the URL line, after a ?. The URL size is limited.

The parameters appear in the browser URL field, hence showing any password to the world...

GET is supposed to be used to GET things (only retrieval, no modification on the server).

GET processing must be idempotent.

A click on a hyperlink always sends a GET request.

GET is the default method used by HTML forms. Use the `method="POST"` to change it.

In POST requests, the parameters are passed in the body of the request, after the header. There is no size-limit. Parameters are not visible from the user.

POST is supposed to be used to send data to be processed.

POST may not be idempotent and is **the only one**.

IDEMPOTENT : Can do the same thing over and over again, with no negative side effect.
--

Chapter 2 : Web App architecture

Containers

A container runs and controls the servlets.

A full J2EE application server must have both a web container and an EJB container.

When a request is received by the webserver and needs to access a servlet, the webserver hands the request to the servlet-helper app : the container. Then the container hands the request to the servlet itself.

Role

- Match the request URL to a specific servlet;
- Creates request and response objects used to get informations from the client and send back informations to him;
- Creates the thread for the servlet;
- Calls the servlet's `service()` method, passing request and response objects as parameters;
 - `service()` method will call the `doXXX()` method of the servlet depending on request type;
- Get the response object from the finished `service()` method and converts it to an HTTP response;
- Destroys request and response objects;
- Sends back response to the webserver

Capabilities of containers

Containers provide :

- **Communications support** : handles communication between the servlets and the webserver
- **Lifecycle management** : controls life and death of servlets, class loading, instantiation, initialization, invocation the servlets' methods, and making servlet instances eligible for GC
- **Multithreading support** : Automatically creates a new thread for every servlet request received. When the Servlet `service()` method completes, the thread dies.
- **Declarative security** : manages the security inside the XML deployment descriptor file. Security can be configured and changed only by modifying this file.
- **JSP support** : Manages the JSPs.

URL to servlet mapping

The servlet container uses the deployment descriptor to map URL to servlets. Two DD elements are used to achieve this :

`<servlet>` : maps internal name to fully qualified class name;

`<servlet-mapping>` : maps internal name to public URL name.

Model-View-Controller (MVC) Design

MVC does not only separates the *Business Logic* from the *presentation*, the *Business Logic* doesn't even know there is a *presentation*.

MVC : separate Business Logic (*model*) and the *presentation*, then put something between them to connect them (*controler*), so the business can become a real re-usable code, and the presentation layer can be modified/changed at will.

Chapter 3 : Mini MVC tutorial

Application creation process

Review the **user's views** and the high level architecture;
Create the **development environment** (if necessary);
Create the **deployment environment** (if necessary);
Perform *iterative* development and testing.

Create user's views and architecture

Define the views (jsp/html) : list of screens reachable by the user.

Create the architecture:

- define the static part (webserver logic, static documents);
- define architecture components (i.e. MVC model, controller and view components) (inside the container)

Create the development environment

A development directory with one subdirectory for each web application.

Each of these subdirectories contain :

- one **etc** directory : deployment descriptor, and other configuration files;
- one **lib** directory : stores all the external jar files needed;
- one **src** directory : where all the source files are;
- one **classes** directory : containing the compiled java files;
- one **web** directory : stores all the web resources (jsp/html/etc...)

Create the deployment environment

The Servlet specification says a web app is deployed in a context root, containing :

- the web resources : html/jsp/etc...
- WEB-INF : holds the deployment descriptor;
- WEB-INF/lib : contains all the external jars needed;
- WEB-INF/classes : contains the compiled java class files with hierarchy.

Create the deployment descriptor

The deployment descriptor, named `web.xml`, must be created in the **development** env. Under the `etc/` directory, then copied into the `WEB-INF/` directory of the **deployment** env.

The `<servlet-class>` element holds the real, fully-qualified (including package) servlet class name.

The `<url-pattern>` holds the user friendly name, the name we want users to use to access the servlet. In this element, the `/` represents the **context root** of the webapp.

In a webapp stored in the *mvc* context-root, the url-pattern :

```
<url-pattern>/login.do</url-pattern>
```

is reached by this url :

```
http://servername:serverport/mvc/login.do
```

The two are linked *via* the `<servlet-name>` element, which is an internal servlet name, reserved to the DD.

Create the model class (POJO)

Create the POJO model class and update the controller servlet to make it use the model to get informations and send it to the client.

Chapter 4 : Being a servlet

Servlets are controlled by the container

Request arrive for a servlet

The container creates the `HttpServletRequest` and `HttpServletResponse` objects

It finds the correct servlet based on the URL, then creates or allocates a thread for that request;

Container calls the servlet's `service()` method, and passes the request and response objects as parameters

The `service()` method calls the `doXXX()` method corresponding to the request HTTP method (GET/POST/...)

Servlet uses the response object to write the response to the client

`service()` method completes, so the thread dies or returns to a Container-managed thread pool. The request and response objects are not referenced any more, so they become eligible for GC.

Servlet Life

Servlets have one main state : initialized.

Servlets are loaded when the container starts up, or when the client first uses the servlet.

Servlet's constructor makes it an object.

`init()` makes it a servlet.

Servlet life steps (controlled by the container):

Loading;

Instantiation (no-args constructor);

`init()`; (Called exactly once in servlet's life)

`service()`; handles client requests

Calls `doGet`, `do Post`,... depending on HTTP method of the request

Each request runs in a separate thread.

`destroyed()`; (Called exactly once in servlet's life)

`service()`, `init()` and `destroy()` are inherited from `Servlet` interface.

`init()` and `destroy()` are implemented in `GenericServlet`

`service()` is implemented in `HttpServlet`

NOTE : There is always only one instance of a servlet (one per JVM). All requests made to this servlet are run in a separate thread.
--

Method : init()

When : Container calls this when the servlet has been instantiated but before it can handle any request.

Why : Perform some initialization

Override : Possible, to initialize some connections (db,etc...) for example.If not, the `GenericServlet`'s one is called. **Only override the no-args version !!**

NOTE : <code>init()</code> always complete before the <code>service()</code> method is called.
--

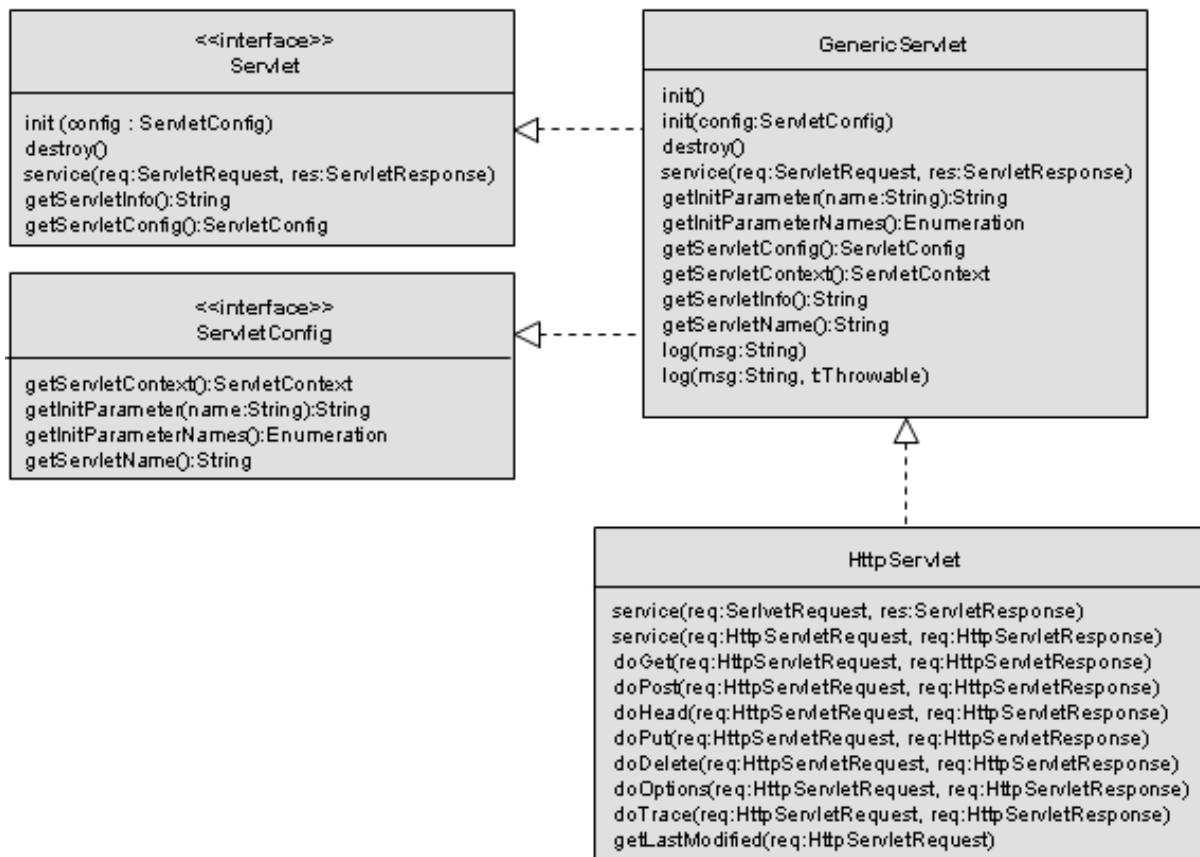
Method : service()

When : Called by the container when a request is sent to the servlet.
Why : The service method determines the HTTP method used from the request and calls the corresponding `doXxxx()` method
Override : No. Shouldn't. If overridden, the `doXxxx()` methods cannot be called anymore. If not, the `HttpServlet`'s one is called.

NOTE : `service()` is always called on its **own stack**.

Method : doGet()/doPost()...

When : Called by the `service()` method when a corresponding HTTP method is used in a request.
Why : Should contain all the actual webapp code to handle the request.
Override : Yes, of course. Any overridden `doXxxx()` method is known as implemented by the webapp. If not, the `HttpServlet`'s one will return an error message saying ***the HTTP method is not implemented***.



ServletConfig

One ServletConfig per servlet
Used for deploy-time information to be given to the servlet
Used to access the ServletContext
Parameters configured in the DD

ServletContext

One ServletContext per webapp

Used to access the webapp parameters (configured in DD)

Can store information for the rest of the webapp servlets (attributes)

Used to get server info, container info, API version info...

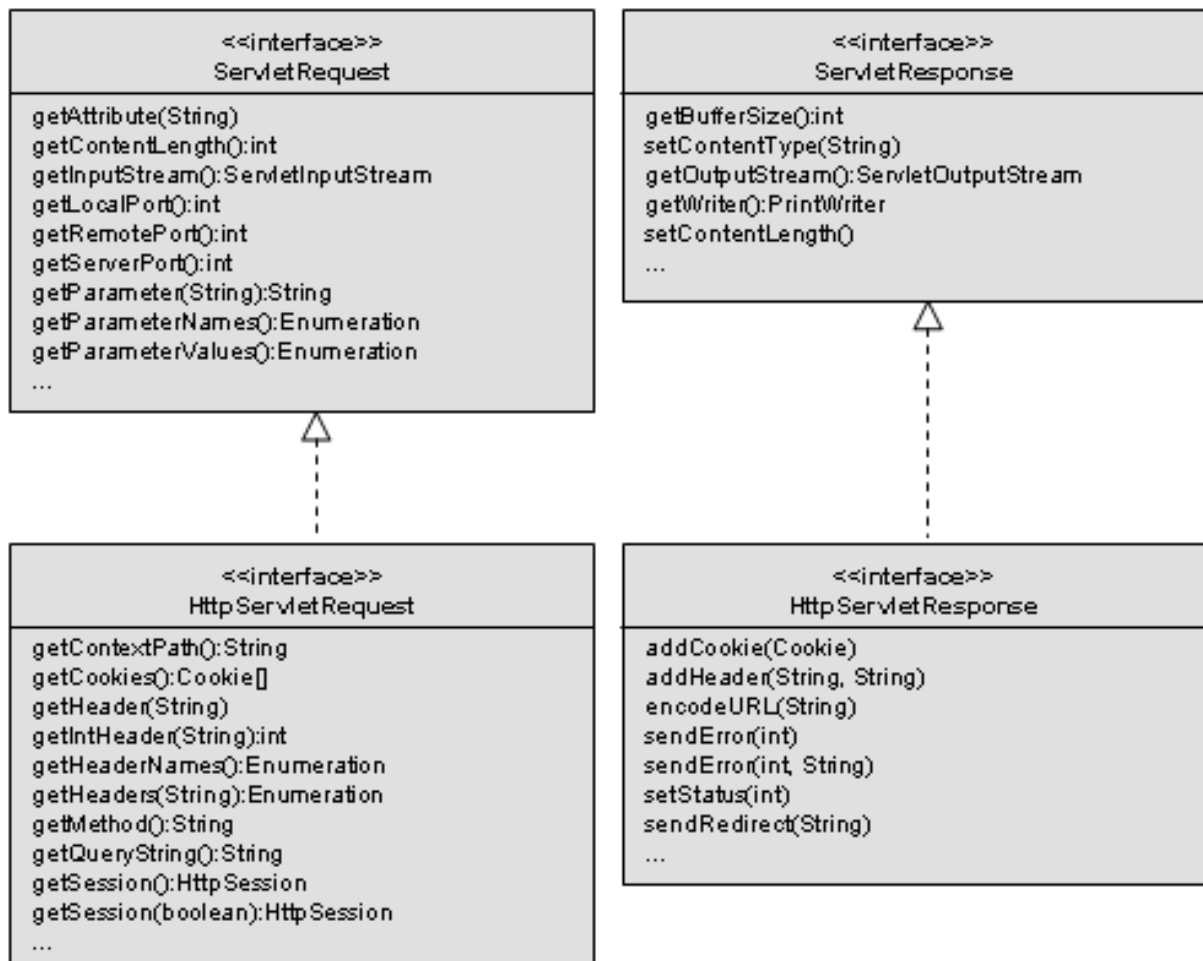
HttpServletRequest

Methods inherited from ServletRequest interface :

```
getAttribute(String)
getContentType()
getInputStream()
getLocalPort()                << port the server passes the request to (one per
thread)
getRemotePort()              << client port
getServerPort()              << port the server is listening to
getParameter(String)
getParameterValues(String)
getParameterNames()
```

Methods inherited from HttpServletRequest interface :

```
getCookies()
getHeader(String):String
getHeaders(String):String[]
getIntHeader(String):int & getDateHeader(String):Date
getHeaderNames():Enumeration
getQueryString()
getSession()
```



HttpServletResponse

Response points :

Used to send data back to the client

Most common methods are `setContentType()` and `getWriter()` (NOT `getPrintWriter()`)

`getWriter()` allows to make character I/O to the response stream

The `PrintWriter` writing method is : `println(String)`

Response can be used to set headers, add cookies or send errors

Usually we use JSP to send HTML data, but the response stream can be used to send binary data to the client

The method used to get an `OutputStream` is `getOutputStream()`

The `ServletOutputStream` writing method is : `write(byte[])`

`setContentType()` allows the browser to know how to handle the data. (text/html, application/pdf, image/jpeg, ...)

```
setHeader() <<    overwrites the existing value or creates it.
addHeader() <<   adds an additional value or creates it.
```

To redirect a request, use `sendRedirect(url:String)`. The redirection is made by the client.

`sendRedirect()` cannot be called if the response has already been committed (flushed).

`sendRedirect(String)` : the url string may start with a /, to force URL to be relative to the root of this web container, or without, to make the url string relative to the current request path.

A request dispatch happens on the server, and hands the request to another server component.

Methods inherited from `ServletResponse` interface :

```
setContentType()
getOutputStream()
getWriter()
setContentLength()
```

NOTE : there is *NO* other method to get the writer or the `OutputStream`. Only **`getOutputStream()`** and **`getWriter()`** are correct.

Methods inherited from `HttpServletResponse` interface :

```
addCookie()
addHeader()
encodeURL()
encodeRedirectURL()
sendError()
setStatus()
sendRedirect()
```

NOTE :

`println()` to a `PrintWriter`
`write()` to a `ServletOutputStream`

NOTE : Drop the first word from the type to make the method name

```
ServletOutputStream
    response.getOutputStream();
PrintWriter
    response.getWriter();
```

NOTE : `sendRedirect(url:String)` takes a **STRING** argument, not an URL object.

Chapter 5 : Being a Web App

Servlet Init parameters

Defined in the DD, in the `servlet` element :

```
<init-param>
  <param-name>...</param-name>
  <param-value>...</param-value>
</init-param>
```

Accessible in the servlet via the `ServletConfig` : `getServletConfig()` : inherited from `Servlet` interface.

```
getServletConfig().getInitParameter(name:String)
```

NOTE : init parameters are read ONCE and passed ONCE to the servlet, when it's initialized, in the `ServletConfig` parameter of the `init()` method.
They cannot be used in the servlet before it's initialized.

NOTE : init parameters are for one servlet only, so they're defined in an `<init-param>` block, nested in the `<servlet>` block

Context init parameters

Context init parameters are just like servlet init parameters, except that they are available to the whole web app, not just one servlet.

They are defined in the DD, in a `<context-param>` block :

```
<context-param>
  <param-name>...</param-name>
  <param-value>...</param-value>
</context-param>
```

They are accessible via the `ServletContext` object, that one can get from the `getServletContext()` method, inherited from the `ServletConfig` interface.

```
getServletConfig().getServletContext().getInitParameter(name:String)
```

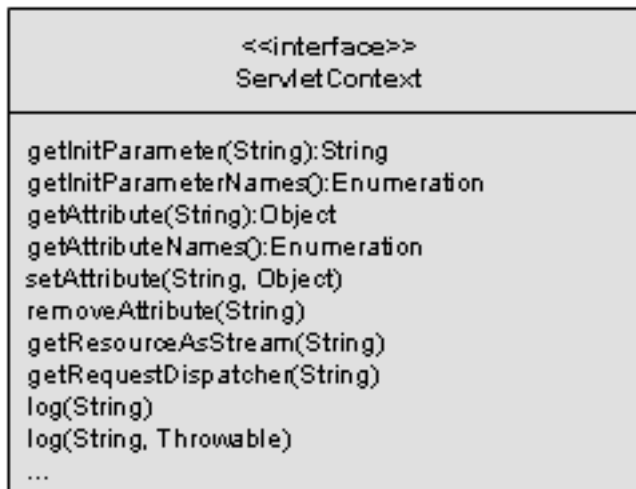
NOTE : context init parameters are for the whole web app, so they are NOT nested in the `<servlet>` element. The `<context-param>` block is not nested in anything except the `<web-app>` root element.

NOTE : servlet and context parameters are accessed via the same method name :
`getInitParameter(String)`
Servlet init parameters are accessed via the `ServletConfig`
Context init parameters are accessed via the `ServletContext` (accessed via `ServletConfig`)

NOTE : there is ONE `ServletConfig` per `SERVLET`
And there is ONE `ServletContext` per `WEB APP`

NOTE : By default, assume that "init parameter" stands for "servlet init parameter"

Servlet and context init parameters must be seen as DEPLOY-TIME constants.
There are getter methods to retrieve them, but there is NO setter



ServletContextListener

The ServletContextListener receives context initialization and destruction events.
It's an interface with two methods :

```
public void contextInitialized(ServletContextEvent);  
public void contextDestroyed(ServletContextEvent);
```

A ServletContextEvent has one method : `getServletContext()`

A listener is declared in the web app DD, in a `<listener>` block.

```
<listener>  
  <listener-class>...</listener-class>  
</listener>
```

The ServletContextListener allows to mess with context attributes (not parameters), which can be initialized, modified, retrieved and even removed, via the **ServletContext** methods :

```
getAttribute(String):Object  
getAttributeNames():Enumeration  
setAttribute(String, Object)  
removeAttribute(String)
```

NOTE : `getAttribute(String): Object` returns an **OBJECT**, a cast is necessary to access it.

The Eight Listeners

Listener Interface	Event
<code>javax.servlet.ServletContextAttributeListener</code> <code>attributeAdded()</code> <code>attributeRemoved()</code> <code>attributeReplaced()</code>	<code>ServletContextAttributeEvent</code>
To be notified when web app context attributes are added, removed, replaced.	
<code>javax.servlet.http.HttpSessionListener</code> <code>sessionCreated()</code> <code>sessionDestroyed()</code>	<code>HttpSessionEvent</code>
To be notified of session's creation/destruction.	
<code>javax.servlet.ServletRequestListener</code> <code>requestInitialized()</code> <code>requestDestroyed()</code>	<code>ServletRequestEvent</code>
To be notified when a servlet request comes in, to be able to log it.	
<code>javax.servlet.ServletRequestAttributeListener</code> <code>attributeAdded()</code> <code>attributeRemoved()</code> <code>attributeReplaced()</code>	<code>ServletRequestAttributeEvent</code>
To be notified when a request attribute has been added, removed, replaced.	
<code>javax.servlet.http.HttpSessionBindingListener</code> <code>valueBound()</code> <code>valueUnbound()</code>	<code>HttpSessionBindingEvent</code>
To be notified when <i>this</i> object is bound/unbound to/from the session (poor's man EJB)	
<code>javax.servlet.http.HttpSessionAttributeListener</code> <code>attributeAdded()</code> <code>attributeRemoved()</code> <code>attributeReplaced()</code>	<u>HttpSessionBindingEvent</u>
To be notified when an <i>object</i> is added, removed or replaced from the session.	
<code>javax.servlet.ServletContextListener</code> <code>contextInitialized()</code> <code>contextDestroyed()</code>	<code>ServletContextEvent</code>
To be notified of the initialization & destruction of the web app context.	
<code>javax.servlet.http.HttpSessionActivationListener</code> <code>sessionDidActivate</code> <code>sessionWillPassivate</code>	<u>HttpSessionEvent</u>
To detect when an object must be notified the session it's bound to is going to migrate to another JVM.	

NOTE : The `HttpSessionBindingListener` **MUST NOT** be described in the DD. It's called by the Container anyway.

Listeners notification order

The container uses the DD to determine the notification order of the registered listeners. At load-time and during life, the listeners are notified in the order they were registered (order of appearance in the DD).

NOTE : At web app shutdown time, the listeners are notified in the **reverse** order.

Attributes

An attribute is a name/value pair (name is a `String`, value is an `Object`).
Attributes can be bound to one of the three other servlet objects (except `Servlet`):
`HttpSession`;
`(Http)ServletRequest`;
`ServletContext`.

Attribute/Parameter differences

Attributes can be bound to context, request, session.
Parameters can be bound to context, request, servlet.

Attributes can be set with `setAttribute(String, Object)`.
Parameters cannot be set (only read once from DD).

Attributes return type is `Object`.
Parameters return type is `String`.

Attributes are retrieved with `getAttribute(name:String):Object`. **A cast is needed.**
Parameters are retrieved with `getInitParameter(name:String):String`.

Attribute scope

- **Context** : available to everyone in the web app.
- **Session** : accessible to those who have access to the specific `HttpSession`.
- **Request** : accessible to those who have access to the specific `ServletRequest`.

Attribute API

Attribute API is the same for the three objects that can handle attributes.

```
getAttribute(String):Object  
setAttribute(String, Object)  
removeAttribute(String)  
getAttributeNames():Enumeration
```

They belong to the interfaces :

- `ServletContext`
- `ServletRequest` << NOT `HttpServletRequest`, but inherited
- `HttpSession`

Synchronization

Context synchronization

Context attributes are not Thread-safe. They can be modified by other thread using the same servlet instance, or threads using other servlets' instances. To synchronize the context attributes and make the context thread-safe, the best way is to synchronize on the `ServletContext` object.

```
synchronized (getServletContext())
```

NOTE : This make the context thread-safe ONLY if ALL the servelts synchronize on the context.

Session synchronization

Session sounds thread-safe, but a client could actually open two browsers and send in two request with the same session id.

The same method should be used to make it thread-safe. We synchronize on the HttpSession object.

```
synchronized (request.getSession())
```

Thread-safe attributes

Thread-safe attributes are :

- Request attributes;
- Method local variables.

Not thread-safe attributes :

- Context attributes;
- Session attributes;
- Servlet instance variables;
- Servlet static variables.

What Not To Do

What is really bad about thread protection attempt is :

- Use the SingleThreadModel;
- Synchronize the service method.

Because it makes the servlet run for only ONE client at a time, reducing the web app performances, and still do *NOT* protect the context attributes, while another thread may still access them.

RequestDispatcher

Request attributes make sense when we want another webapp component to take over the request. For example, controller servlet gets info from the model, and passes it to the JSP view with a request attribute (which makes sense because this information is needed in the request only, not in the session or in the context).

To pass the request to another web app component, use the RequestDispatcher.

RequestDispatcher can be accessed via a ServletRequest method:

```
RequestDispatcher dispatch = request.getRequestDispatcher(url:String)
```

where url is the url (represented as a STRING) of the next component.

Or it may be obtained from the ServletContext object :

```
RequestDispatcher view = getServletContext().getRequestDispatcher(url:String)
```

Or

```
RequestDispatcher view = getServletContext().getNamedDispatcher(name:String)
```

Then forward with the forward method:

```
view.forward(request, response);
```

Or we can include the resource specified by the url with the include() method.

```
view.include(request, response);
```

The forward and include methods set request attributes like:

```
javax.servlet.forward.request_uri
javax.servlet.forward.query_string
...
```

NOTE : These attributes are not set if the RequestDispatcher has been obtained via the `getNameDispatcher()` method.

From context, or from request?

The RequestDispatcher can be obtained from the ServletRequest or from the ServletContext objects. The difference is that from the **request**, the component url **may** be **relative** to the current resource. From the **context**, the url **MUST** be absolute, i.e. it **MUST** begin with a slash (/).

IllegalStateException

A request **CANNOT** be forwarded **AFTER** the response has been sent/committed to the client. The `flush()` method actually sends the response, so there is no forwarding possible after that. If tried, the Container will send an **IllegalStateException**.

But the `include()` method do **NOT** suffer this restriction. It can be called anytime.

Url Mapping

The mapping rules applied to the `<url-pattern>` of the DD are :

A string beginning with a `'/'` character and ending with a `'/*'` suffix is used for path mapping.

A string beginning with a `'*.'` prefix is used as an extension mapping.

A string containing only the `'/'` character indicates the "default" servlet of the application. In this case the servlet path is the request URI minus the context path and the path info is null.

All other strings are used for exact matches only.

Chapter 6 : Session

Session management

Sessions are identified by a **JSESSIONID**.

Two ways can be used to manage cookies:

Cookies;

URL encoding.

Cookies

Cookies are the default mechanism to manage sessions.

The cookie is exchanged in the request/response header. Its name **MUST** be **JSESSIONID**.

URL encoding

When cookies don't work (browser refusing cookies), the URL encoding method is used. It needs a little bit work from the developer, as the urls must be encoded to add the session ID at the end of the request.

URL rewriting ALWAYS works.

A container first tries BOTH cookies and URL encoding, then depending on cookies success, the container uses cookies or encoding.

Encoding URL is handled by the response object:

```
response.encodeURL(...)  
response.encodeRedirectURL(...) << encoding version of sendRedirect()
```

Methods

Session creation :

```
request.getSession()  
request.getSession(bool) << if bool==false, return an existing session or null.  
                           << if bool==true, works like getSession().
```

Session methods :

```
isNew() << returns true if the client didn't send a  
        response with this session ID  
getCreationTime()  
getLastAccessedTime()  
setMaxInactiveInterval(int) << -1 works like invalidate, time must be set in  
seconds  
getMaxInactiveInterval()  
invalidate() << invalidates the session immediatly  
setAttribute(String, Object)  
getAttribute():Object
```

Setting time out in the DD :

```
<session-config>  
  <session-timeout>minutes</session-timeout>  
</session-config>
```

Managing cookies

Cookies are added to a response, and retrieved from a request object.
Cookies are a name/value pair.

```
HttpServletRequest.getCookies():Cookie[]  
HttpServletResponse.addCookie(Cookie)
```

NOTE : There is **NO** such method as : addCookie(name:String, value:String). **NO!**

Cookie main methods

Cookies methods are :

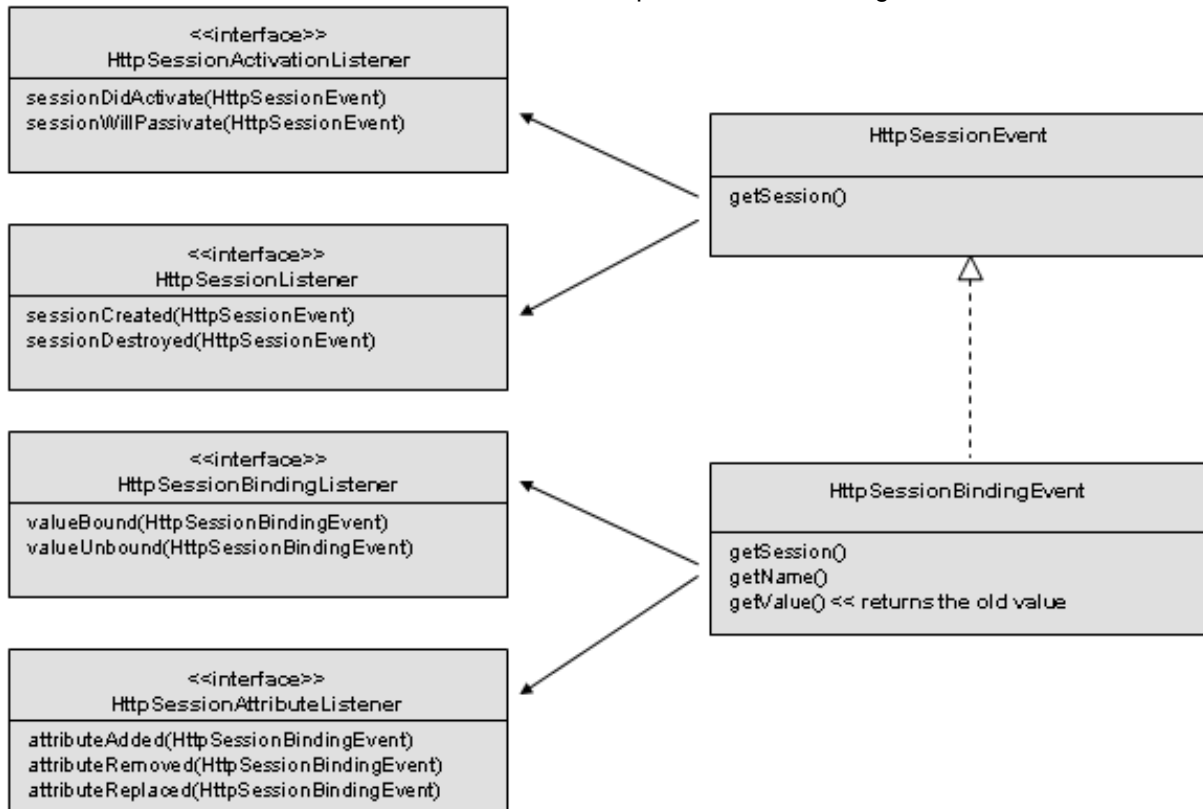
```
getName():String  
getValue():String  
getMaxAge():int  
setMaxAge(int) << -1 means cookie disappear when browser is closed.
```

Session migration

If a web app is distributed, only HttpSession moves from one VM to another (and it's GUARANTEED).
There is ONE ServletContext per VM, and ONE ServletConfig per servlet, per VM.
Before a session is migrated, a passivation event is thrown.
Just after it activates, an activation event is thrown.
Those events can be listened to by the HttpSessionActivationListener.

Session events and Listeners

The listeners and events dedicated to sessions are represented in this diagram :



Session migration and serialization

If attributes of a session are `Serializable`, they ARE migrated (guaranteed by the specs).

If not, the `HttpSessionActivationListener` must be used and the listener methods must be used as one would use the `writeObject/readObject` methods.

```
writeObject()    <>    sessionWillPassivate(HttpSessionEvent)
readObject()     <>    sessionDidActivate(HttpSessionEvent)
```

Chapter 7 : Being a JSP

General information

Exam objective: Know exactly how every part of JSP code is used during translation to the final Servlet.

JSP are translated into servlet and compiled the first time they are accessed. This translation/compilation process happens only once in the JSP life.

Classpath

For classes used in JSPs, the container looks for them in **WEB-INF/classes/** directory. Any package information is based on this classpath root.

JSP elements

Template text

Any raw text such as html tags and user text (i.e. *User email is :*) is called *template text*.

Directives

Semi-colon : NO

Directives are informations given to the container about how the translation of the JSP to a servlet should be achieved. Directives have this syntax :

```
<%@ directive {attr="value"}* %>
```

White spaces are allowed between `<%@` and the directive, and after the attributes and the `%>`.
No white space is allowed in the `<%@` and `%>` strings.

The three directives are :

- page
- taglib
- include

The page directive

The page directive defines a number of page properties and communicates these to the web container. The exam-scope attributes are :

- `import` : like java import; creates import statements in the final servlet. The syntax allows multiple imports in one directive, separated by a comma (,).

```
<%@ page import="org.phoenix.scwd.*, org.toto.truc.*" %>
```

- `isThreadSafe` : default=true; indicates if the servlet is thread safe. If not, it will use the `SingleThreadModel` interface, which is **bad AND** deprecated in servlet specs 2.4.
- `contentType` : defines MIME type and character encoding of the response. Default is "text/html" for MIME, and "ISO-8859-1" for character encoding.
- `isELIgnored` : default=false(2.4) or true(2.3); indicates whether EL expressions within the JSP must be ignored. The default value depends on the servlet spec version defined in the DD.

- `isErrorPage` : default=false; indicates if this JSP is the target of another JSP `errorPage`. If true, then the `exception` implicit object is available. If false, `exception` is not available, and any use attempt of this object creates **a translation error**.
- `errorPage` : defines the URL of a resource that should be called when an uncaught `Throwable` is raised. The `Throwable` object is passed in the `ServletRequest` object as an attribute. Attribute name is `javax.servlet.jsp.JspException` (backward compatibility) and `javax.servlet.error.exception` (match servlet spec).

The include directive

The include directive is used to substitute text and/or code at JSP page translation-time. The directive inserts the text of the specified resource into the page or tag file.

```
<%@ include file="relativeURLspec" %>
```

The taglib directive

The set of significant tags a JSP container interprets can be extended through a tag library. The `taglib` directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.

If a JSP container implementation cannot locate a tag library description, a fatal translation error shall result. It is a fatal translation error for the `taglib` directive to appear after actions or functions using the prefix.

Declaration :

```
<%@ taglib ( uri="tagLibraryURI" | tagdir="tagDir" ) prefix="tagPrefix" %>
```

Usage :

```
<super:doMagic> ... </super:doMagic>
```

Scriptlet

Semi-colon : YES

Scriptlet are meant to hold plain java code. So, ; is mandatory.

Scriptlet content is pasted **AS-IS** in the `_jspService()` method.

Any variable declared in a scriptlet is a **LOCAL** variable (not initialized,...).

Syntax :

```
<% java code %>
```

Expressions

Semi-colon : NO

An expression element in a JSP page is a scripting language expression that is evaluated and the result is coerced to a `String`. The result is passed as a parameter to an `out.print()` statement in the final `_jspService()` method. So of course, ; is **NOT** allowed.

Syntax :

```
<%= expression %>
```

Declarations

Semi-colon : YES

Declarations are used to declare member variables and methods in the final servlet. The content is pasted AS-IS in the servlet member declaration area. It must be correct java code, so ; is mandatory.

Syntax :

```
<%! Declaration of methods&vars %>
```

JSP implicit objects

Within a JSP code (meant to be used in the service method of a generated servlet), as in any servlet, some objects are available, such as the ServletConfig, the ServletContext, ... In the JSP, these objects are retrieved automatically and set as implicit object. Here they are with their API type :

```
JspWriter           out ;           1
HttpServletRequest   request ;
HttpServletResponse response ;
HttpSession         session ;
ServletContext       application ;   2
ServletConfig       config ;
JspException        exception ;
PageContext         pageContext ;
Object              page ;
```

¹ JspWriter is similar to PrintWriter but not in the same package.

² There is no **context** variable. It's **application**, because ServletContext refers to the WebApp context.

Generated servlet API

The servlet methods take no argument. They are called from the Servlet normal methods.

jspInit()

Called by the init(...) method. ServletConfig and ServletContext objects ARE available.
This method may be overridden.

jspDestroy()

Called by the destroy() method.
This method may be overridden.

_jspService()

Called by the service() method. This is where the expressions and scriptlets are pasted.
This method **MAY NOT** be overridden.

NOTE : All methods starting with an **underscore (_)** may not be overridden.

Scopes and attributes

The available scopes for a JSP are the same as for servlet, plus one.

- PAGE_SCOPE (commonly used for *custom tags*)
- REQUEST_SCOPE
- SESSION_SCOPE
- APPLICATION_SCOPE

Each scope can hold attributes. The Jsp runs in a PageContext, which is a subclass of JspContext. The Page context describes the constants for each scope (PAGE_SCOPE, REQUEST_SCOPE, ...). The JspContext defines methods to get/set/remove attributes as usual, plus method to find attributes in other scopes.

```
findAttribute(name:String):Object
getAttribute(name:String):Object          /\ PAGE_SCOPE ONLY
getAttribute(name:String, scope:int):Object
getAttributeNamesInScope(scope:int):Enumeration
getAttributeScope(name:String):int
getOut():JspWriter
```

The `findAttribute(name:String)` method searches for the attribute which name matches the method parameter in the PAGE_SCOPE. If it doesn't find it, it goes up the scopes, from the more restrictive to the most restrictive (REQUEST, SESSION then APPLICATION). **The first attribute matching the parameter wins.**

Overriding the `jspInit()` method

The `jspInit()` method may be overridden exactly the same way one would declare any method in a JSP, by using the declaration `jsp` element.

```
<%! public void jspInit() {
    // override
} %>
```

Jsp and Deployment descriptor

Jsp init parameters

Like servlets, Jsp, can receive some init parameters. This is done exactly the same way one would declare init param for servlets, except one tag differs. `<servlet-class>` is replaced by `<jsp-file>`.

The philosophy is : *Bind these parameters to the servlet class represented by this jsp file.*

```
<servlet>
  <servlet-name>MyJsp</servlet-name>
  <jsp-file>myJsp.jsp</jsp-file>
</servlet>
```

Disabling scripting

To avoid the use of java code (and all jsp elements like directives, declarations, expressions and scriptlets) in a jsp, one may disable scripting in the DD.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>          << disable for all JSPs (*.jsp)
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

NOTE : Usage of scripting elements in a JSP where scripting is disabled result in a **translation error**.

Expression Language

EL expressions are **ALWAYS** like this :

```
{ expression }
```

Content of EL expressions is translated to java code and put in the `_jspService()` method.

NOTE : EL is **ENABLED** by default.

Invalidating EL

Like Scripting, EL expressions can be disabled. There are two ways to disable EL :

In the DD :

Use the same declaration block as for scripting, except the `<scripting-invalid>` tag is replaced by `<el-ignored>`.

In a page directive :

Using the `isELIgnored` attribute:

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

NOTE : the DD tag is **<el-ignored>**, NOT **<is-el-ignored>**

Invalidating priority

As there are two ways to disable EL, one must be priority.

NOTE : The **page directive** takes priority over the DD setting.

Chapter 8 : Scriptless Pages (EL)

Standard actions

Bean declaration

The standard action tag for bean declaration is :

```
<jsp:useBean>
```

Declares a bean which can be used in the rest of the JSP.

Attributes are :

- id;
- class;
- type;
- scope.

If the attribute is not found, the `useBean` action will create one using the attribute `class` constructor.

id attribute

The id attribute specify the name of the attribute to be retrieved, as set in the servlet with the `setAttribute()` method. This name will be used in the JSP to reference the attribute bean.

```
request.setAttribute("titi", obj) <> <jsp:useBean id="titi" .../>
```

scope attribute

The scope attribute specify in which scope the attribute is to be looked for. For a **request** attribute name `titi`, usage should be :

```
<jsp:useBean id="titi" ... scope="request"/>
```

Available scopes are :

- page;
- request;
- sessions;
- application.

NOTE : default value for `scope` attribute is : **page**.

class attribute

The class attribute defines the type of the attribute **object** (the actual instance). For an attribute named `titi` which type is `org.phoenix.MyType`, usage is :

```
<jsp:useBean id="titi" class="org.phoenix.MyType" .../>
```

`class` value **MUST** be a concrete class, and must have a no-arg constructor (because a bean may be created if missing).

type attribute (optional)

The optional type attribute is used to specify the type of the **reference** to the actual attribute object instance. If attribute `titi` object is of type `Employee` which is a subclass of `Person`, one could use :

```
<jsp:useBean id="titi" type="Person" class="Employee" .../>
```

type value may be a class, an abstract class or an interface.

class & type attributes

If class is used : class value must NOT be abstract, and MUST have a no-arg constructor.

If type is used alone : the attribute MUST exist (one will NOT be created).

<p>NOTE : Don't confuse <code>class</code> & <code>type</code>. <code>type</code> = reference type. <code>class</code> = object type. <code>type attr = new class();</code></p>

useBean action body

The `<jsp:useBean>` standard action may have a body. Instead of this :

```
<jsp:useBean ... scope="page" />
```

one can use it like this :

```
<jsp:useBean ... scope="page" />
...
</jsp:useBean>
```

The `useBean` tag body is executed IF and ONLY IF the attribute is not found AND one is created. If the `useBean` tag is used with the `type` attribute alone (with no `class`) attribute, and the bean is not found, an object will **NOT** be created (see `type` attribute) so the `useBean` tag body is **NOT** executed.

<p>NOTE : <code>useBean</code> body is executed ONLY IF the bean attribute is NOT found AND one IS created.</p>

Bean property tags

Standard actions provide tags for bean properties (getters and setters).

getProperty tag

This tag takes two attributes :

- `name` : the bean name, reference to the `useBean` id attribute;
- `property` : the bean property name. Based on the bean getter **method** naming convention.

It returns the value of the bean property.

setProperty tag

This tag takes four attributes :

- `name` : the bean name;
- `class` : the bean type;
- `property` : name of the bean property;
- `value` : the value to give to the bean property;
- `param` : name of a request parameter (param value is used to set property value).

NOTE : When a useBean tag is used with a setProperty in its body and the attribute is not found, it results in the creation of a new bean object, and the call of a setter method on it.

The value attribute may be replaced by the param attribute. In this case, the parameter value is used as the new property value.

NOTE : if bean property name and request parameter name match, param may be ignored. The matching parameter will be used to set matching property's value.

```
<jsp:setProperty name="titi" property="matching_name"/>
```

NOTE : property attribute can be set to the * wildcard: all matching property/parameter pairs will be used.

All properties are filled, EVEN **inherited** properties.

```
<jsp:setProperty name="titi" property="*/>
```

Automatic conversion

With the *parameter-to-property* facility, String parameters are automatically converted to primitives if the matching property is of a primitive type.

NOTE : If the parameter value is retrieved by a **scripting expression**, the conversion will **NOT** be done. An **explicit cast** is required.

```
<jsp:setProperty name="n" property="prim" value="<%= request.getParameter("prim") %>" />
```

=> DOES NOT COMPILE.

Hierarchy navigation

Standard actions allow to get/set bean properties if they are String or primitive. But they don't allow navigation in the object hierarchy (properties of property of a bean). Expression Language must be used.

Include

Includes may be used within a Jsp. Two mechanisms are available :

- the include directive;
- the include standard action.

include directive

The include directive is used to insert the included file's source into a Jsp at translation time. This actually copies and pastes the included file's source into the current Jsp. The include directive uses the *file* attribute.

```
<%@ include file="header.jsp" %>
```

include standard action

The include standard action is used to insert the included page's result (output) into the the current Jsp at **runtime**. At translation time, the generated servlet uses kind-of an `include()` method call.

The include standard action uses the `page` attribute, because it deals with a runtime *page*, not a file.

```
<jsp:include page="header.jsp" />
```

Forward

A Jsp may forward the request to another webapp component with the forward mechanism. This can only be achieved using a standard action : `<jsp:forward/>`. The `forward` standard action uses the `page` attribute, like the `include` standard action.

```
<jsp:forward page="invalidUser.jsp" />
```

Like the `RequestDispatcher` `forward` method, the response must NOT have been committed before the forward is called. If response has been committed, then the client will receive the committed response only. The forward **WILL** throw an `IllegalStateException`, but no one will see it.

NOTE : if response has not been committed, it's cleared before the forward is done.
--

Include and forward parameters

Like the `RequestDispatcher`'s `include()` and `forward()` methods, the corresponding standard actions may add parameters to the request object they transmit. This is achieved by adding a body tag to the `include/forward` tag. The body tag is `<jsp:param>`. It takes two attributes :

- `name` : the parameter name;
- `value` : the parameter value (*duh!*)

```
<jsp:forward page="invalidUser.jsp" ><< NO ending slash !
  <jsp:param name="userName" value="..." />
</jsp:forward>
```

This works the same for `<jsp:include>`.

Expression Language

EL Syntax

The EL syntax is ALWAYS like this :

```
${firstArg.secondArg}
```

EL first argument

The first argument appearing in an EL expression MUST be one of these :

- One of EL implicit objects;
- An attribute in one of the 4 scopes.

EL second argument

The second argument of an EL expression MUST be :

- A map key;
- A bean property.

EL operators

EL provides two navigation operators :

- The **dot** operator;
- The **[]** operator.

dot operator

Syntax :

```
leftVariable.rightVariable
```

The dot operator is simple and restrictive. The rules of the dot operator are :

- a variable on the left of a dot is :
 - a Map;
 - a bean.
- a variable on the right of a dot is :
 - a map key;
 - a bean property.

The name of the right hand side variable must respect Java naming conventions :

- starts with a letter, a \$, or a _
- after first character, digits may be used
- can't be a java keyword.

[] operator

Syntax :

```
leftVariable["content"]
```

Rules are :

- a variable on the left of the [] may be :
 - a Map;
 - a bean;
 - a List;
 - an array.
- the content of the [] may be :
 - a Map key;
 - a bean property;
 - an index into a List;
 - an index into an array.

For indices into Lists/arrays, the quotes can be omitted :

```
myList[2]
```

If quotes are used with indices, the String index is automatically coerced into an int :

```
myList["2"]
```

If content is a String and quotes are NOT used, EL assumes it's an attribute and replaces it by its value, or by null if it's not found.

```
toto[titi]
```

```
titi : assumed to be an attribute >> will be evaluated.
```

EXAM NOTE : Watch out quote vs no quotes !!!

The [] operator allows nested expressions. Evaluation is made from inner most [] outer.

```
${titi[toto[foo[bar]]]}
```

Operators usage

Both operators may be used by :

- beans;
- Maps.

NOTE : Lists and array may ONLY use the [] operator.

`#{myList.1}` is **ILLEGAL** : use `#{myList[1]}` or `#{myList["1"]}`

EL implicit objects

EL provides implicit objects :

- Maps of scopes' attributes:
 - `pageScope`;
 - `requestScope`;
 - `sessionScope`;
 - `applicationScope`.
- Maps of request's parameters :
 - `param` : used when there is ONE value for ONE param name
 - `paramValues` : used when ONE param name has MULTIPLE values.
- Maps of request's headers :
 - `header` : same as param, for header name/value
 - `headerValues` : same as paramValues, for header name/values
- Map of cookies' values :
 - `cookie`
- Map of CONTEXT init parameters :
 - `initParam`
- reference to the pageContext object :
 - `pageContext`.

NOTE : `initParam` is for **CONTEXT** parameters, set in the DD with the `<context-param>` block. They do NOT give access to servlet parameters.

NOTE : Scope objects are NOT the matching objects. `requestScope` is NOT the `request` object. Same for `sessionScope/HttpSession`, `applicationScope/ServletContext`.

Scope objects and [] operators

Sometimes, scopes' attributes get non-Java conform names. It's then impossible to use them with the dot operator. If an attribute has been named : `org.phoenix.request.email`, it cannot be used like this :

```
#{org.phoenix.request.email}
```

EL will try to find an attribute named `org`, which it will not find.

The solution is to use the matching scope object and get the attribute with the [] operator :

```
#{requestScope["org.phoenix.email"]}
```

EL functions associate Java public static methods to EL callable symbols. EL functions usage is :

1. Create a java class with a public static method. The method MUST be **public AND static**, and it may have parameters. The return type shouldn't be void, but it's not an error.
2. Write a tag library descriptor file (tld). This file creates the mapping between the actual java public static method and the EL accessible function name. The two names may be *different*.
3. Put a taglib directive into the JSP file. It describes the uri of the linked *tld* and the prefix to associate to it.
4. Use the EL function by calling the **function name** as defined in the `<name>` tag of the tld.

The Java class

The java class holding the method is a plain old java class. The method defined MUST be public AND static.

The class must be stored in the **WEB-INF/classes/** directory or one of its subdirectories. It may be stored in a jar file too.

The Tag Library Descriptor

The tld file creates the mapping between the actual java public static method and the EL function name. Function description structure is :

```
<uri>myTldUri</uri>
<function>
  <name>elFunctionName</name>
  <function-class>org.phoenix.tld.TldMethodClass</function-class>
  <function-signature>java.lang.String myMethod(int i)</function-signature>
</function>
```

Comments :

1. EL function name is defined in a `<name>` tag (NOT `<function-name>`);
2. EL function name and actual Java method name MAY be different;
3. The method signature uses fully qualified types for objects.

Tld files must be deployed in the **WEB-INF/** directory or one of its subdirectories. It may be packaged into a jar file also.

The taglib directive

To use an EL function in a JSP, it must be declared with the `taglib` directive. The taglib directive takes two attributes :

- `prefix` : defines the prefix to associate to THIS tld library within the CURRENT Jsp;
- `uri` : refers to the `uri` declared in the tld file.

NOTE : uri value is NOT the name of the tld file.

```
<%@ taglib prefix="phoenix" uri="myTldUri" %>
```

EL function call

Eventually the EL function is called using the `prefix:functionName` syntax :

```
${phoenix:elFunctionName(6)}
```

NOTE : This is NOT java : do NOT use ;

EL provides its own operators to perform small logic (business logic shouldn't be in the *view*). These operators are arithmetic, logical and relational.

Arithmetic operators

The arithmetic operators are :

- **+**
- **-**
- *****
- **/** or **div**
 - division by zero returns *infinity*; it does **NOT** throw an exception.
- **%** or **mod**
 - modulo by zero **DOES** throw an `ArithmeticException`

Logic operators

Logic operators are :

- **&&** or **and**
- **||** or **or**
- **!** or **not**

Relational operators

Relational operators are :

- **==** or **eq**
- **!=** or **ne**
- **>** or **gt**
- **<** or **lt**
- **>=** or **ge**
- **<=** or **le**

EL reserved words

More than the already seen operators, EL has special reserved words that **cannot** be used as identifiers :

- `true`
- `false`
- `instanceof`
- `null`
- `empty`

EL and `null`

EL behaves nicely with `null`.

Within String evaluations, `null` is evaluated to an *empty string*.

Within arithmetic operations, `null` is evaluated to `0`.

Within logic operations, `null` is evaluated to *false*.

Summary :

```
null may be evaluated as :  
an empty String;  
0;  
false.
```

Chapter 9 : JSTL

Core Custom Tags

c:forEach

Allows iteration through a Collection, a Map, an array (even coma-delimited String, but *deprecated*).

```
<c:forEach[var="varName" ] items="collection" [varStatus="varStatusName" ]  
[begin="begin" ] [end="end" ] [step="step" ]>
```

NOTE : var has a *nested* visibility.

- begin and end attributes : int values, indices in the iteration (first element is 0);
- varStatus holds the current iteration status (iteration index).

c:if

```
<c:if test="testCondition" [var="varName" ]  
[scope="{page|request|session|application}" ]>
```

c:choose

Linked to c:when and c:otherwise.

NOTE : Only **ONE** c:when is executed, even if two test evaluate to true.

c:set

Syntax 1: Set the value of a scoped variable using attribute value

```
<c:set value="value"  
var="varName" [scope="{page|request|session|application}" ]/>
```

Syntax 2: Set the value of a scoped variable using body content

```
<c:set var="varName" [scope="{page|request|session|application}" ]>  
  body content  
</c:set>
```

Syntax 3: Set a property of a target object using attribute value

```
<c:set value="value"  
target="target" property="propertyName" />
```

Syntax 4: Set a property of a target object using body content

```
<c:set target="target" property="propertyName">  
  body content  
</c:set>
```

var version : (Attribute)

- var attribute value is the **NAME** of the attribute (String type)
- if attribute doesn't exist AND value is not null, the attribute is created;
- if attribute is found AND value is null, the attribute is removed.

target version : (Bean or Map)

- target attribute value **MUST NOT** be null; it **MUST** be the attribute **OBJECT** (NOT its name);

c:remove

```
<c:remove var="varName"
[scope="{page|request|session|application}"/>
```

Removes the attribute.

c:import

```
<c:import url="url" [context="context" ]
[var="varName" ] [scope="{page|request|session|application}" ] >
```

Adds contents of imported resource to the current page at request time.
May receive `c:param` tags in its body.

URL may be relative to current context, absolute, or relative to another context (context specified in the `context` attribute).

c:url

Handle url, with automatic url rewriting (session tracking) and implicit url encoding when used with `c:param`.

```
<c:url value="value" [context="context" ]
[var="varName" ] [scope="{page|request|session|application}"/>
  <c:param> subtags encoded
</c:url>
```

Error pages

page directive attributes :

- `isErrorPage: true` means exception implicit object is accessible and initialized;
- `errorPage` : declares the page to which to redirect in case uncaught exception occurs.

DD error pages

The `error-page` block allows to declare one or more error pages for any type of exception:

```
<error-page>
  <exception-type> : fully qualified exception class (java.lang.Throwable
catches everything)
  or
  <error-code> : allows to map error codes to specific error pages
  <location> : location of error page, relative to webapproot (MUST start with
/ )
</error-page>
```

NOTE : Declare an error page in the DD doesn't make it an error page (ie. Exception implicit object is NOT accessible).

c:catch

the `c:catch` custom tag allows to wrap risky code into *sort-of* a `try/catch` block.

```
<c:catch [var="varName" ]>
  risky code
</c:catch>
```

NOTE : the `var` attribute allows to keep the exception in a page scoped attribute.

NOTE : The code AFTER the exception is NEVER executed.

Custom tag TLD definition

TLD is equivalent to a custom tag API. Every thing needed to use a custom tag is in the TLD.

TLD attributes :

- `tlb-version` : mandatory;
- `short-name` : mandatory;
- `uri` : mandatory; uniquely identifies the taglib in the container taglibs map.

NOTE : `uri` is a NAME; it is NOT a location.

Tag description :

```
<tag>
  <description>
  <name>
  <tag-class>
  <body-content>
  <attribute>
    <name>
    <type>
    <required>
    <rtexprvalue>
  </attribute>
</tag>
```

- `name` : declares the tag name (as in `<prefix:name>`);
- `tag-class` : fully qualified tag handler class name;
- `body-content` : specifies if tag allows content in its body and what kind of content;
- `attribute` : one attribute block per tag attribute;
- `name` : attribute's name;
- `required` : true means attribute is mandatory;
- `rtexprvalue` : (RunTime EXPRESSION VALUE) : Whether the attribute's value may be dynamically calculated at runtime or it must be a String literal. Default is **false**. An attribute allowing `rtexprvalue` can take :
 - EL expressions : `user = "${userName}"`
 - Scripting **expressions** : `user = "<%=.....%>"`
 - The `<jsp:attribute>` standard action (**MUST** be put in the tag body, even if `body-content` is declared `empty`).

NOTE : In a JSP, each taglib must be loaded in one taglib directive with a UNIQUE prefix.

Body content

The body content may take 4 values :

- `empty` : no body is allowed. Tag may appear in three ways :
 - `<p:tag />`
 - `<p:tag></p:tag>`
 - `<p:tag><jsp:attribute name="..."></jsp:attribute></p:tag>`
- `scriptless` : Can hold EL, standard and custom actions, but no scripting elements :
 - expressions `<%=`
 - scriptlets `<%`
 - declarations `<%!`
- `tagdependent` : contents is treated as plain text; EL, SA and CA are NOT evaluated;
- `JSP` : everything allowed in a JSP is allowed.

Before JSP 2.0

Before the JSP 2.0 spec, tld had to be declared in DD :

```
<taglib>
  <taglib-uri>blah</taglib-uri>
  <taglib-location>...</taglib-location>
</taglib>
```

Since JSP 2.0

Notinh has to be declared in DD. Container loads `tld` files and constructs a uri/file map.

TLD files locations

To be automatically loaded, tld files must be stored :

- In WEB-INF/
- In a WEB-INF/ subdirectory
- In META-INF/ of a JAR file stored in WEB-INF/lib/
- In a META-INF/ subdirectory of a JAR file stored in WEB-INF/lib/

taglib directive

In taglib directive, no reserved prefix may be used :

- java
- javax
- jsp
- jspc
- sun
- sunw

NOTE : If mutiple taglib are used, <code>uris</code> MUST be unique.
--

Chapter 10 : Custom tags development

Tag files

Allows to create custom tags for content inclusion.

Tagfiles are JSP fragments renamed with the **.tag** extension.
Made accessible in a JSP with the `taglib` directive, using `tagdir` attribute.

```
<%@ taglib prefix="..." tagdir="..." %>
```

`tagdir` attribute : directory containin the Tag files.

Usage :

```
<prefix:TagFileName/>
```

TagFileName is the tag file name minus the **.tag** extension.

Tag attributes

Tag may be sent attributes using two ways :

- tag attributes;
- tag body contents.

Attributes

Attributes are declared in the tag file itself, using the `attribute` directive.

```
<%@ attribute name="attrName" required="true" rtexprvalue="true" %>
```

NOTE : if `required` is set to `true` and attribute is not present, a runtime error occurs.

In Jsp :

```
<prefix:tagName attrName="blah" />
```

Body content

Big values may be given to tag files using their body :

```
<prefix:tagName>
  This is the tag value passed in its body
</prefix:tagName>
```

In the tag file, body value is displayed using the `doBody` standard action :

```
<strong> <jsp:doBody/> </strong>
```

The kind of value a tag's body content may receive can be declared with the `tag` directive (like the `page` directive, but for tag files). `tag` directive offers same attributes as `page` directive, plus one : `body-content`.

```
<%@ tag body-content="scriptless|empty|tagdependant" %>
```

Default value is `scriptless`.

NOTE : A tag body CANNOT use scripting elements.

Storage

Tag files may be deployed in :

- WEB-INF/tags/ directory (or one of its subdirectories);
- META-INF/tags (or one of its subdirectories) in a JAR file in the WEB-INF/lib/ directory.

NOTE : If a tag file is deployed in a JAR file, it **MUST** be declared in a tld file.

Tag file TLD entry

In the tag lib descriptor file, tag files may be described like this :

```
<tag-file>
  <name>tagFileName</name>
  <path>/META-INF/tags/ tagFileName.tag</path>
</tag-file>
```

Tag Handlers

Tag files implement their functionality in another page (JSP).

Tag handlers implement their functionality in a special java class.

Simple tags

Extends SimpleTagSupport, implements SimpleTag

Lifecycle

Container :

- Loads the simple tag handler class;
- Instantiate class (no-args cstr);
- Calls setJspContext(JspContext);
- If tag is nested, calls the setParent(JspTag) method;
- If attributes, calls the attributes setter methods;
- If tag declared to have a body and is NOT empty : calls setJspBody(JspFragment) method
- Calls doTag() method.

NOTE : Simple tags handlers are NEVER reused; each call has a NEW instance.

Response writer

Writer can be obtained with :

```
getJspContext().getOut()
```

NOTE : Return value of getJspContext is a JspContext. **Needs a cast** to access PageContext objects/methods.

Body content processing

To process tag's body contents, call (in doTag()) :

```
getJspBody().invoke(java.io.Writer)
```

NOTE : invoke argument is null, output goes to response.

Uninitialized EL expressions

If tag body contents has an EL expression resolving to an unknown variable, this variable may be set in the doTag() method of the simple tag handler. Iterations may be made to process multiples values for the same EL variable.

Tag attributes

Tag attributes must be defined in the TLD file with the `attribute` block :

```
<attribute>
  <name>tagAttribute</name>
  <required>...</required>
  <rtexprvalue>...</rtexprvalue>
</attribute>
```

Tag Handler setter method respect the JavaBean naming convention :

```
setTagAttribute(...)
```

Stop page processing

Simple tag handlers may stop further page processing by throwing a `SkipPageException`.

The rest of the page will not be processed. However, everything that was already processed WILL appear in output.

NOTE : For included JSP invoking simple tags : if tag throws a <code>SkipPageException</code> , ONLY the included page processing is stopped. The including page continues its normal processing.

JSP Fragments

JSP fragment MUST NOT contain ANY scripting element.

JSPFragment defines two methods :

```
getJspContext():JspContext
invoke(java.io.Writer)          /\ if writer is null, output goes to the response
```

Classis tags

Extends `TagSupport` or `BodyTagSupport`, implements `Tag`, `IterationTag`, `BodyTag`.

Lifecycle

Container :

- Loads the classic tag handler class;
- Instantiate class (no-args cstr);
- Calls `setPageContext(PageContext)`;
- If tag is nested, calls the `setParent(Tag)` method;
- If attributes, calls the attributes setter methods;
- Calls the `doStartTag()` method
- If tag declared to have a body **and** is NOT empty **and** `doStartTag` returns `EVAL_BODY_INCLUDE` : evaluates body
- If body was evaluated, calls `doAfterBody()` method;
- Calls `doEndTag()` method.

NOTE : Classic tags handlers MAY be reused; one instance my be pooled and used more than once.
--

NOTE : doStartTag() and doEndTag() are **ALWAYS** called once and **ONLY ONCE**. doAfterTag() may be called zero to many times.

Response writer

Writer can be obtained with :

```
pageContext.getOut()
```

Body content processing

To process tag's body contents, doStartTag MUST return :

```
EVAL_BODY_INCLUDE
```

Stop page processing

Classic tag handlers may stop further page processing by returning a SKIP_PAGE value in the doEndTag() method.

Methods return types

Classic tag methods return types are int values.

doStartTag()

EVAL_BODY_INCLUDE	Tag body is evaluated
SKIP_BODY	Tag body is NOT evaluated (default)

doAfterTag()

EVAL_BODY_AGAIN	Tag body is re-evaluated
SKIP_BODY	calls doEndTag (default)

doEndTag()

SKIP_PAGE	Stops page processing (like SkipPageException)
EVAL_PAGE	Goes on with page processing (default)

NOTE : doStartTag() **MUST** be overridden if body is to be evaluated, because default return value of doStartTag() is : **SKIP_BODY**.

NOTE : if tld file declares that the tag body content is empty, then doStartTag() **MUST** return **SKIP_BODY**.

BodyTagSupport

BodyTagSupport is a subclass of TagSupport. It adds three methods and changes doStartTag() default return value to a new return value.

New return value :

```
EVAL_BODY_BUFFERED
```

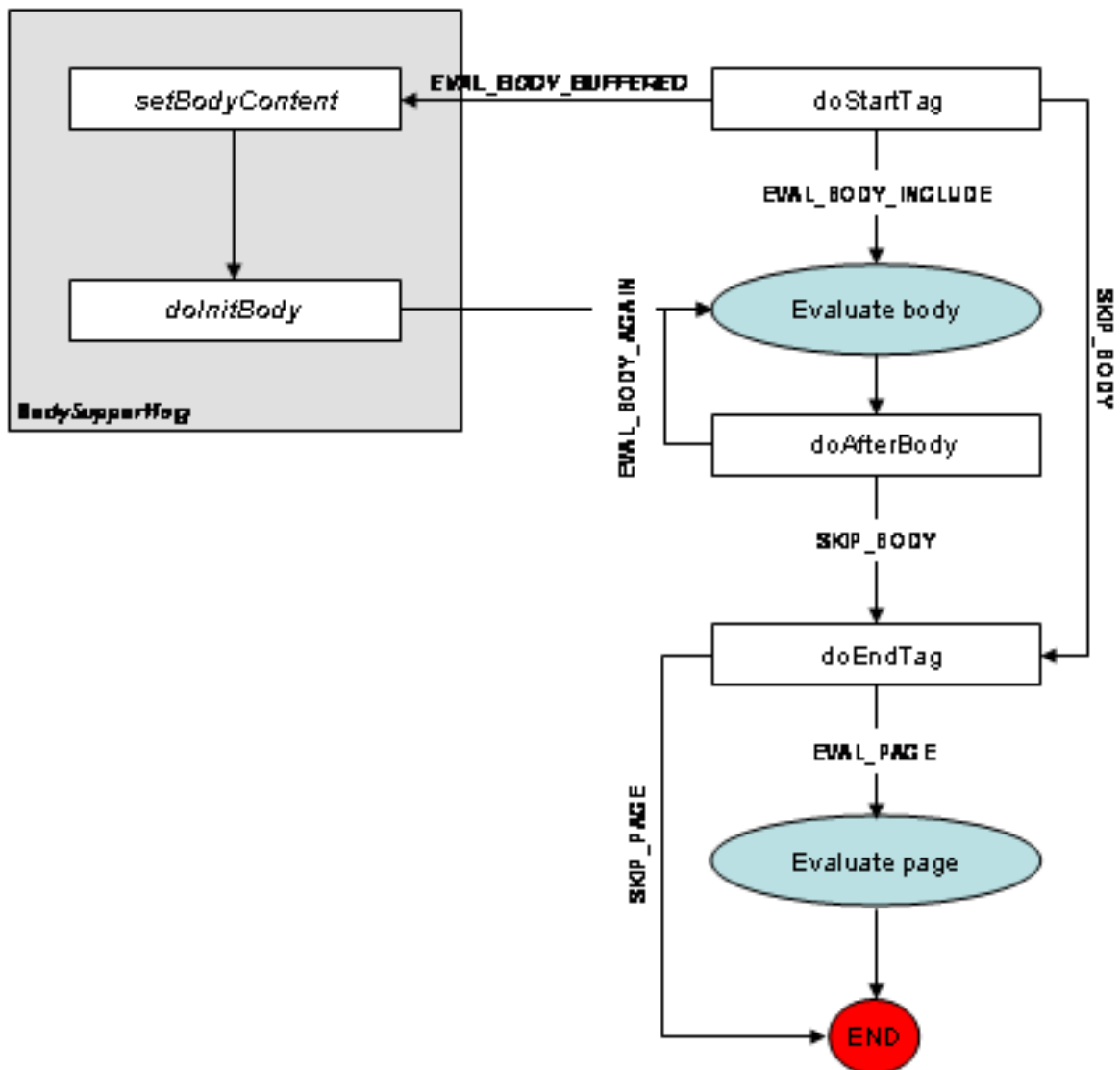
doStartTag() default return type :

```
EVAL_BODY_BUFFERED
```

New methods :

```
setBodyContent (BodyContent )  
doInitBody ()  
getPreviousOut () :JspWriter
```

Classis tag handler algorithm



Nested tags

Nested tags handlers may access their parent tag handler with :

```

getParent():JspTag << Simple Tag
getParent():Tag << Classic Tag
  
```

NOTE :getParent() returns a (Jsp)Tag; a cast IS NEEDED to access its methods/values.

Simple & classic tag mix

Simple tags and classic tags may be mixed in a nesting hierarchy. Parent access, for inheritance reasons, is reduced :

NOTE :

Classic tags can access classic tags parents **only**;
 Simple tags can access simple **and** classic tags (as Tag extends JspTag).

Nested tags and body skipping

If a tag contains nested tags but its body is skipped (SKIP_BODY), then the nested tags are **NOT** processed.

Tag ancestors

A tag can find one of its ancestors with a specific static method :

```
findAncestorWithClass(Tag, Class)
```

Returns the first ancestor which class is the method argument.

Dynamic attributes

Simple tags and classic tags can handle dynamic attributes. Two things are necessary to allow dynamic attributes in custom tags :

- Declare dynamic attributes in DD;
- Implement DynamicAttributes interface.

DD declaration

To declare the use of dynamic attributes, one element must be added in the tag definition :

```
<dynamic-attributes>true</dynamic-attributes>
```

Custom tag class

The corresponding custom tag class must implement the `DynamicAttributes` class, which defines one method :

```
public void setDynamicAttribute(String uri, String localName, Object value)  
throws JspException
```

Most common implementation is to store the attributes in a `<String, Object>` map with the `localName` as key, and the value object as value.

Chapter 11 : Web app deployment

War files

A webapp may be deployed as a directory structure or as a **WAR** file. A WAR file is a Web Archive. It's just a JAR containing a web app directory structure.

NOTE : All rules about web app directory structure remain **unchanged** in a WAR.

Only difference is WAR files contains a **META-INF/** directory at the same level as WEB-INF/. This directory contains a Manifest.mf which can be used to declare **libraries and classes dependencies**.

Direct access

Anything placed under the **WEB-INF/** directory, or under the **META-INF/** of a **WAR** file is **protected** against direct access from client requests.

Deployment descriptor

Servlet mapping

Servlet uses two DD blocks :

- `<servlet>` : declares the servlet internal name and the servlet class;
- `<servlet-mapping>` : associates a url-mapping to the servlet internal name.

URL patterns

Containers looks for matches in DD for three different patterns, in the following order :

- exact match
 - leading /
 - may have an extension (not mandatory)
- directory match
 - leading /
 - real/virtual path
 - ends with /*
- extension match
 - **NO** leading /
 - starts with a *
 - followed by a "dot extension" (ex : *.do)

NOTE : if exact match not found, the longest mapping wins.

Welcome files

Each time no servlet mapping is found, the container takes the directory specified in the request uri and tries to find a welcome file in the list declared in the DD :

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

NOTE : **NO** leading / in the welcome file declaration. This is just the **NAME** of a file, **NOT** a **PATH**.

Error pages

Error pages are declared in the DD with the `error-page` tag :

```
<error-page>
  <exception-type>fully qualified exception class</exception-type>
OR
  <error-code>404</error-code>
  <location>/error.jsp</location>
</error-page>
```

NOTE : `exception-type` **OR** `error-code` may be used. **NEVER** both.

NOTE : location value **MUST** begin with a /.

To handle everything, catch **java.lang.Throwable**. It catches everythin, not only JSP errors.

NOTE : the DD error pages may be overridden by the `page` directive's `error-page` attribute.

Startup loading

To load servlets at container start-up (or web app reload), use the `load-at-startup` tag in the servlet block :

```
<servlet>
  ...
  <load-at-startup>x</load-at-startup>
</servlet>
```

Values of x :

- negative or 0 : no preloading; load at first request;
- 1 or more : servlet loaded at startup.

NOTE : if **x** has different positive values for different servlets, it declares the loading order of servlets. If two or more servlet have the same **x** value, they're loaded in the order they are declared in the DD.

EJB

EJB declaration :

```
<ejb-local-ref>          <ejb-ref>
  <ejb-ref-name>          <ejb-ref-name>
  <ejb-ref-type>          <ejb-ref-type>
  <local-home>           <home>
  <local>                 <remote>
</ejb-local-ref>        </ejb-ref>
```

JNDI environment

Declaration of JNDI environment entries (quite like context attributes for JNDI).

```
<env-entry>
  <env-entry-name>      JNDI lookup name
  <env-entry-type>      any type using a String as a constructor arg.
  <env-entry-value>     will be passed as a String
</env-entry>
```

NOTE : `<env-entry-type>` may **NEVER** be of a **primitive** type.

Mime type

Mime type declaration for the web app :

```
<mime-mapping>
  <extension>
  <mime-type>
</mime-mapping>
```

NOTE : **NO** file-type, **NO** content-type : **mime-type**.

JSP documents (XML-view)

XML correspondance of JSP elements :

Directives	<code><%@ page import="...></code>	<code><jsp:directive.page import="... </></code>
Declaration	<code><%! int x ;></code>	<code><jsp :declaration> int x; </></code>
Scriptlet	<code><% ...></code>	<code><jsp:scriptlet> ... </></code>
Text	Email is :	<code><jsp:text>Email is : </></code>
Scripting expr.	<code><%= it.next() %></code>	<code><jsp:expression>it.next()</></code>

Chapter 12 : Security

Security concepts

The four J2EE security concepts are :

1. authentication
2. authorization
3. confidentiality
4. data integrity

Authentication

Realm : place where authentication information is stored :
Memory realm, JDBC realm...

Roles definition

Two steps are necessary :

1. declare login/passwords and associated roles in a vendor specific place;
2. declare roles in the webapp DD.

For step 2, in DD under web-app :

```
<security-role>
  <role-name>admin</role-name>
  <role-name>guest</role-name>
</security-role>
```

Enabling authentication

Login configuration is declared in DD :

```
<login-config>
  <auth-method>...</auth-method>
</login-config>
```

auth-method is one in :

- BASIC : most simple protection, weak encryption (base64)
- DIGEST; better encryption; NO guarantee that container implements this one
- CLIENT-CERT; very strong encryption, uses Public Key Cryptography; requires client certificate
- FORM : like BASIC, but login/pwd form is customizable; login/pwd info are NOT encrypted.

Security constraints

Security constraint declaration in DD is the way to declare protected/restricted **REQUESTs**.

NOTE : Constraints are applied to resource/http method pair; not only to the resource.

One or more security-constraint block may be declared. Each of them allows to declare :

- Web resource collection : resource name, resource/http method pair;
- Authorized roles : allows declaration of roles allowed to make requests defined by resource/http method pairs;

```

<security-constraint>
  <web-resource-collection>
    <web-resource-name>blah</web-resource-name>
    <url-pattern>/toto/*</url-pattern>           // one or more
    <http-method>GET</http-method>             // zero or more
  </web-resource-collection>
  <auth-constraint>
    <role-name>guest</role-name>               // zero or more
  </auth-constraint>
</security-constraint>

```

Security constraint rules

Web-resource-collection has two primary sub-elements :

- url-pattern;
 - 1 or more
- http-method;
 - 0 or more.
 - If zero, all method are constrained;
 - If not zero, only declared methods constrained; other method NOT constrained.

NOTE : url-pattern AND http-method together define the security constraint.
At least ONE url-pattern is REQUIRED.

NOTE : A web-resource-name is MANDATORY. A description is OPTIONAL.

One or more web-resource-collection may be defined in one security-constraint block.
The auth-constraint declared applies to ALL web-resource-collection blocks.

NOTE : auth-constraint does NOT define who has access to the resources;
auth-constraint define who can make the constrained REQUESTs.

NOTE : ***Restrictions apply to requests made from OUTSIDE the webapp.*** Other webapp elements MAY access the constrained requests with no authentication.

role-name rules

- role-name element are optional;
- if role-name present, only the declared roles are allowed to make the requests;
- if role-name not present, **NO USER** is allowed;
- <role-name>*</role-name> : **ALL** users are allowed
- role names are case sensitive

auth-constraint rules

- auth-constraint is optional;
- if auth-constraint present, associated urls **WILL** require authentication;
- if auth-constraint not present, **NO** authentication is required.
- Empty <auth-constraint/> element : **NO** user allowed.
- A description element is optional.

NOTE : <auth-constraint/> (empty) is DIFFERENT from NO auth-constraint element.
<auth-constraint/> : no user allowed;
no auth-constraint : all users allowed (NO authentication).

auth-constraint combinations

If multiple web-resource-collection blocks are declared and refers to the same resource, the auth-constraint declaration follow these rules :

- If each block declare a list of users, the allowed users are the UNION of the two lists;
- If one auth-constraint has a wild-card role-name (*) : ALL users are allowed;
- If one auth-constraint is empty (<auth-constraint/>) : NO user allowed;
- If one has NO auth-constraint block : ALL user allowed.

Programmatic authentication

User authentication may be used in code, usgin `HttpServletRequest` methods :

- `getUserPrincipal()` : for EJBs;
- `getRemoteUser()` : User login
- `isUserInRole(String role)` : allows to restrict code fragments to specific roles.

Programmatic roles are declared in DD by security role references; references refers to real roles declared in DD under `security-role` element.

```
<security-role-ref>
  <role-name>manager</role-name>           // used as isUserInRole() parameter
  <role-link>admin</role-link>             // links to real role-name
</security-role-ref>
```

refers to :

```
<security-role>
  <role-name>admin</role-name>           // real role-name; refers to vendor
</security-role>                         // specific realm
```

NOTE : When a role-name is used in code (`isUserInRole()`) the container looks for it in the `security-role-ref` block first. If the same role-name exists in the real `security-role` block, the role-name declared in `security-role-ref` wins.

Authorization

Four types of authorization :

- BASIC;
- DIGEST;
- CLIENT-CERT;
- FORM.

Authorization method implementation.

BASIC, DIGEST, CLIENT-CERT

Just declare in the DD, under the `login-config` element, in the `auth-method` element.

FORM

Form authorization method requires three steps :

1. declare in the `login-config` element;
2. create an HTML login form;
3. create an HTML error page.

```
<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>
```

The error page may be any html/jsp page.

The login form MUST respect these rules :

- form action must be `j_security_check`;
- login input element name MUST be `j_username`;
- password input element name MUST be `j_password`;

NOTE : If FORM method is used, usage of cookies or SSL for session tracking is recommended, to keep track of session (request -> form -> login info sent).

Confidentiality

Confidentiality may be configured in DD :

```
<security-constraint>
  <transport-guarantee>...</transport-guarantee>
</security-constraint>
```

Transport guarantee is one of these :

- NONE : useless; no confidentiality;
- INTEGRAL : data MUST NOT be altered in any way;
- CONFIDENTIAL : data MUST NOT be shown to anyone.

NOTE : in general, containers use SSL for transport guarantee, which means CONFIDENTIAL and INTEGRAL have the same effect. But this is NOT guaranteed (not in spec).

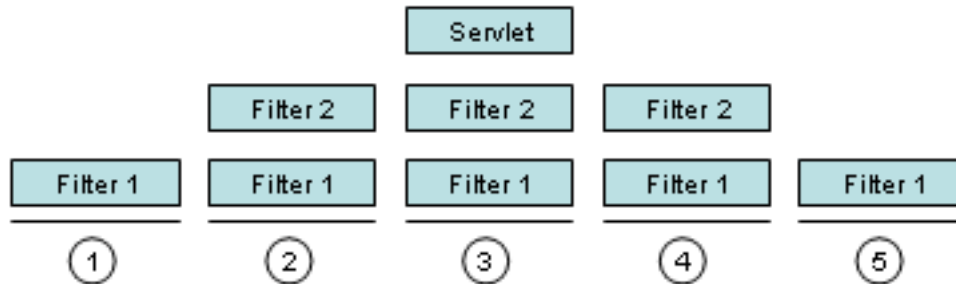
Authentication AND confidentiality

Authentication process with confidentiality :

1. request is made on HTTP : container looks for transport guarantee;
2. if restricted acces, container redirects to the same resource on **HTTPS**;
3. request sent again : containers receives request, and send back a 401 error code, meaning authentication is required : login popup appears;
4. request is sent again with **encrypted** login info on **HTTPS**.

Chapter 13 : Filters

Filters may be used to filter the request and/or the response the servlets works with.
Filters work as a stack :



Filter 1 receives request and enters its doFilter() method. Method executes until FilterChain.doFilter() is called, then hand passes to next Filter/servlet. When processing completes, hand comes back to Filter after the call to FilterChain doFilter() method.

Key points about Filters :

- Container knows Filter API;
- Container manages Filter's lifecycle;
- Filters are declared in the DD;
- Only ONE interface : javax.servlet.Filter (and FilterChain).

Lifecycle

Filter lifecycle is managed by the Container :

- Call to `init(FilterConfig)`;
- Call to `doFilter(ServletRequest, ServletResponse, FilterChain)`;
- Call to `destroy()`.

NOTE : doFilter() takes ServletRequest and ServletResponse, NOT HttpServletRequest...

FilterChain

The FilterChain object is initialized with DD information about what Filters are to be applied and in which order they must be chained. FilterChain defines one method :

```
doFilter(ServletRequest, ServletResponse)
```

NOTE : Filters do NOT know who comes next in the Stack; the FilterChain object knows this.

NOTE : a Filter is NOT required to call its FilterChain's doFilter() method.

Deployment Descriptor

Filter declaration in DD requires three things :

- Declare the Filter;
- Map the filter to a web resource;
- Arrange mappings to create invocation sequence.

Declaration

```
<filter>
  <filter-name>...</filter-name>           // Mandatory
  <filter-class>...</filter-class>        // Mandatory
  <init-param>                             // Optional : 0...*
    <param-name>...</param-name>
    <param-value>...</param-value>
  </init-param>
</filter>
```

Mapping

```
<filter-mapping>
  <filter-name>...</filter-name>           // refers to filter group name
  <url-pattern>...</url-pattern>
</filter-mapping>
```

OR

```
<filter-mapping>
  <filter-name>...</filter-name>
  <servlet-name>...</servlet-name >
</filter-mapping>
```

Ordering

1. All the filters with a matching URL pattern are placed first in the chain, in the order they are declared in the DD;
2. All the filters with a matching servlet name are placed after, in the order they are declared in the DD.

dispatcher

Filters are applied when a request comes from outside the webapp. The dispatcher element allows the Filter to be applied when a web app element calls the filtered resource *via* a RequestDispatcher.

Dispatch functionality is declared in the `filter-mapping` group with the `dispatcher` element. Zero to four `dispatcher` elements are allowed, one for each allowed value :

- `REQUEST` : default, value if no dispatcher is declared : filters client requests;
- `INCLUDE` : applies filter to webapp `include()` calls;
- `FORWARD` : applies filter to webapp `forward()` calls;
- `ERROR` : activates filter for resources called by the error handler.

Wrapper (aka *Decorator pattern*)

When a servlet's service method completes, the response is sent back to the client immediately, even if a filter still has to work. It means a filter cannot modify the generated response object. The solution is to pass a fake response object to the servlet : a **wrapper**.

Four types of wrappers :

- `ServletRequestWrapper` and `HttpServletRequestWrapper`;
- `ServletResponseWrapper` and `HttpServletResponseWrapper`.

Wrappers :

- Delegates the calls to wrapped object methods to the wrapped object itself;
- Calls to overridden methods are handled by the wrapper;
- Adds capabilities to the wrapped object; these new capabilities are handled by the Wrapper.

Chapter 14 : Design Patterns

Business Delegate

Used to shield the web tier controllers from the fact that some model components are remote.

Features

- Acts as a proxy, implementing remote service's interface;
- Initiates communications with remote service;
- Handles communication details and exceptions;
- Receives request 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 details of remote component lookup and communications, makes controller more cohesive.

Principles

- Business Delegate is based on :
 - Hiding complexity;
 - Coding to interfaces;
 - Loose coupling;
 - Separation of concerns.
- Minimizes changes on web tier when changes occur on business tier;
- Reduces coupling between tiers;
- Adds a layer to the app, which increases complexity;
- Calls to Business Delegate should have thick granularity, to reduce network traffic.

Service Locator

Used to perform registry lookups to simplify the other components that would need lookups (like Business Delegate).

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, like JNDI, UDDI, RMI...

Principles

- Service Locator is based on :
 - Hiding complexity;
 - Separation of concerns.
- Minimizes impact on the web tier when remote components change locations or containers;
- Reduces coupling between tiers.

Transfer Object

Used to minimize network traffic by providing a local representation of fine-grained component (entity).

Functions

- Provides a local representation of a remote entity,
- Minimizes network traffic;
- Can follow JavaBean conventions so that it can move across the network;
- Implemented as a Serializable object so that it can move across the network;
- Typically easily accessible by view components.

Principles

- Transfer Object is based on :
 - Reducing network traffic;
- Minimizes impact on web tier when remote components' data is accessed with fine-grained calls;
- Minimizes coupling between tiers;
- A drawback is that components accessing the Transfer Object can receive out-of-date data, because the Transfer Object is a snapshot of remote data;
- Making updatable Transfer Objects concurrency-safe is complex.

Intercepting Filter

Used to modify requests received by servlets or to modify responses being sent to users.

Functions

- 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 DD;
- Filters are modular so that they can be executed in chains;
- Filters have lifecycle managed by the Container;
- Filters must implement Container callback methods.

Principles

- 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

Used to create a logical structure that separates the code into three basic types of components (Model, View and Controller). Increases cohesiveness and allows greater reusability.

Features

- Views can change independently from controllers and models;
- Model components hide internal details (data structures) from views and controllers;
- If model adheres to a strict contract, then they can be reused in other applications as GUIs or J2ME;
- Separation of model and controller code allows for easier migration to using remote business components.

Principles

- MVC is based on :
 - Separation of concerns;
 - Loose coupling.
- Increases cohesion of individual components;
- Increases the overall complexity of the application;
- Minimizes the impact of changes in other tiers of the application.

Front controller

Used to gather common, redundant request processing code into a single component. Allows application to be more cohesive and less complex.

Features

- Centralizes a webapp initial request handling task to a single component;
- Using 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. Creating a webapp based on Front Controller would end up in rewriting most of Struts features.

Principles

- 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.