Jsp, Servlet Part - I

Server:

Server is a computer machine with a server software installed in it. The basic purpose of such software is to provide resource or services. Such server system is capable of serving several clients at a time.

Ex.: Mail Server, FTP Server, Http Server are few such server software which serve different services and resources.

Servers are of Two types:

Static Server: is a server software which serves static resource. You just need to request a resource of static nature: by specifying File-Name and Location. If such file exists then the server simply sends the file as a response.

Such server is also stateless, which means they forget the client as soon as it sends back the requested resource.

In short you can say such servers are: **static** and **stateless**.

Dynamic Server: is a server software which generates dynamic response. In other words the response is created when the request is received. The response generated are not only just-in-time but they are also customized.

Such server is also state full, because they can remember the clients and their activities.

Here we will focus on the Http Server.

Client: is a software installed on a computer to make end user request for resource and services.

Ex.: Any Browser: IE, Mozilla and Chrome etc..

Network: includes connection amongst different devices with capability to connect and make requests and responses. The network is comprise of wired connections, wireless connections, satellites, network-towers radars and different types of network devices and they connect client and server machines.

Http Request: Client machines makes requests for resource and services and these requests are not just plain requests, even better they are http-requests. Browser formats user request and forward it to the server.

When a client makes a request then the client software takes care of converting it into an http-request.

Http Response: Server receives a request and responds back with an appropriate response. A server is also abide by the rules of http and formats a response accordingly, thus it is not a plain response, instead an http response.

There are certain protocols which is followed when making requests and responses. The protocols are followed by both clients and servers. These protocols are known as HTTP: Hyper Text Transfer Protocol.

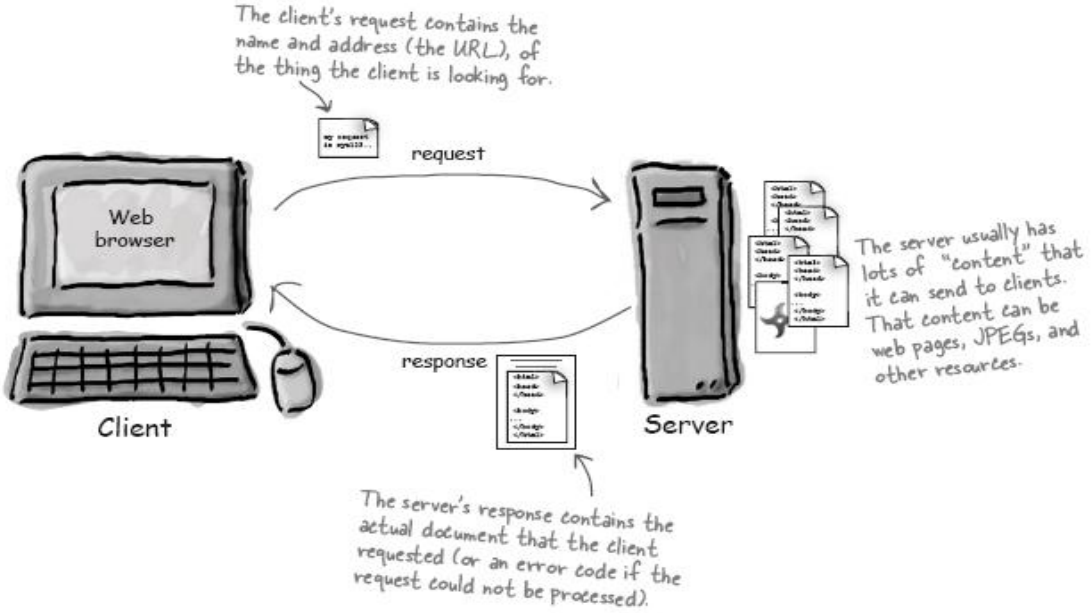
Resource & Services: A client either requests for some resource, which can be a file with any extension or can request for a service.

On the other hand the server job is to provide the requested resource and services accordingly.

Some services: A Mail Servers provide mail services like sending and receiving mails, A Chat Server allows to join a chat room and chat with others. Here our focus is to learn http servers and services provided by them are online banking service, social and professional networking etc.

How many ways a client can make a request:

1. User can type a URL in the address-bar.
2. User can click over a link.
3. User can submit a form.



Clients and servers know HTML and HTTP

HTML	**HTTP**
When a server answers a request, it often send the browser a set of instructions written in HTML, the Hyper Text Markup Language. The HTML tells the browser how to present the content to the user. **(HTML stands for Hyper Text Markup Language.)**	Most of the conversations held on the web between clients and servers are held using the HTTP protocol, which allows for simple request and response conversations. The client sends an HTTP request, and the server answers with an HTTP response. **(**HTTP stands for **Hyper Text Transfer Protocol.)**

In order to communicate, Client and Server must share a common language. On the web, clients and servers must speak HTTP, and browsers must know HTML.

What is the HTTP protocol?

HTTP runs on top of TCP/IP.

TCP is responsible for making sure that a file sent from one network node to another ends up as a complete file at the destination, even though the file is split into chunks when it's sent.

IP is the underlying protocol that moves/routes the chunks (packets) from one host to another on their way to the destination.

HTTP, then, is another network protocol that has Web-specific features, but it depends on TCP/IP to get the complete request and response from one place to another.

The structure of an HTTP conversation is a simple Request/ Response sequence; a browser requests, and a server responds.

Key elements of a request stream:	Key elements of a response stream:
1. Http Method (The action to be performed). 2. The page to access (a URL).	1. A status code (for whether the request was successful).

3. Form Parameters (Like arguments to a method).	2. Content-type (text, picture, html etc.) 3. The content (the actual html, image etc.)

Requests can be made either by:	**Browser takes care of user requests:**
1. Submitting a form. 2. Clicking over a link. 3. Typing and submitting a URL in the address-bar.	The browser formats the request before it forward it to the server. A method is associated with any request. By default requests are sent using GET method. A form can be submitted either by GET or POST method.

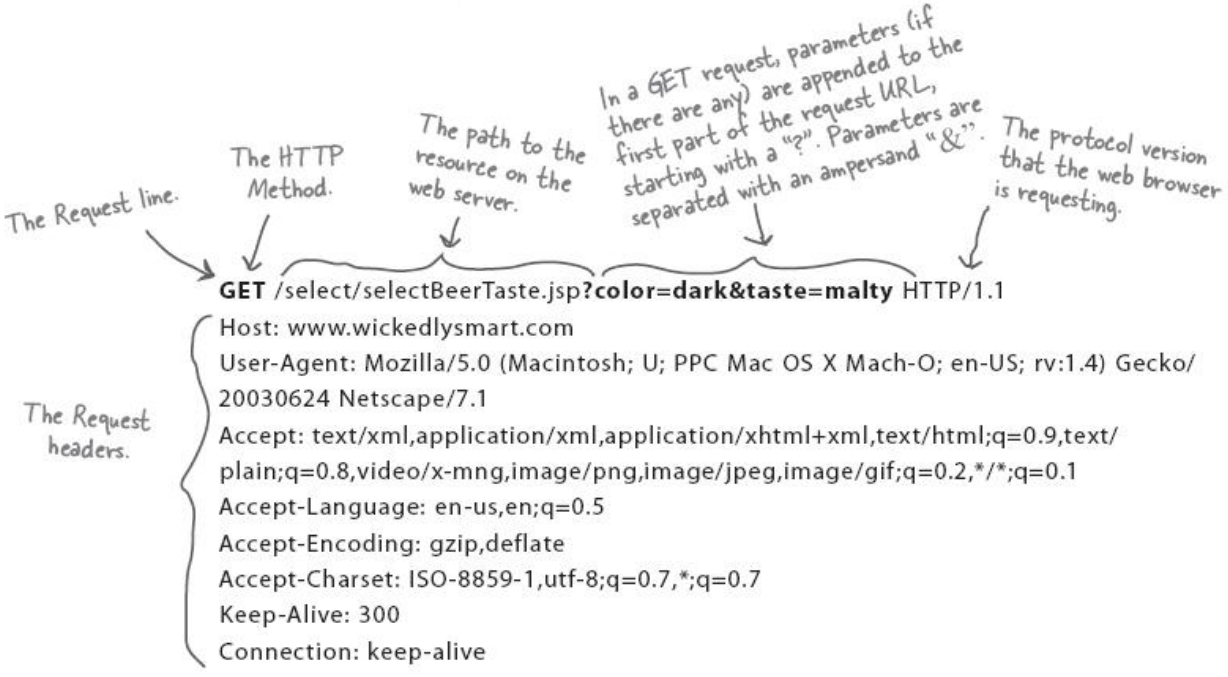
GET vs. POST:

Using GET you can send limited information. POST does not limit the information you can send.
The data we send using GET is appended to the URL, up in the browser bar. This way whatever we send is exposed.
If we POST, we can't bookmark the address URL.

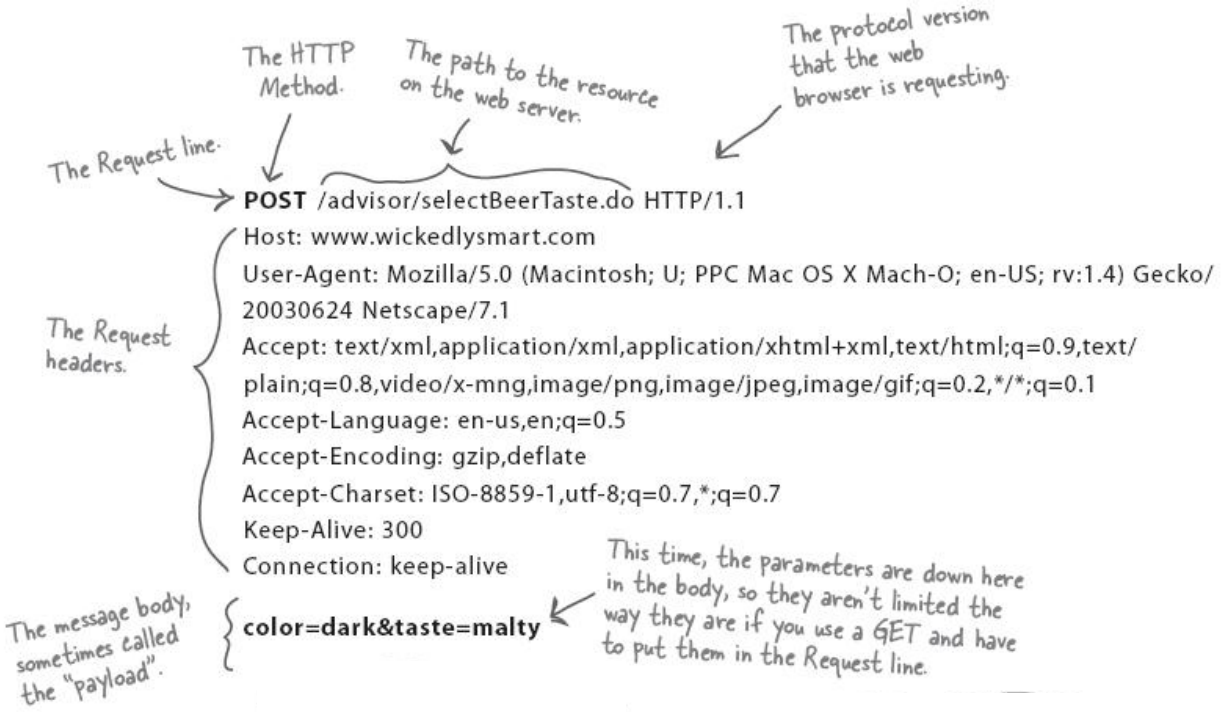
GET method says that the request is not intended to make changes at the server end.
POST method says that the request is intended to make changes at the server end.
The methods does not enforce, they just indicate the intention of the request.

Http GET request:

The path to the resource, and any parameters added to the URL are all included on the "request line".



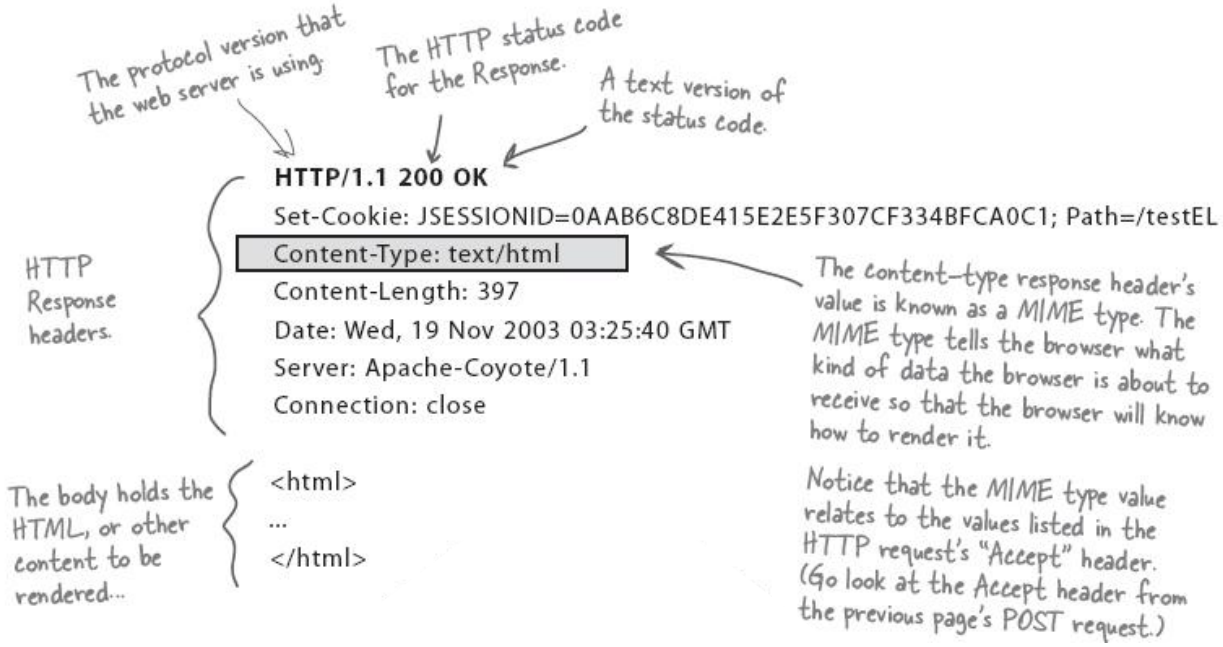
Http POST request:



The data to be sent back to the server is known as the "message body" or "payload" and can be quite large.

Http response:

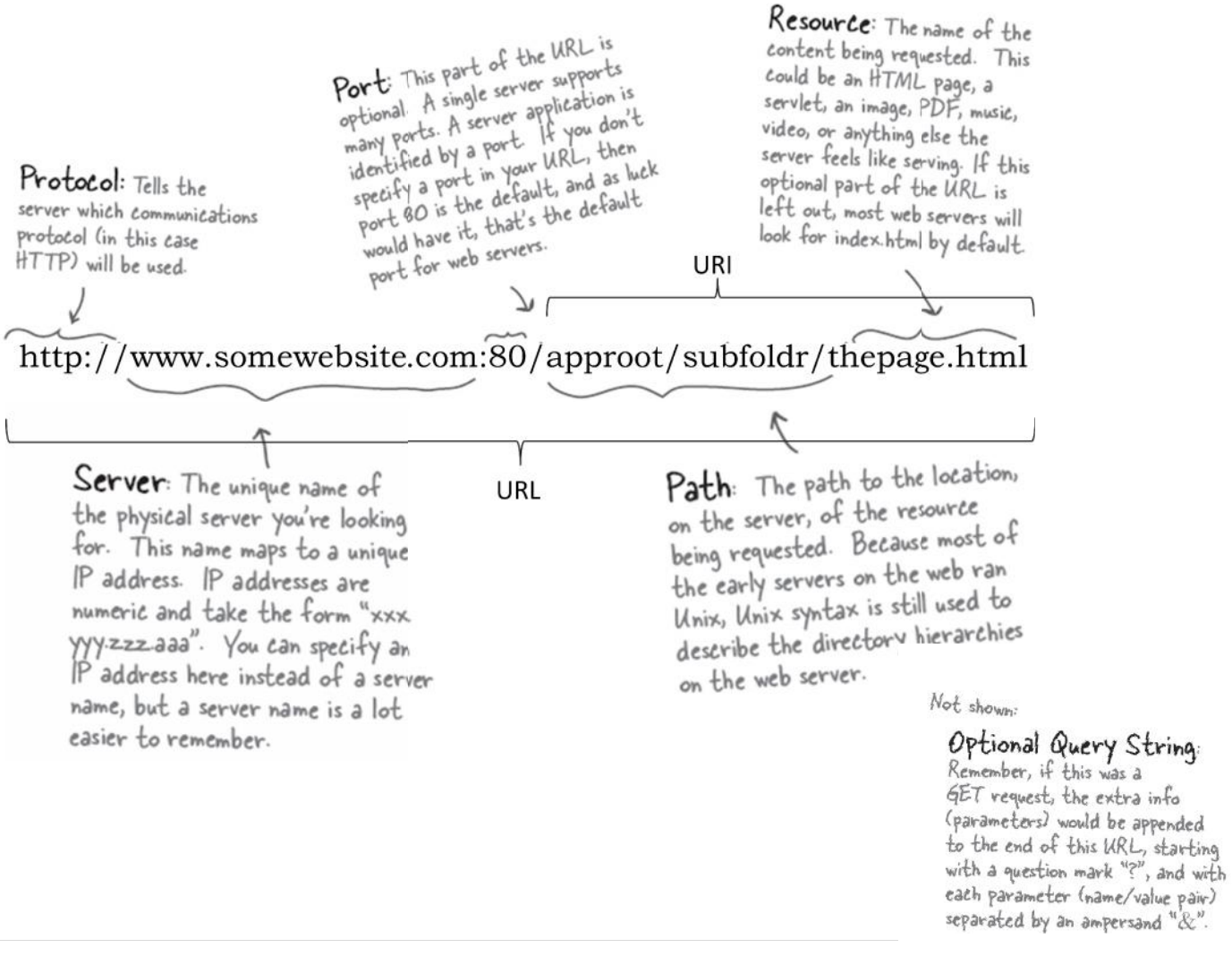
An HTTP response has both a header and a body.
The header info tells the browser about:
1. The protocol being used,
2. Whether the request was successful,
3. And what kind of content is included in the body.



1. The HttpServlet's doGet() and doPost() methods take an HttpServletRequest and an HttpServletResponse. The service() method determines whether doGet() or doPost() runs based on the HTTP Method (GET, POST, etc.) of the HTTP request.
2. POST requests have a body; GET requests do not, although GET requests can have request parameters appended to the request URL (sometimes called "the query string").
3. GET requests are inherently (according to the HTTP spec) idempotent. They should be able to run multiple times without causing any side effects on the server. GET requests shouldn't change anything on the server. But you could write a bad, non-idempotent doGet() method.
4. POST is inherently not idempotent, so it's up to you to design and code your app in such a way that if the client sends a request twice by mistake, you can handle it.
5. If an HTML form does not explicitly say "method=POST", the request is sent as a GET, not a POST. If you do not have a doGet() in your servlet, the request will fail.
6. You can get parameters from the request with the getParameter("paramname") method. The return value is always a String.
7. If you have multiple parameter values for a given param- eter name, use the getParameterValues("paramname") method that returns a String array.
8. You can get other things from the request object including headers, cookies, a session, the query string, and an input stream.

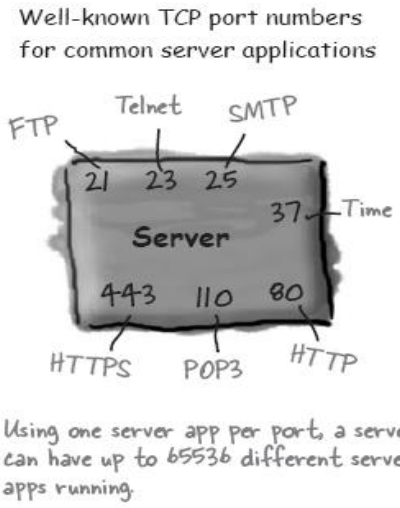
URL:

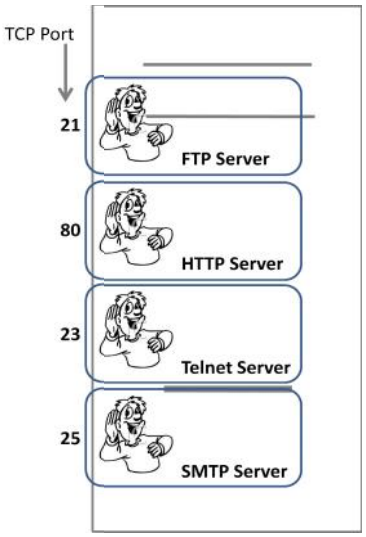
The URL: Uniform Resource Locators. Every resource on the web has its own unique address, in the URL format.



TCP port:

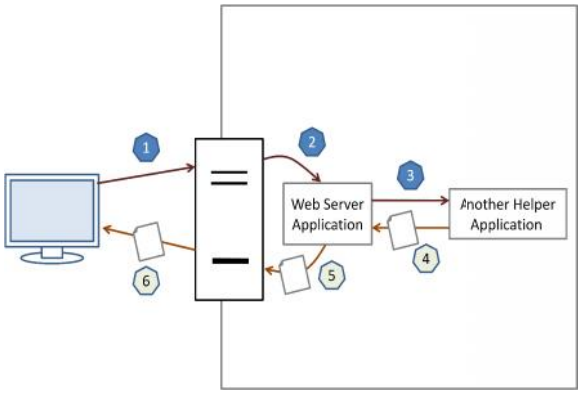
A 16-bit number that identifies a specific software program on the server hardware.



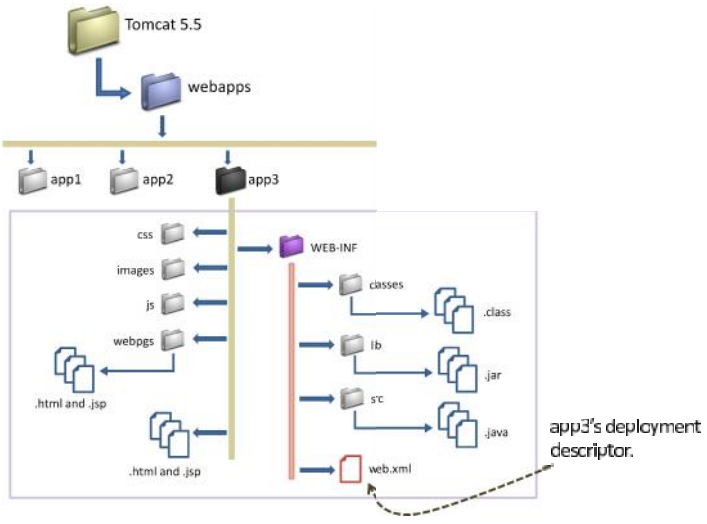


Think of ports as unique identifiers. A port represents a logical connection to a particular piece of software running on the server hardware. You can't see backside of your hardware box and find a TCP port. You have 65536 of them on a server (0 to 65535). They do not represent a place to plug in physical devices. They're just numbers representing a server application. Without port numbers, the server would have no way of knowing which Application a client wanted to connect to. What if your web browser, for example, landed at the POP3 mail server instead of the HTTP server?

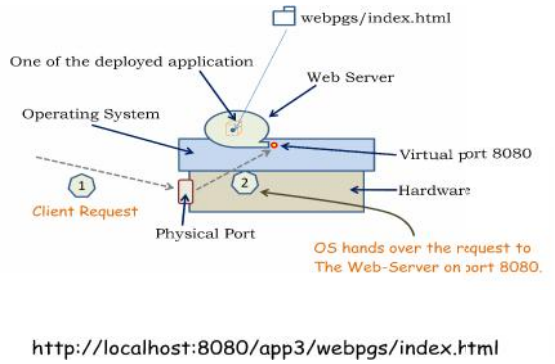
Request For Dynamic Server:



Directory Structure of Tomcat Website:

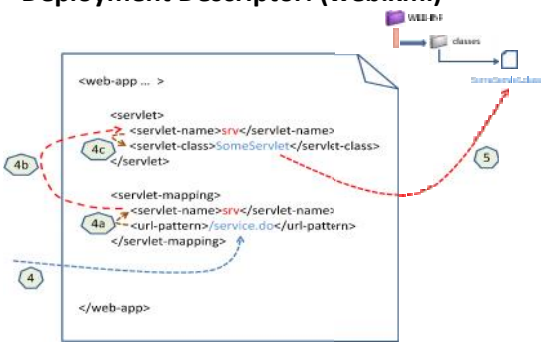


Server-machine receives requests and hands it over to the server-software:



http://localhost:8080/app3/webpgs/index.html

Deployment Descriptor: (web.xml)



Deployment Descriptor(DD => web.xml):

1. One DD per web-application.
2. A DD can declare many servlets.
3. A <servlet-name> ties the <servlet> element to the <servlet-mapping> element.
4. A <servlet-class> is the java class.
5. A <url-pattern> is the name the client uses for the request.

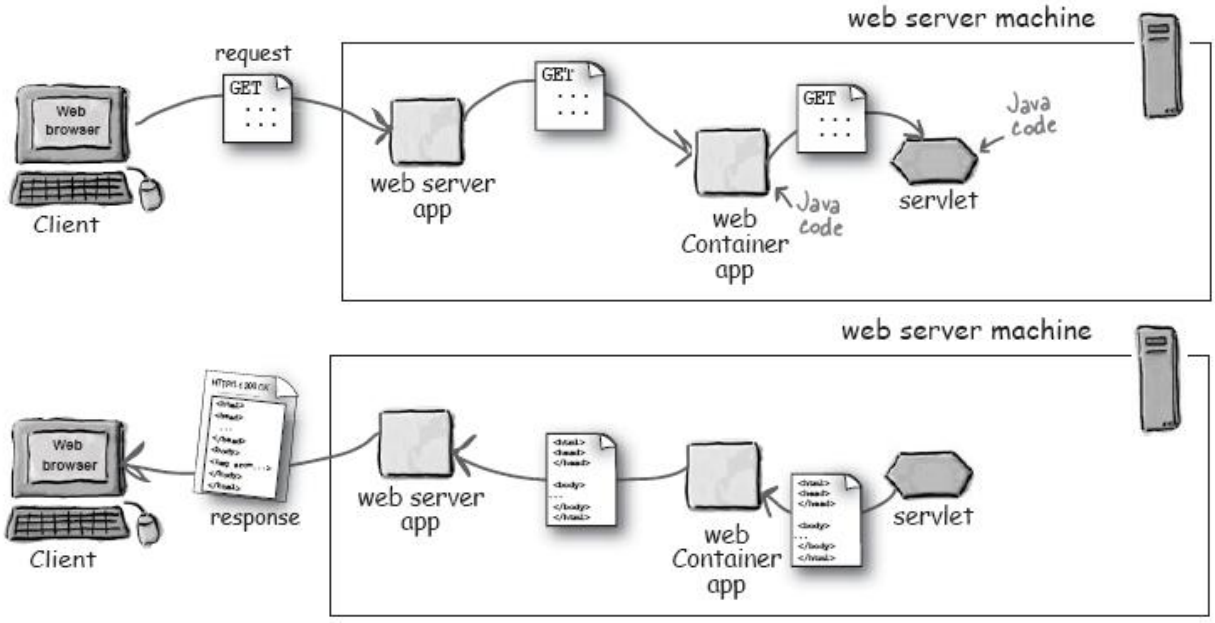
Servlet (write, deploy and run):

1. Write a servlet class:
   a. Servlet class must extend HttpServlet class.
   b. Must be declared public.
   c. Servlet class must either override doPost or doGet or both.
      (It depends on the Http-Request-Method.)
   d. specify required imports & declare the package if any.
   c. Write code in the overriding method to handle the request.
2. Build the directory tree under the existing tomcat directory.
3. Now compile servlet.
4. Copy the generated class file to WEB-INF/classes and copy the the web.xml file to WEB-INF.
5. From Tomcat directory start the Tomcat… C:/Tomcat 5.5/bin/tomcat5.exe
6. Launch the browser and type in: http://localhost:8080/app3/service.do

What is a Container:

Servlet's don't have a main() method. They are under the control of another application called a container. Tomcat is an example of a container. When our web server application(like Apache) gets a request for a Servlet *(as opposed to, a plain old static HTML page),* the server hands the request not to the servlet itself, but to the Container in which the servlet is deployed.

It's the Container that gives the servlet the HTTP request and response, and it's the Container that calls the servlet's methods *(like doPost() or doGet()).*



What does the container gives us?

Why can't we handle the request without servlets and container support, using plain Java? Because container provides:

1. Communication Support

The container provides an easy way for our servlets to talk to our web server. We don't have to build a ServerSocket, listen on a port, create streams, etc. The Container knows the protocol between the web server and itself, so that our servlet doesn't have to worry about an API between, say, the Apache web server and our own web application code. All we have to worry about is our own business logic that goes in our Servlet (like accepting an order from our online store).

2. Lifecycle Management

The Container controls the life and death of our servlets. It takes care of loading the classes, instantiating and initializing the servlets, invoking the servlet methods, and making servlet Instances eligible for garbage collection. With the Container in control, we don't have to worry as much about resource management.

3. Multithreading Support

The Container automatically creates a new Java thread for every servlet request it receives. When the servlet's done running the HTTP service method for that client's request, the thread completes (means it dies).
But having the server create and manage threads for multiple requests still saves us a lot of work.

4. Declarative Security

With a Container, we get to use an XML deployment descriptor to configure (and modify) security without having to hard-code it into our servlet (or any other) class code. Think about that! We can manage and change our security without touching and recompiling our Java source files.

5. JSP Support

We know how cool JSPs are. Who do you think takes care of translating that JSP code into real Java? Of course. The *Container.*

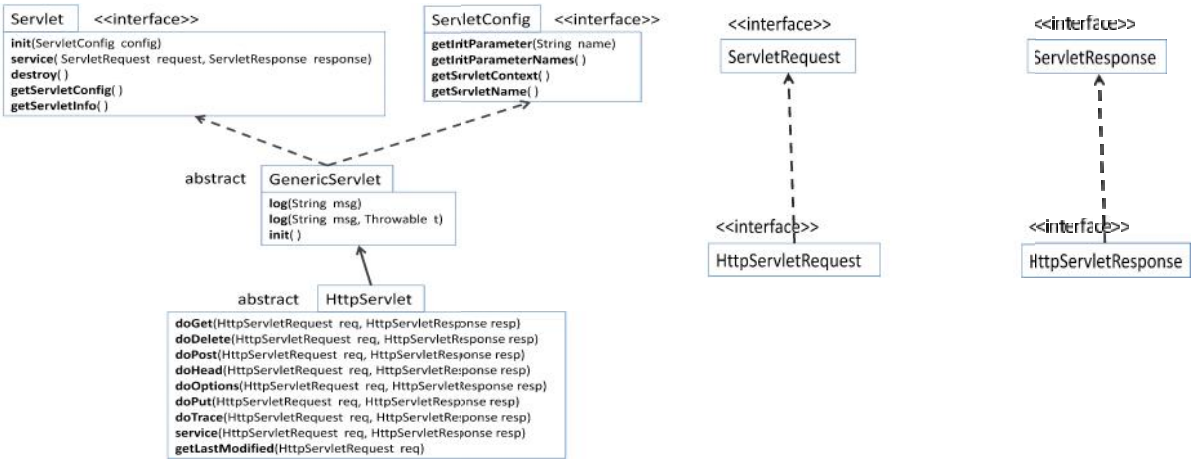
But there is more to a servlet's life:

There are several questions related to the servlet's life:

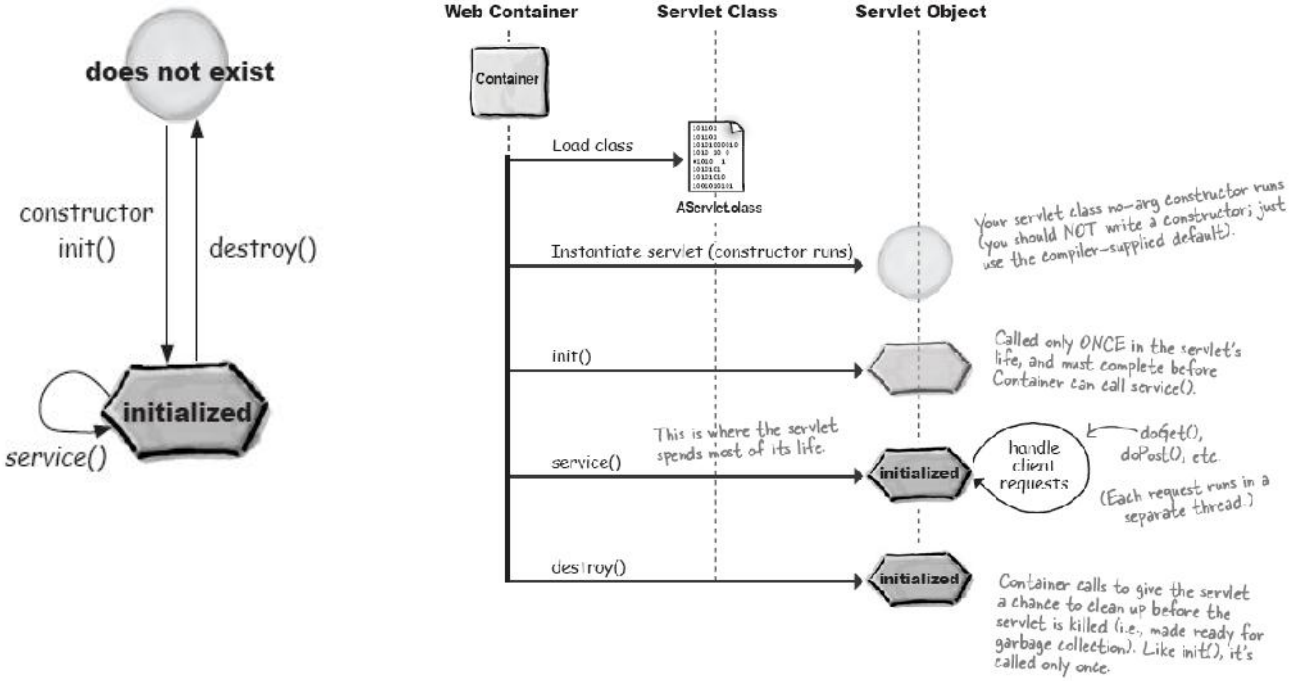
Q. When was the servlet class loaded?
Q. When did the servlet's constructor run?
Q. How long does the servlet object live?
Q. When should your servlet initialize resources?
Q. When should it clean up its resources?

Loading & Initializing:

The servlet starts life when the Container finds the servlet class file. This virtually always happens when the Container starts up (for example, when you run Tomcat). When the Container starts, it looks for deployed web apps and then starts searching for servlet class files. Finding the class is the first step. Loading the class is the second step, and it happens either on Container startup or first client use. Your Container might give you a choice about class loading, or it might load the class whenever it wants.

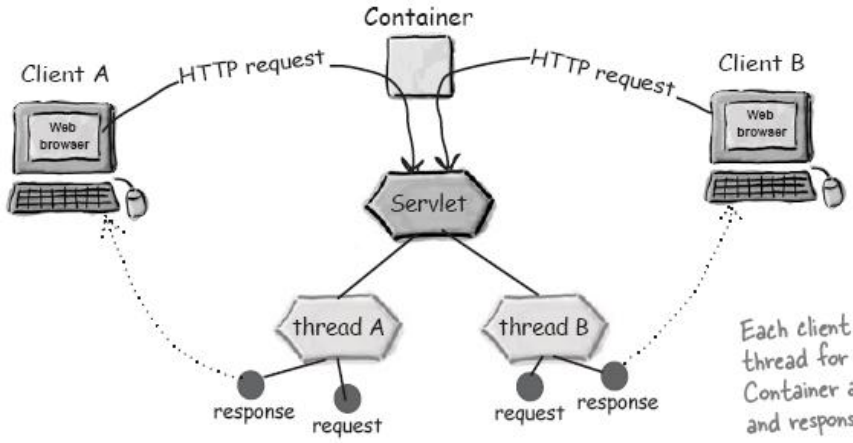


The servlet lifecycle is simple; there's only one main state—initialized.
If the servlet isn't initialized, then it's either being initialized (running its constructor or init()method), being destroyed (running its destroy() method), or it simply does not exist.



Each request runs in a separate thread!

There aren't multiple instances of any servlet class, except in one special case (called SingleThreadModel, which is inherently evil).The Container runs multiple threads to process multiple requests to a single servlet. And every client request generates a new pair of request and response objects.



In this diagram one thread per request, not one thread per client.

Deployment Descriptor:
Deployment Descriptor is a simple XML Document to tell the Container how to run our servlets and JSPs.
We'll use two XML elements to map URLs to servlets:

The two DD elements for URL mapping:
1. <servlet>
maps internal name to fully-qualified class name
2. <servlet-mapping>
maps internal name to public URL name

The common questions:
Do we really want the client to know exactly how things are structured on our server?

From the security point of view: our Servlet location is not exposed to the outside world. This is our server which locates the file according the configuration given in DD. Tomcat does not allow to navigate the content kept in WEB-INF by typing a URL in the address bar.
Why don't we all just use the one, real, non-confusing file name?
What if we hardcode the path or the file name into JSPs and other HTML pages.

Moving things around: DD gives us flexibility to move files. No need to track each link and make changes in the code.

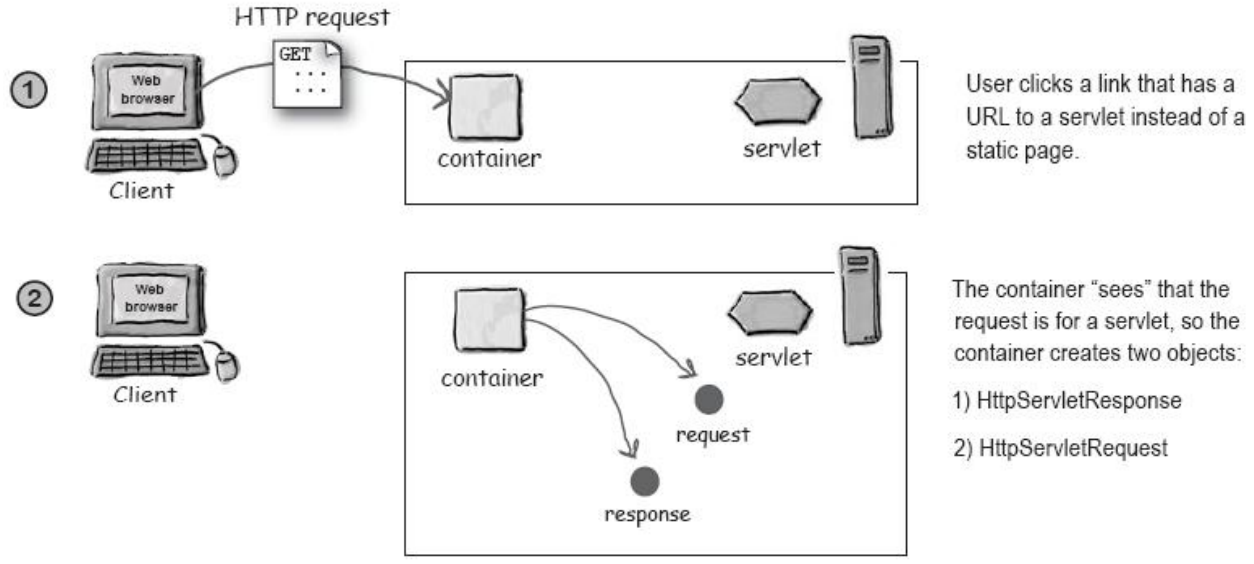
Besides mapping URLs to actual servlets, we can use the DD to customize other aspects of our web application including:

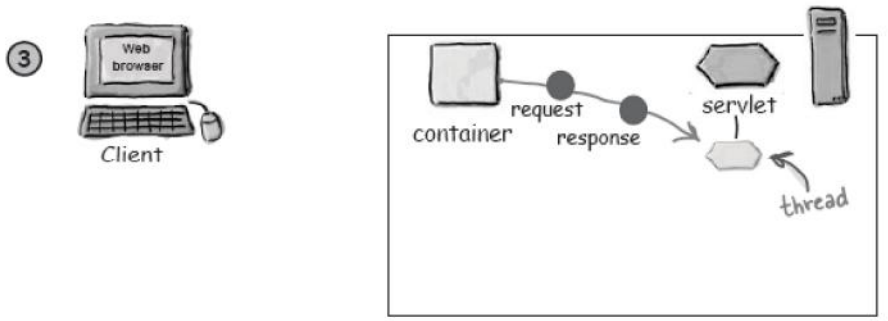
- Security roles
- Error pages
- Tag libraries
- Initial configuration information
- We can even declare that we'll be accessing specific enterprise javabeans. (if it's a full J2EE server)

DD gives us a way to **declaratively** modify our application without changing source code!
Those who aren't Java programmers can customize our Java web application.

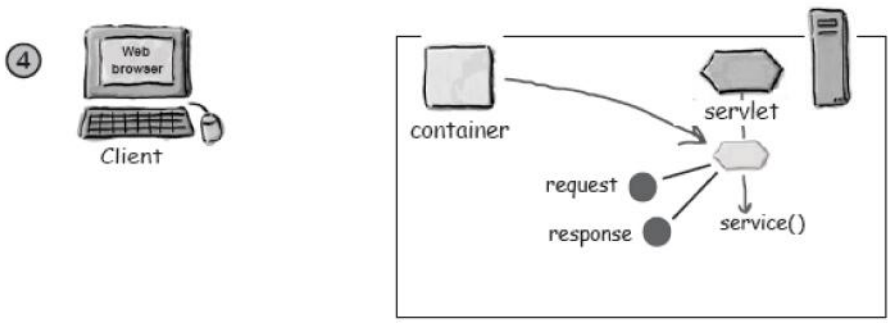
The service() method is always called in its own stack…

How the Container Handles a Request



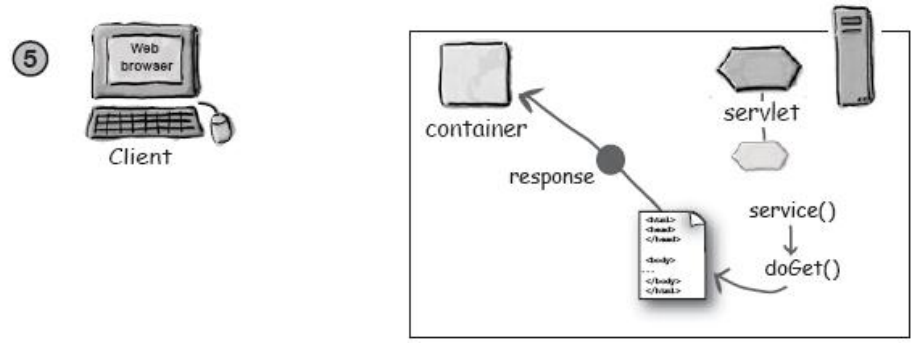


The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

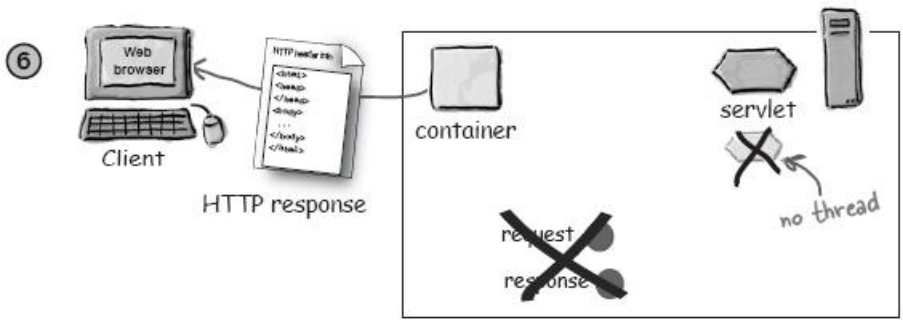


The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.



The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!



The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

The Three Lifecycle Moments:

1

init()

When it's called

The Container calls init() on the servlet instance *after* the servlet instance is created but *before* the servlet can service any client requests.

What it's for

Gives you a chance to initialize your servlet before handling any client requests.

Do you override it?

Possibly.

If you have initialization code (like getting a database connection or registering yourself with other objects), then you'll override the init() method in your servlet class.

2

service()

When it's called

When the first client request comes in, the Container starts a new thread or allocates a thread from the pool, and causes the servlet's service() method to be invoked.

What it's for

This method looks at the request, determines the HTTP method (GET, POST, etc.) and invokes the matching doGet(), doPost(), etc. on the servlet.

Do you override it?

No. Very unlikely.

You should NOT override the service() method. Your job is to override the doGet() and/or doPost() methods and let the service() implementation from HTTPServlet worry about calling the right one.

3

doGet()

and/or

doPost()

When it's called

The service() method invokes doGet() or doPost() based on the HTTP method (GET, POST, etc.) from the request.

(We're including only doGet() and doPost() here, because those two are probably the only ones you'll ever use.)

What it's for

This is where *your* code begins! This is the method that's responsible for whatever the heck your web app is supposed to be DOING.

You can call other methods on other objects, of course, but it all starts from here.

Do you override it?

ALWAYS at least ONE of them! (doGet() or doPost())

Whichever one(s) you override tells the Container what you support. If you don't override doPost(), for example, then you're telling the Container that this servlet does not support HTTP POST requests.

Servlet initialization

Thread A

The Container calls init() on the servlet instance *after* the servlet instance is created but *before* the servlet can service any client requests.

If you have initialization code (like getting a database connection or registering yourself with other objects), then you'll override the init() method in your servlet class. Otherwise, the init() method from GenericServlet runs.

Client request 1

Thread B

When the first client request comes in, the Container starts (or finds) a thread and causes the servlet's service() method to be invoked.

You normally will NOT override the service() method, so the one from HttpServlet will run. The service() method figures out which HTTP method (GET, POST, etc.) is in the request, and invokes the matching doGet() or doPost() method. The doGet() and doPost() inside HttpServlet don't do anything, so you have to override one or both. This thread dies (or is put back in a Container-managed pool) when service() completes.

Client request 2

Thread C

When the second (and all other) client requests come in, the Container again creates or finds a another thread and causes the servlet's service() method to be invoked.

So, the service() --> doGet() method sequence happens each time there's a client request. At any given time, you'll have at least as many runnable threads as there are client requests, limited by the resources or policies/configuration of the Container. (You might, for example, have a Container that lets you specify the maximum number of simultaneous threads, and when the number of client requests exceeds that, some clients will just have to wait.)

Methods we can invoke on a request object:

getRemotePort():
Here the client is remote to the srever, so getRemotePort() means “get the client’s port”. In other words, the port number on the client from which the request was sent. So if you are a servlet, **remote** means **client**.

getServerPort():
Server port means the port to which the request originally sent.

getLocalPort():
Local port means the port where the request finally end up.
- The request are sent to a single port where the server is listening.
- The server turns around and finds a different local port for each thread so that the app can handle multiple clients at the same time.

getHeader( ):
Headers have both name and value:
User-Agent : Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624 Netscape/7.1
Host : www.wickedlysmart.com
The values come back from headers are always in a String form. But few of them represent a number.
Content-Length : 23
*The number of bytes that make up the message-body.
Max-Forwards : 2
*returns an integer indicating how many router hops the request is allowed to make. (Useful if you’re trying to trace a request that you think is getting stuck in a loop somewhere.)

getIntHeader( ):
You could get the value of the “Max-Forwards” header by using getHeader():

```
String forwards = request.getHeader(“Max-Forwards”);
int forwardsNum = Integer.parseInt(forwards);
```

getIntHeader() saves the extra step of parsing the String to an int:

```
int forwardsNum = request.getIntHeader(“Max-Forwards”);
```

Some other important methods:
getCookies(), getSession(), getMethod(), getInputStream(), getParameter(String), getParameterNames(), getParameterValues(String) etc.

HttpServletRequest & HttpServletResponse, both are interfaces.
Who implements them?
Answer: the Container. The implementation is left to the Container Vendor. The implementing class is not part of the API.
All we need to know are methods those we can call on the objects the Container gives us as part of the request.
We refer these objects by the interface types.
The generic or non-http classes or interfaces are designed such that the people who want to:
- use it without http protocol or
- use it with other protocol like smtp or
- use their own proprietary protocol.

Init Parameters

Imagine there is a situation where one has some information like contact email to show to the end user. But we understand the email may change time to time.
To make changes, we need to understand that the email is hard coded in the servlet. This requires you to make necessary changes in the servlet and compile the servlet.

There is a technique which allows to put such information in DD.
In the DD (web.xml) file:

```
<servlet>
    <servlet-name>Contact</servlet-name>
   <servlet-class>ContactServlet</servlet-class>
   <init-param>
     <param-name>managerEmail</param-name>
    <param-value>info@isrdc.com</param-value>
  </init-param>
</servlet>
```

No need to stop the container, no need to re-compile the classes. We just need to configure such information in the deployment descriptor(web.xml). Now our response will get the updated information.
Steps to configure the initial parameter:

1. Find the Servlet which has the hard coded value.
2. Find the Servlets entry(<servlet>) in the web.xml.
3. Now insert a tag <init-param> inside the <servlet> tag.
4. Create two sub elements in the <init-param> tag, naming <param-name> and <param-value>.
5. Insert an appropriate **name** in the <param-name>'s opening and closing tag.
6. Insert an appropriate **value** in the <param-value>'s opening and closing tag.
   Note: Close the newly introduced tags.

Access initial parameters in servlet:

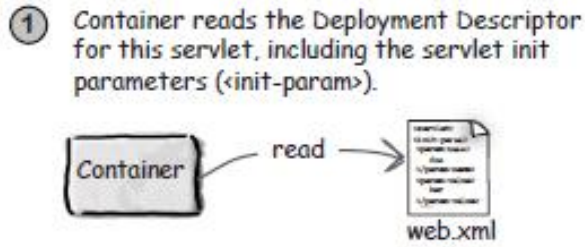
```
ServletConfig conf = getServletConfig();
String managerEmail = conf.getInitParameter("managerEmail");
```

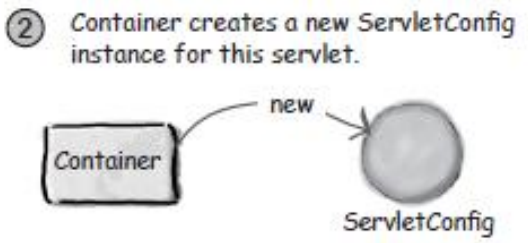
Steps to access initial parameters configured in web.xml:

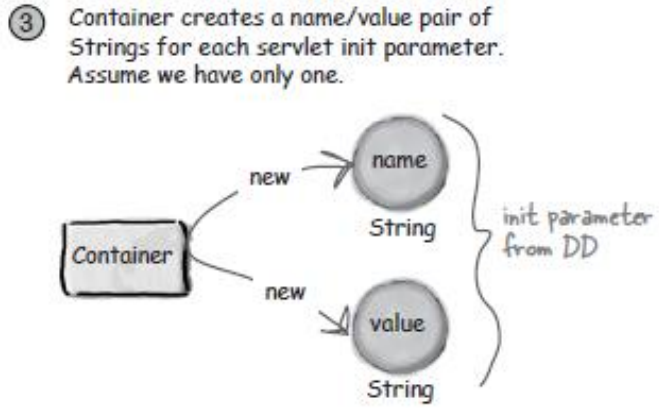
1. Every servlet class inherits getServletConfig() method. The method returns an instance of type ServletConfig(interface).
2. We can call getInitParameter() on the instance of type ServletConfig.
3. The method needs a String argument, which is the name supplied to the tag <param-name>.
4. The method returns a String value, which is the value supplied to the tag <param-value>.

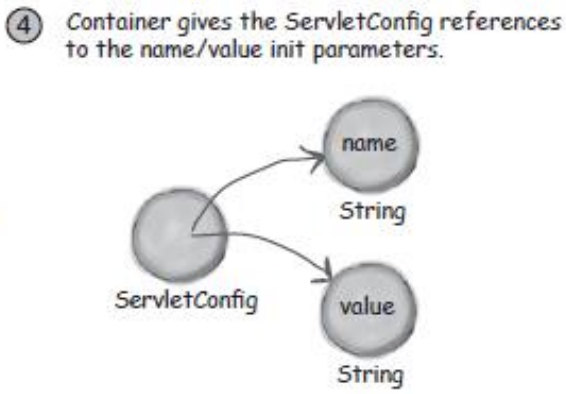
Remember that the configured value is for the particular servlet. Only the servlet can access the value. We can insert more other initial parameters for a servlet.

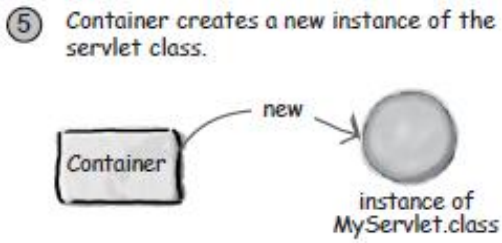
ServletConfig:
When the Container initializes a servlet, it makes a unique ServletConfig for the servlet.
When the Container makes a servlet, it reads the servlet init parameters from the DD and creates the name/value pairs DD and gives them to the ServletConfig, then passes the ServletConfig to the servlet's init() method.
Once the parameters are in the ServletConfig, they won't be read again until/unless you redeploy the servlet.

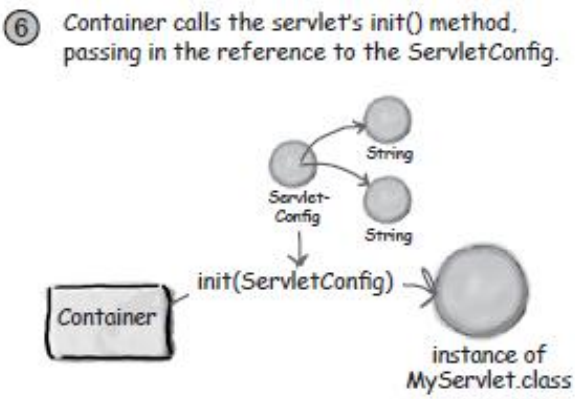












A ServletConfig Object:

- One ServletConfig object per servlet.
- Use it to pass deploy-time information to the servlet (a database lookup name, for example) that you don't want to hard-code into the servlet (servlet init parameters).
- Use it to access the ServletContext.
- Parameters are configured in the Deployment Descriptor.

Context Parameter:

Context init parameters work just like servlet init parameters, except context parameters are available to the entire web application, not just a single servlet. So that means any servlet and JSP in the app automatically has access to the context init parameters, so we don't have to worry about configuring the DD for every servlet, and when the value changes, you only have to change it at one place!

In the DD (web.xml) file:

```
<context-param>
   <param-name>adminEmail</param-name>
   <param-value>clientheaderror@wickedlysmart.com</param-value>
</context-param>
```

Access context parameters in servlet:

```
String  ctxtParmVal  =  getServletContext().getInitParameter("adminEmail"));
```

A ServletContext object :

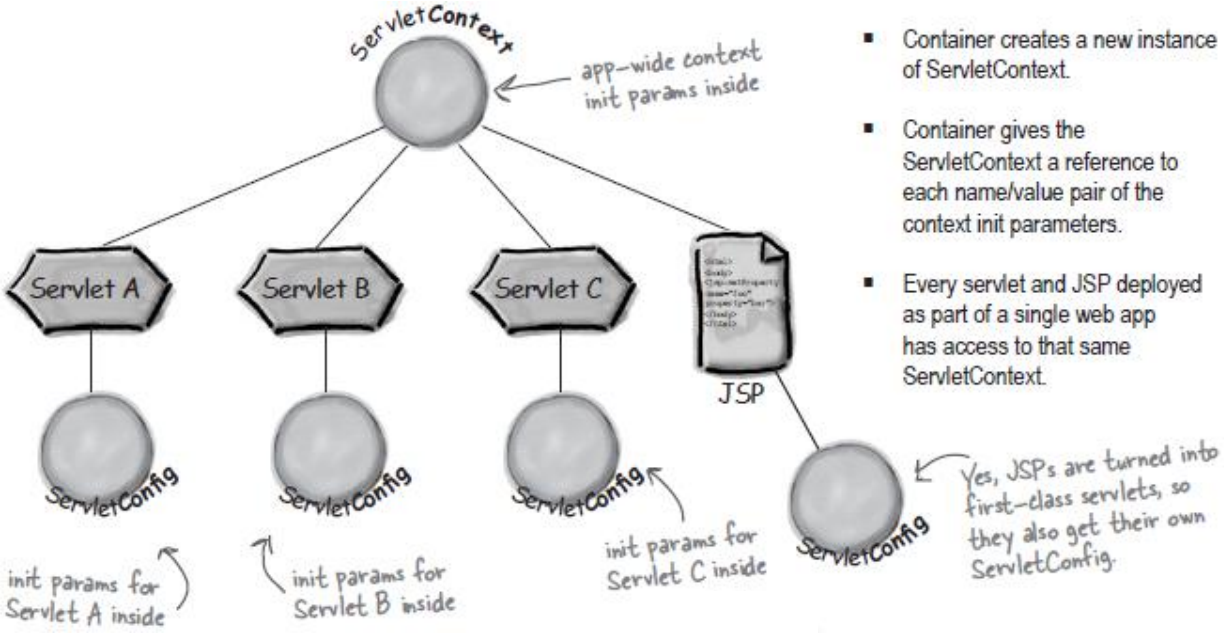
- One ServletContext per web app.
- Use it to access web app parameters (also configured in the DD.
- Use it as a kind of application bulletin-board, where you can put up messages (called attributes) that other parts of the application can access.
- Use it to get server info, including the name and version of the Container, and the version of the API that's supported.

ServletContext should have been named AppContext, because there's only one per web app, NOT one per servlet.

Web app initialization:

1. Container reads the DD and creates a name/value String pair for each <context-param>.
2. Container creates a new instance of ServletContext.
3. Container gives the ServletContext a reference to each name/value pair of the context init parameters.
4. Every servlet and JSP deployed as part of a single web app has access to that same ServletContext.

There's only one ServletContext for an entire web app, and all the parts of the web app share it.
But each servlet in the app has its own ServletConfig.
The Container makes a ServletContext when a web app is deployed, and makes the context available to each Servlet and JSP (which becomes a servlet) in the web app.



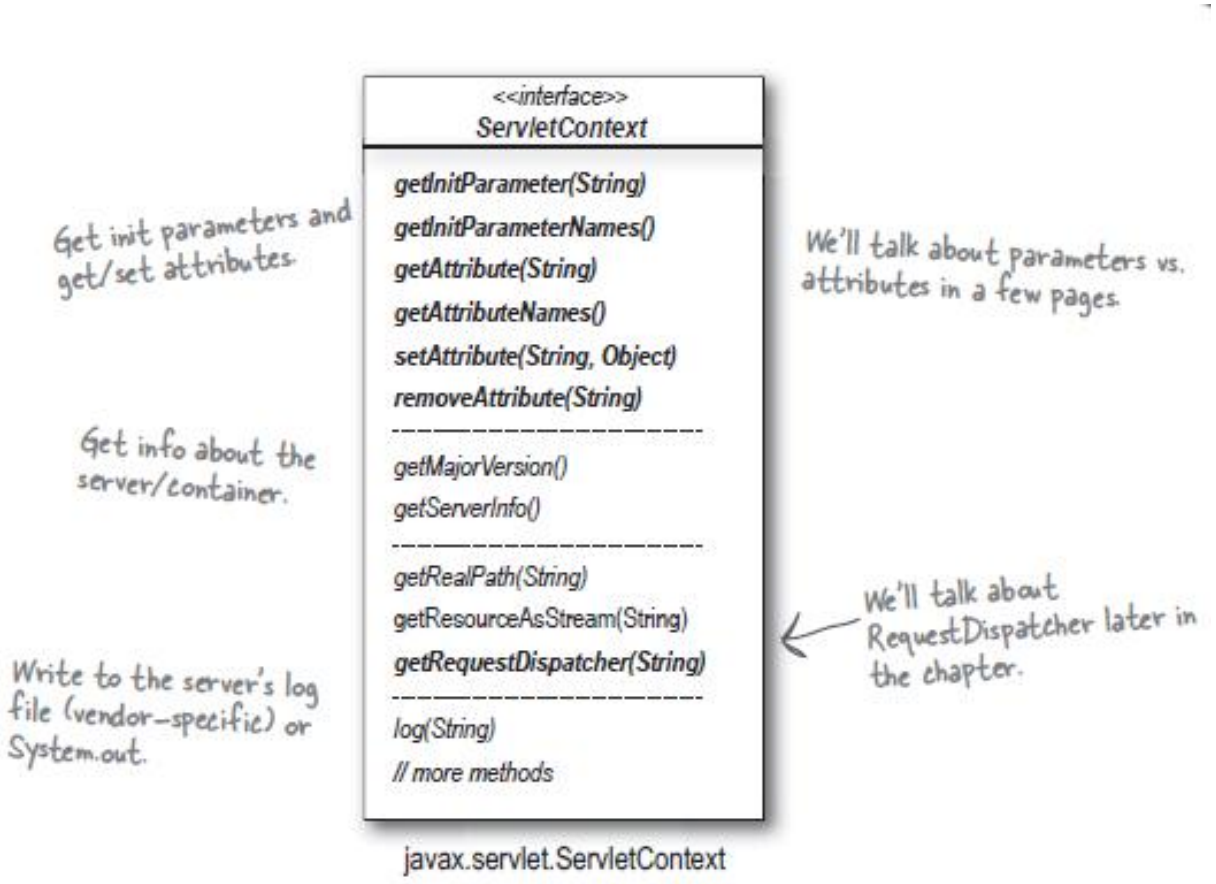
Think of init parameters as deploy-time constants!
We can get them at runtime, but we can't set them. There's no setInitParameter().

We can get ServletContext in two different ways:

getServletConfig().getServletContext(). getInitParameter();

this.getServletContext().getInitParameter();

When we need to use the first way:
The only time we need to go through ServletConfig to get ServletContext, is when we extend a non-generic or non-http type servlet class, which simply implements Servlet. There may be a helper class which may have access to a ServletConfig-type-object and we want to access an associated ServletContext object.

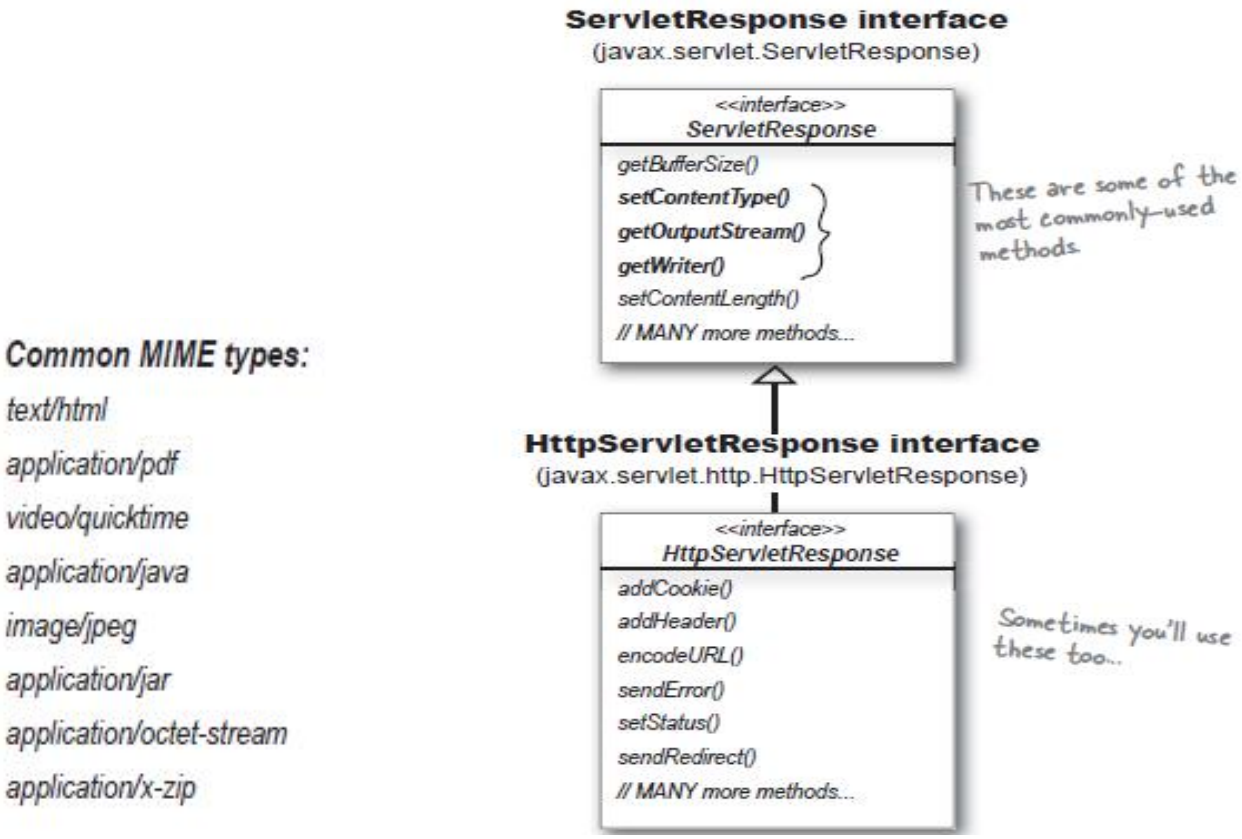


Servlet Lifecycle:

1. The Container initializes a servlet by loading the class, invoking the servlet's no-arg constructor, and calling the servlet's init(ServletConfig cnf) method.
2. The init() method gives the servlet access to the ServletConfig and ServletContext objects, which the servlet needs to get information about the servlet configuration and the web app.
3. The no-arg. init() method (which the developer can override) is called only once in a servlet's life, and always before the servlet can service any client requests.
4. You can override the init() method, and you must override at least one service method (doGet(), doPost(), etc.).
5. The Container ends a servlet's life by calling its destroy() method.
6. Most of a servlet's life is spent running a service() method for a client request.
7. Every request to a servlet runs in a separate thread! There is only one instance of any particular servlet class.

Response:

Response object of type HttpServletResponse has facility methods to generate response.



setContentType():

We have to tell the browser what we're sending back, so the browser can do the right thing: launch a "helper" app like a PDF viewer or video player, render the HTML, save the bytes of the response as a downloaded file, etc. When we say content type we mean the same thing as MIME type.

Content type is an HTTP header that must be included in the HTTP response.

Best practice is to always call setContentType() first, before we call the methods that gives us our output streams(getWriter( ) and getOutputStream( )).

MIME type(Multipurpose Internet Mail Extensions) or Content-Type:

MIME defines mechanisms for sending other kinds of information in email. Also used with communication protocol like HTTP.

getWriter() or getOutputStream():

The ServletResponse interface gives us only two streams to choose from: ServletOutputStream for bytes, or a PrintWriter for character data.

PrintWriter:

```
PrintWriter writer = response.getWriter();
writer.println("some text and HTML");
```

Use it for: Printing text data to a character stream. Although you can still write character data to an OutputStream, this is the stream that's designed to handle character data.

OutputStream:

```
ServletOutputStream out = response.getOutputStream();
out.write(aByteArray);
```

Use it for: Writing anything else!

Other Response Methods:

response.setHeader("hdr","val");
If a header with this name is already in the response, the value is replaced with this value. Otherwise, adds a new header and value to the response.

response.addHeader("hdr","val");
Adds a new header and value to the response, or adds an additional value to an existing header.

response.setIntHeader("hdr",23);
A convenience method that replaces the value of an existing header with this integer value, or adds a new header and value to the response.

setHeader() overwrites the existing value.
addHeader() adds an additional value.

setContentType("text/html")
setHeader("content-type", "text/html");

sendRedirect():
Sometimes we just don't want to deal with the response ourself...

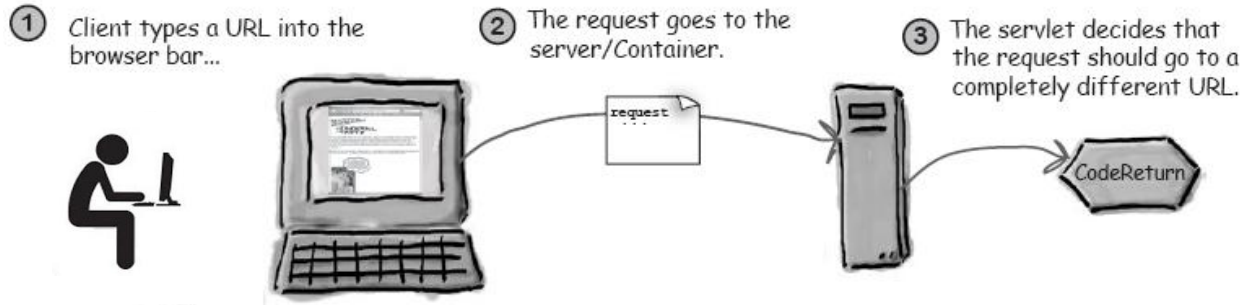
You can choose to have something else handle the response for your request.

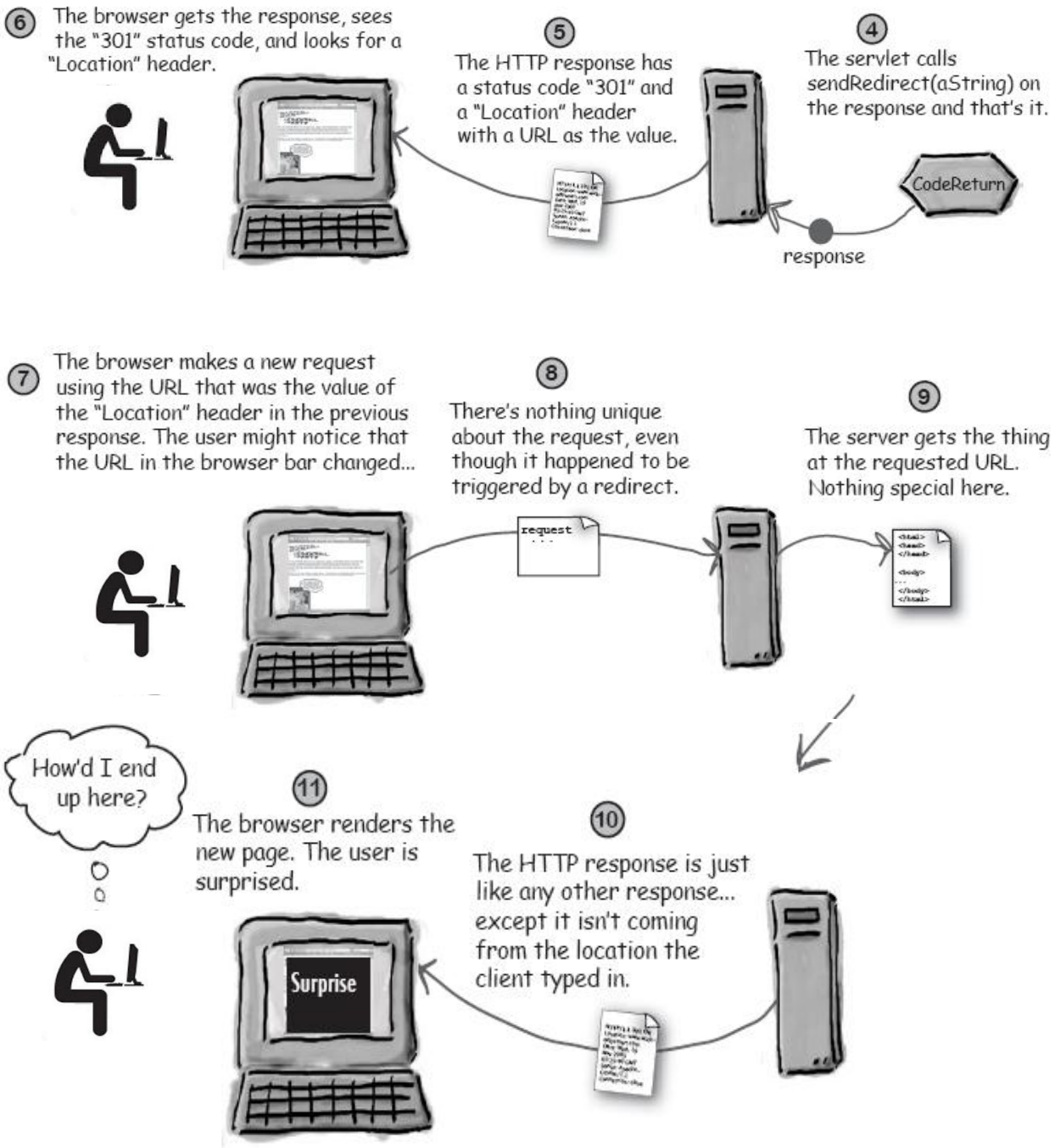
You can either **redirect** the request to a completely different URL, or you can **dispatch** the request to some other component in your web app (typically a JSP).

Servlet redirect makes the browser do the work:

```
response.sendRedirect(String  url);
```

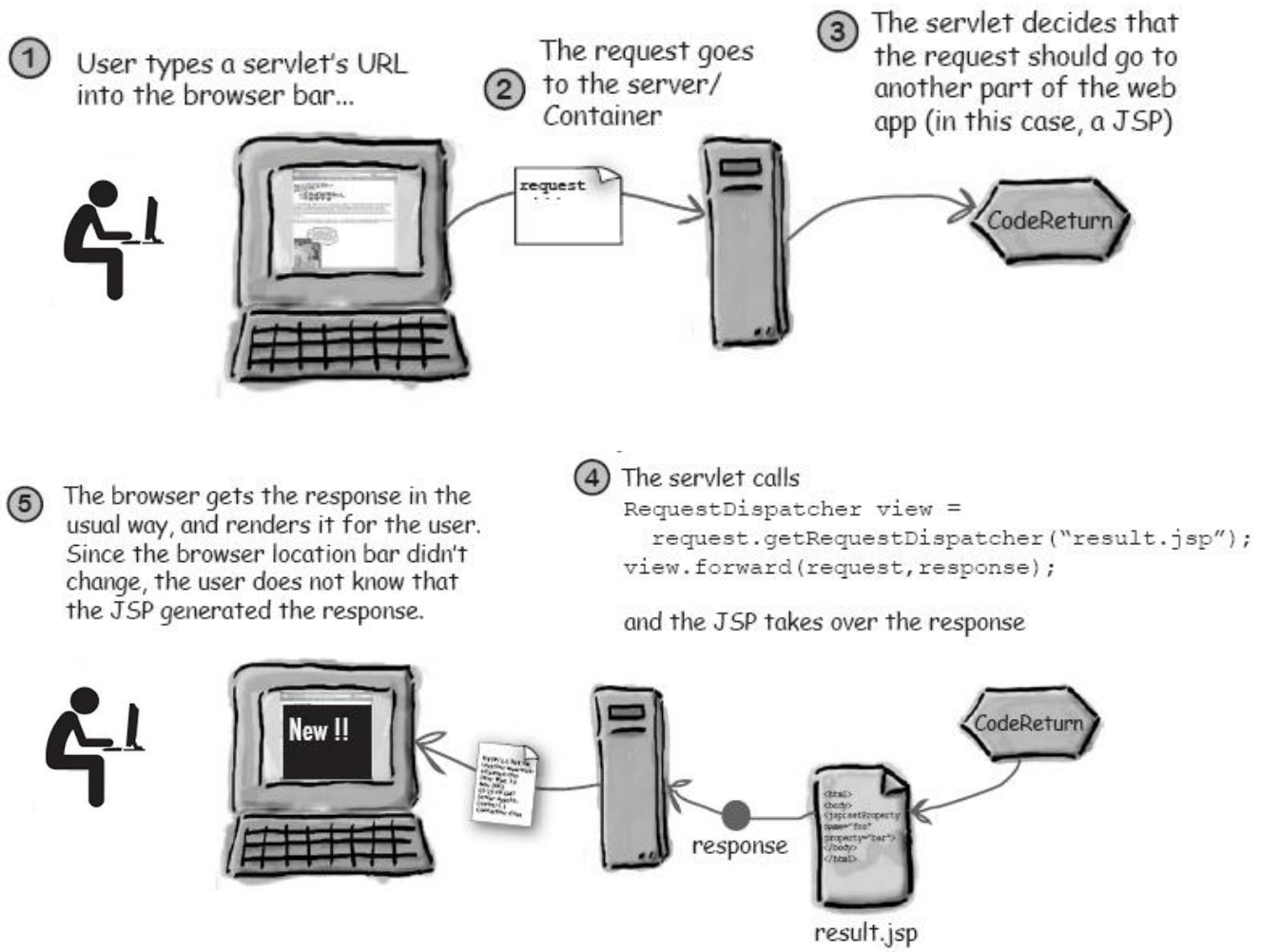
Redirect:





When a servlet does a redirect, it's like asking the client to call someone else instead.
In this case, the client is the browser, not the user.
The browser makes the new call on the user's behalf, after the originally-requested servlet says, "Sorry, call this url instead..."
The user sees the new URL in the browser.

Request Dispatch:



A request dispatch does the work on the server side.

Redirect makes the client do the work while request dispatch makes something else on the server do the work.
So remember:

redirect = client.

request dispatch = server.

Using relative URLs in sendRedirect() :

You can use a relative URL as the argument to sendRedirect(), instead of specifying the whole "http://www..." thing.
Relative URLs come in two flavors: with or without a starting forward slash ("/").

- Imaging a user typed in:
  http://www.mysite.com/myapp/planner/process.do
- When the request comes into the servlet named "process.do", the servlet calls sendRedirect() with a relative URL that does NOT start with a forward slash:
  sendRedirect("prac/stuff.html");

- The Container builds the full URL (it needs this for the "Location" header it puts in the HTTP response) relative to the original request

URL:
http://www.mysite.com/myapp/planner/prac/stuff.html

- But if the argument to sendRedirect() DOES start with a forward slash:
  sendRedirect("/prac/stuff.html");

The forward slash at the beginning means "relative to the root of this Web-Container(Tomcat)".

- The Container builds the complete URL relative to the web Container itself, instead of relative to the original URL of the request. So the new URL will be:
  http://www.mysite.com/prac/stuff.html

prac is a *web-app*. separate from the **myapp** *web-app*.

- You can't do a sendRedirect() after the data has been flushed to the stream, in other words you can't write to the response and then call sendRedirect()!
- sendRedirect() throws IllegalStateException, if you try to invoke it after "the response has already been committed", means the response has been sent.
- This idea that "once it's committed it's too late" also applies to setting headers, cookies, status codes, the content-type, and so on…

The HTTP request Method determines whether doGet() or doPost() runs :

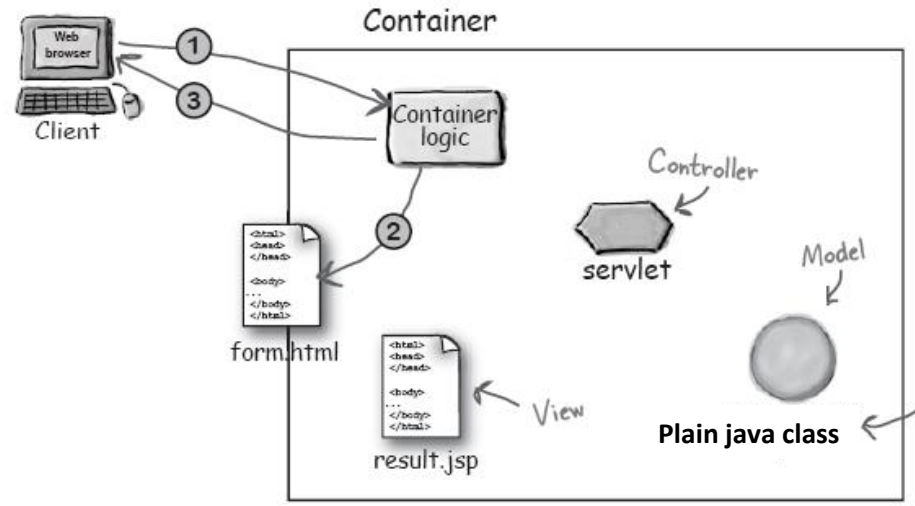
The client's request, always includes a specific HTTP Method.
If the HTTP Method is a GET, the service() method calls doGet().
If the HTTP request Method is a POST, the service() method calls doPost().

GET Asks to get the thing (resource / file) at the requested URL.
POST Asks the server to accept the body info attached to the request, and give it to the thing at the requested URL It's like a fat GET... a GET with extra info sent with the request.
HEAD Asks for only the header part of whatever a GET would return. So it's just like GET, but with no body in the response. Gives you info about the requested URL without actually getting back the real thing.
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.
PUT Says to put the enclosed info (the body) at the requested URL.
DELETE Says to delete the thing (resource/file) at the requested URL.
OPTIONS Asks for a list of the HTTP methods to which the thing at the requested URL can respond.
CONNECT Says to connect for the purposes of tunneling.

The Model-View-Controller (MVC) Design Pattern:

Model-View-Controller (MVC) takes the business logic out of the servlet, and puts it in a "Model"— a reusable plain old Java class. The Model is a combination of the business data (like the state of a Shopping Cart) and the methods (rules) that operate on that data. The clean separation of business logic and presentation Because we are not sure if our business logic will be accessed only from the web!

Here's the Architecture:

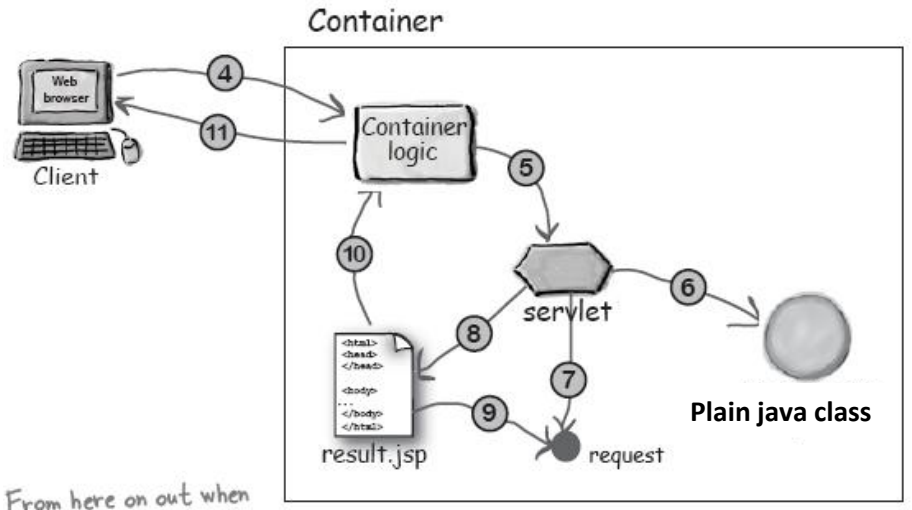


1 - The client makes a request for the *form.html* page.

2 - The Container retrieves the *form.html* page.

3 - The Container returns the page to the browser, where the user answers the questions on the form and...

4 - The browser sends the request data to the container.



5 - The Container finds the correct servlet based on the URL, and passes the request to the servlet.

6 - The servlet calls the BeerExpert for help.

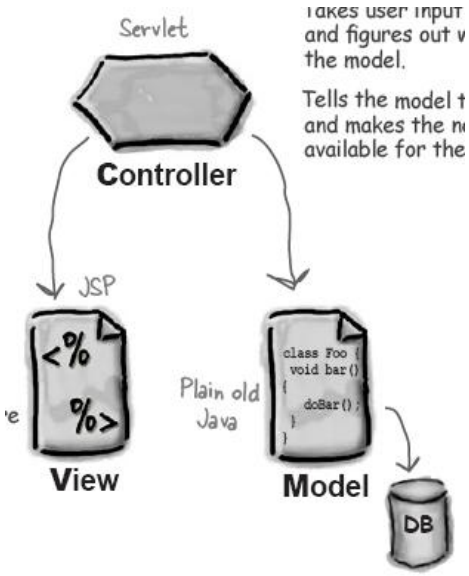
7 - The expert class returns an answer, which the servlet adds to the request object.

8 - The servlet forwards the request to the JSP.

9 - The JSP gets the answer from the request object.

10 - The JSP generates a page for the Container.

11 - The container returns the page to the happy user.



VIEW

1. 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).
2. It's also the part that gets the user input that goes back to the Controller.

CONTROLLER

1. Takes user input from the request and figures out what it means to the model.
2. Tells the model to update itself, and makes the new model state available for the view (the JSP).

MODEL

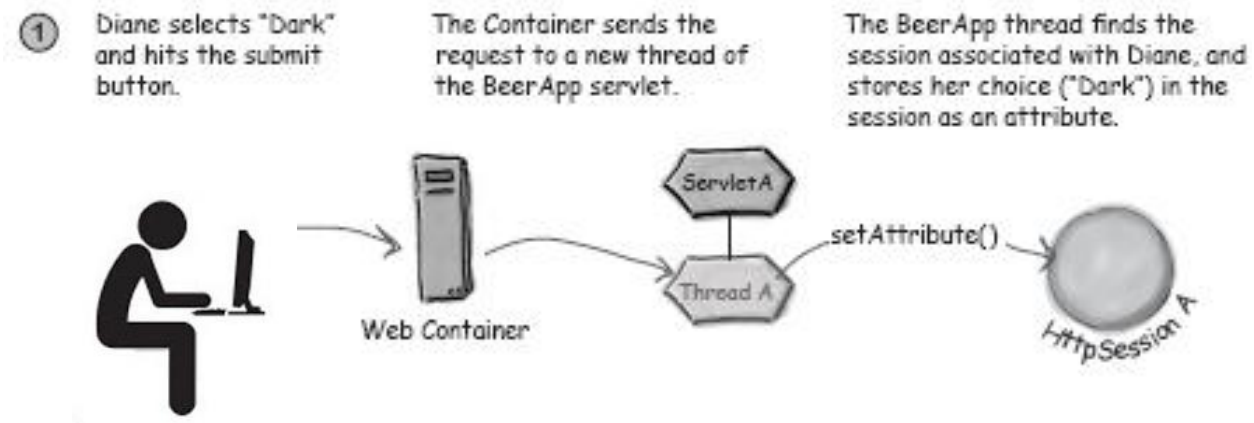
Holds the real business logic and the state. In other words, it knows the rules for getting and updating the state.
A Shopping Cart's contents (and the rules for what to do with it) would be part of the Model in MVC.
It's the only part of the system that talks to the database.

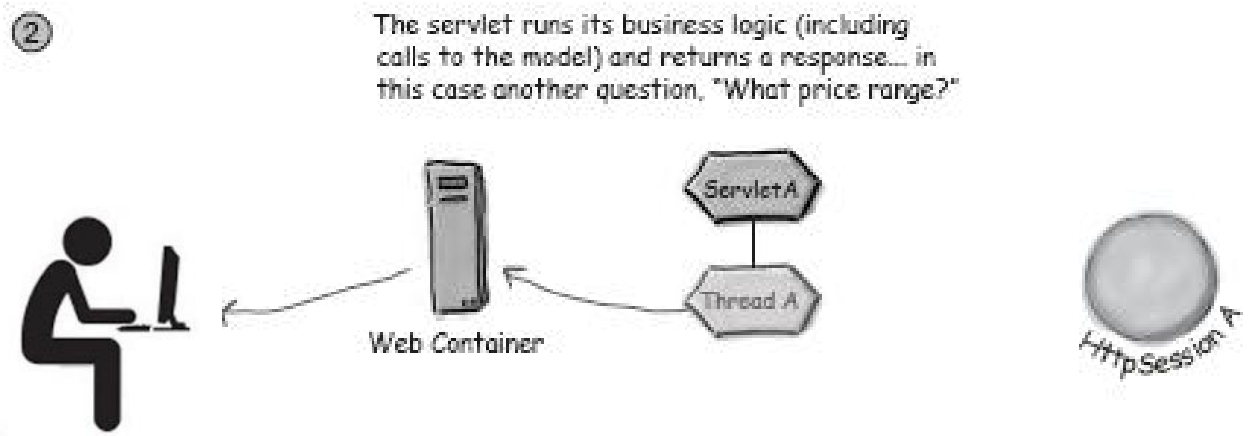
In MVC, the model tends to be the "back-end" of the application. The model shouldn't be tied down to being used by only a single web app, so it should be in its own utility packages.
We don't compile the JSP (the Container does that at first request). The Container provides a mechanism called "request dispatching" that allows one Container-managed component to call another.
The servlet will get the info from the model, save it in the request object, then ***dispatch the request to the JSP.***

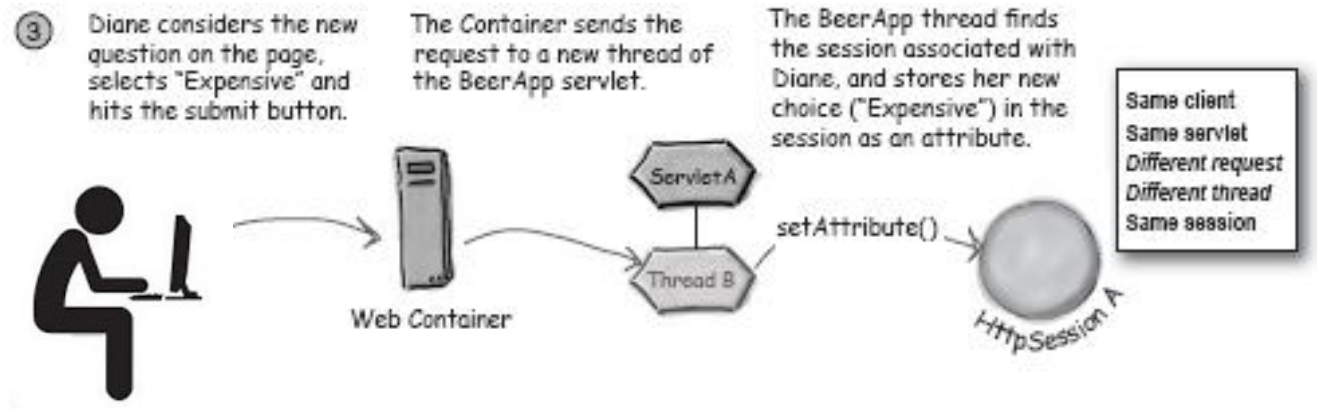
How to keep track of client's conversation:

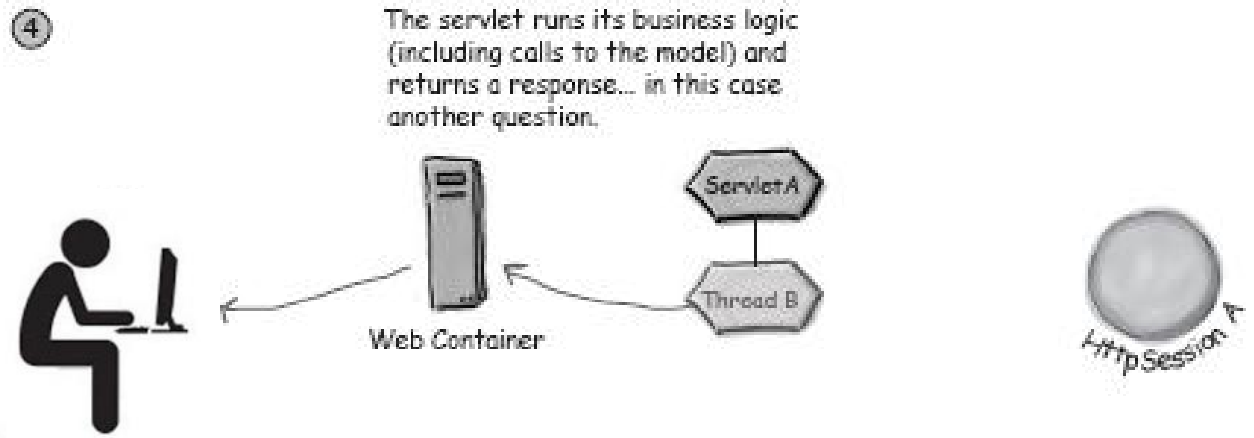
An HttpSession object can hold conversational state across multiple requests from the same client.
In other words, it persists for an entire session with a specific client.
We can use it to store everything we get back from the client in all the requests the client makes during a session.

How session work:









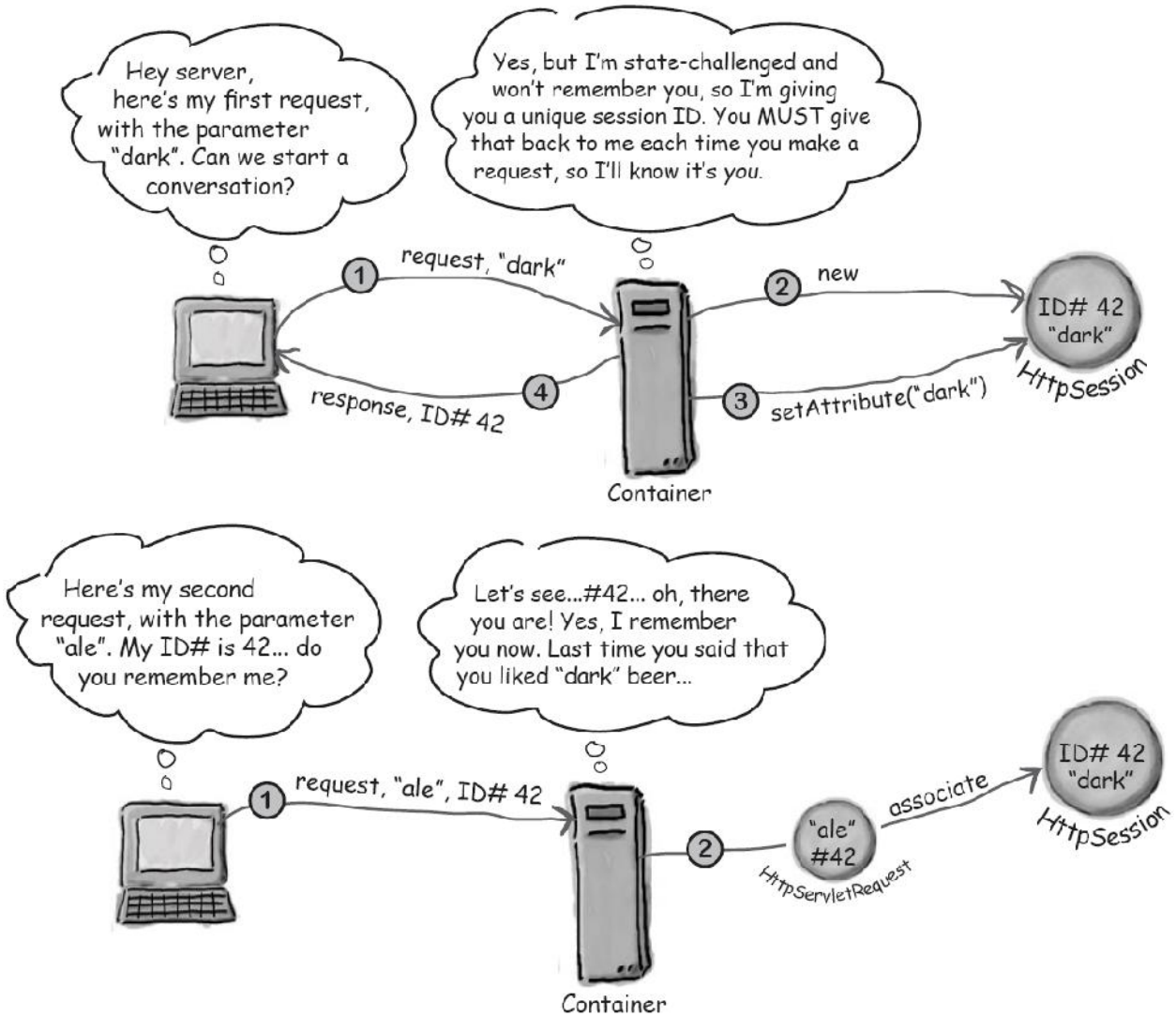
How does the container know who the client is:

The HTTP protocol uses stateless connections. The client browser makes a connection to the server, sends the request, gets the response, and closes the connection. In other words, the connection exists for only a single request/response. As far as the Container's concerned, each request is from a new client.

Now the question is: Why can't the Container just use the IP address of the client? It's part of the request!!
If you're on a local IP network, you have a unique IP address, but chances are, that's not the IP address the outside world sees. To the server, your IP address is the address of the router, so you have the same IP address as everybody else on that network! So that wouldn't help. IP address isn't a solution for uniquely identifying a specific client on the internet.

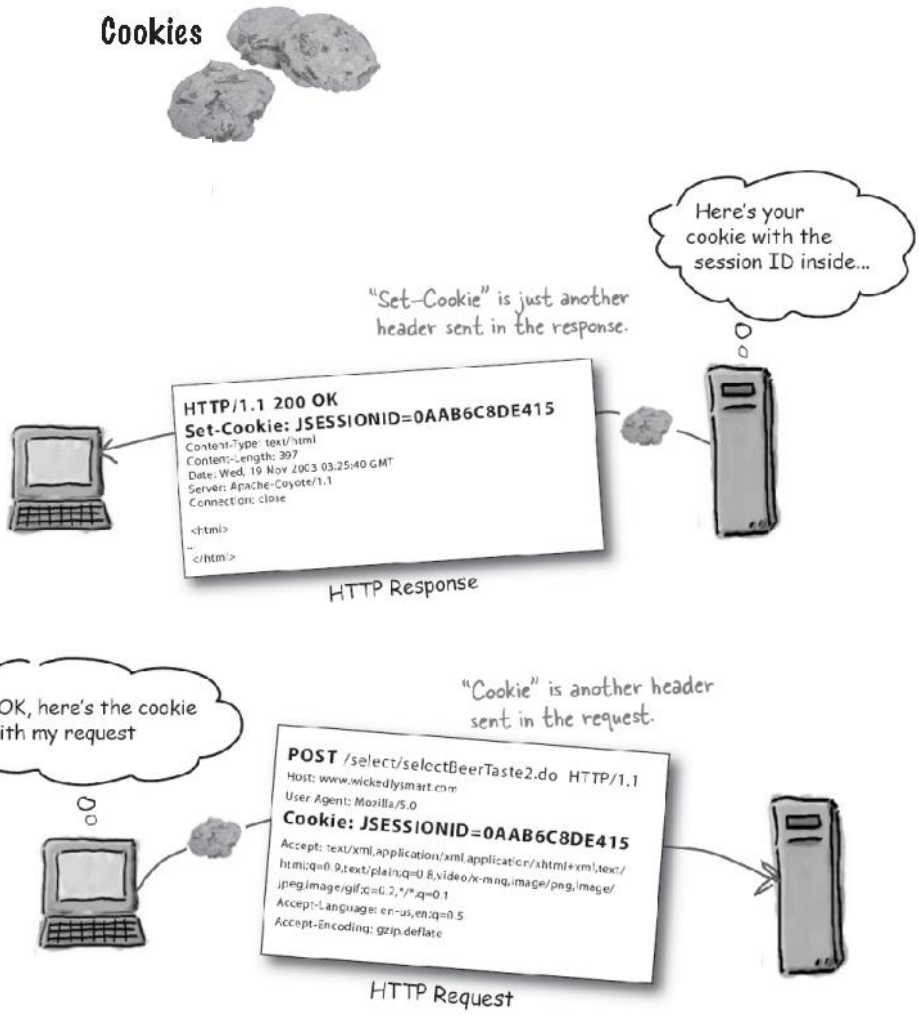
What if user logged through a secure connection:
If the user is logged in and the connection is secure, the Container can identify the client and associate him with a session. But most good web site design says, "don't force the user to log in until it really matters, and don't switch on security (HTTPS) until it really matters. So, we need a mechanism to link a client to a session that doesn't require a securely authenticated client.

The client need a unique session id:
The idea is simple:
On the client's first request, the Container generates a unique session ID and gives it back to the client with the response. The client sends back the session ID with each subsequent request. The Container sees the ID, finds the matching session, and associates the session with the request.



How do the client and container exchange session id:

The Container has to get the session ID to the client as part of the response, and the client has to send back the session ID as part of the request. The simplest and most common way to exchange the info is through *cookies.*



The container does virtually all the cookie work:

We do have to tell the Container that we want to create or use a session:
- but the Container takes care of generating the session ID,
- creating a new Cookie object,
- stuffing the session ID into the cookie, and
- setting the cookie as part of the response.

And on subsequent requests,
- the Container gets the session ID from a cookie in the request,
- matches the session ID with an existing session, and
- associates that session with the current request.

Sending a session cookie in the response:

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

You ask the request for a session, and the Container kicks everything else into action. You don't have to do anything else!

You don’t make the new HttpSession object yourself.
You don’t generate the unique session ID.
You don’t make the new Cookie object.
You don’t associate the session ID with the cookie.
You don’t set the Cookie into the response (under the *Set-Cookie* header*).*
All the cookie work happens behind the scenes.

Getting the session ID from the request:
HttpSession session = **request.getSession();**
The method for GETTING a session ID cookie (and matching it with an existing session) is the same as SENDING a session ID cookie.
IF (the request includes a session ID cookie**)**
find the session matching that ID
ELSE IF (there’s no session ID cookie OR there’s no current session matching the session ID**)**
create a new session.
All the cookie work happens behind the scenes.

Whether the session already existed or was just created?

The no-arg request method, getSession(), returns a session regardless of whether there’s a pre-existing session. The only way to know if the session is new is to **ask the session.**

HttpSession session = **request.getSession();**
…
if (**session.isNew()) {…**

isNew() returns true if the client has not yet responded with this session ID.

What if I want a pre-existing session?
There might have a scenario like checkout sevlet which might not need to start a new session, instead the servlet wants to use only a previously-created session. There is an overloaded getSession(boolean) method just for that purpose. If we don’t want to create a new session, call getSession(false), and we’ll get either null, or a pre-existing HttpSession.

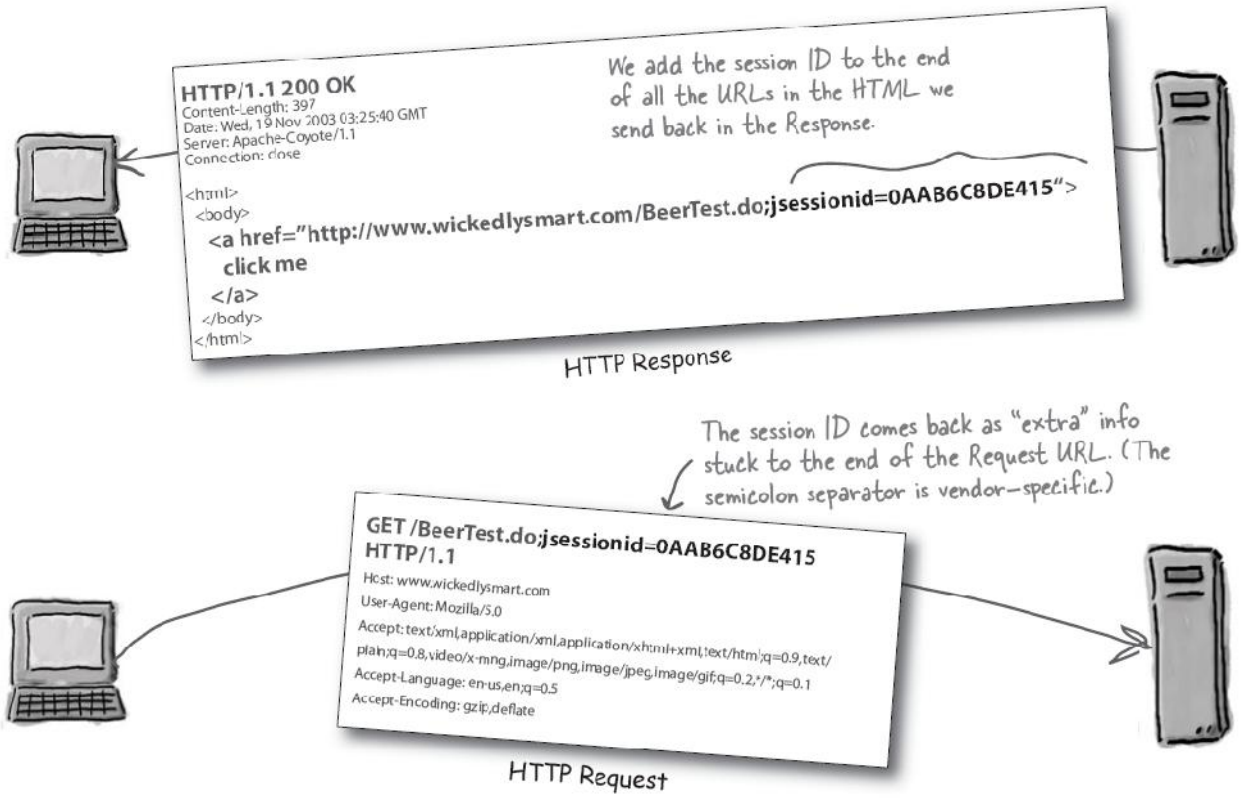
What if Client does not accept Cookies:
If cookies aren’t enabled, it means the client will never join the session. A client with cookies disabled will ignore “Set-Cookie” response headers. The client simply never sends back a request that has a session ID cookie header. In other words, the session’s isNew() method will always return true.If our app depends on sessions, we need a different way for the client and Container to exchange session ID info.

URL rewriting to the rescue:
If the client won’t take cookies, you can use URL rewriting as a back-up. URL rewriting takes the session ID that’s in the cookie and sticks it right onto the end of every URL that comes in to this app. Imagine a web page where every link has a little bit of extra info (the session ID) tacked onto the end of the URL.

When the user clicks that “enhanced” link:

Step 1: The request goes to the Container with that extra bit on the end.

Step 2: The Container simply strips off the extra part of the request URL and uses it to find the matching session.



If you do encode your URLs, the Container will first attempt to use cookies for session management, and fall back to URL rewriting only if the cookie approach fails.

HttPSession’s Key Methods:

getCreationTime()	Returns the time the session was first created.	To find out how old the session is. You might want to restrict certain sessions to a fixed length of time. For example, you might say, “Once you’ve logged in, you have exactly 10 minutes to complete this form...”
getLastAccessedTime()	Returns the last time the Container got a request with this session ID (in milliseconds).	To find out when a client last accessed this session. You might use it to decide that if the client’s been gone a long time you’ll send them an email asking if they’re coming back. Or maybe you’ll invalidate() the session.
setMaxInactiveInterval()	Specifies the maximum time, in seconds, that you want to allow between client requests for this session.	To cause a 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 in your server.
getMaxInactiveInterval()	Returns the maximum time, in seconds, that is allowed between client requests for this session.	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.
invalidate()	Ends the session. This includes *unbinding* all session attributes currently stored in this session. (More on that later in this chapter.)	To kill a session if the client has been inactive or if you KNOW the session is over (for example, after the client does a shopping check-out or logs out). 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.

URL rewriting works with sendRedirect() :

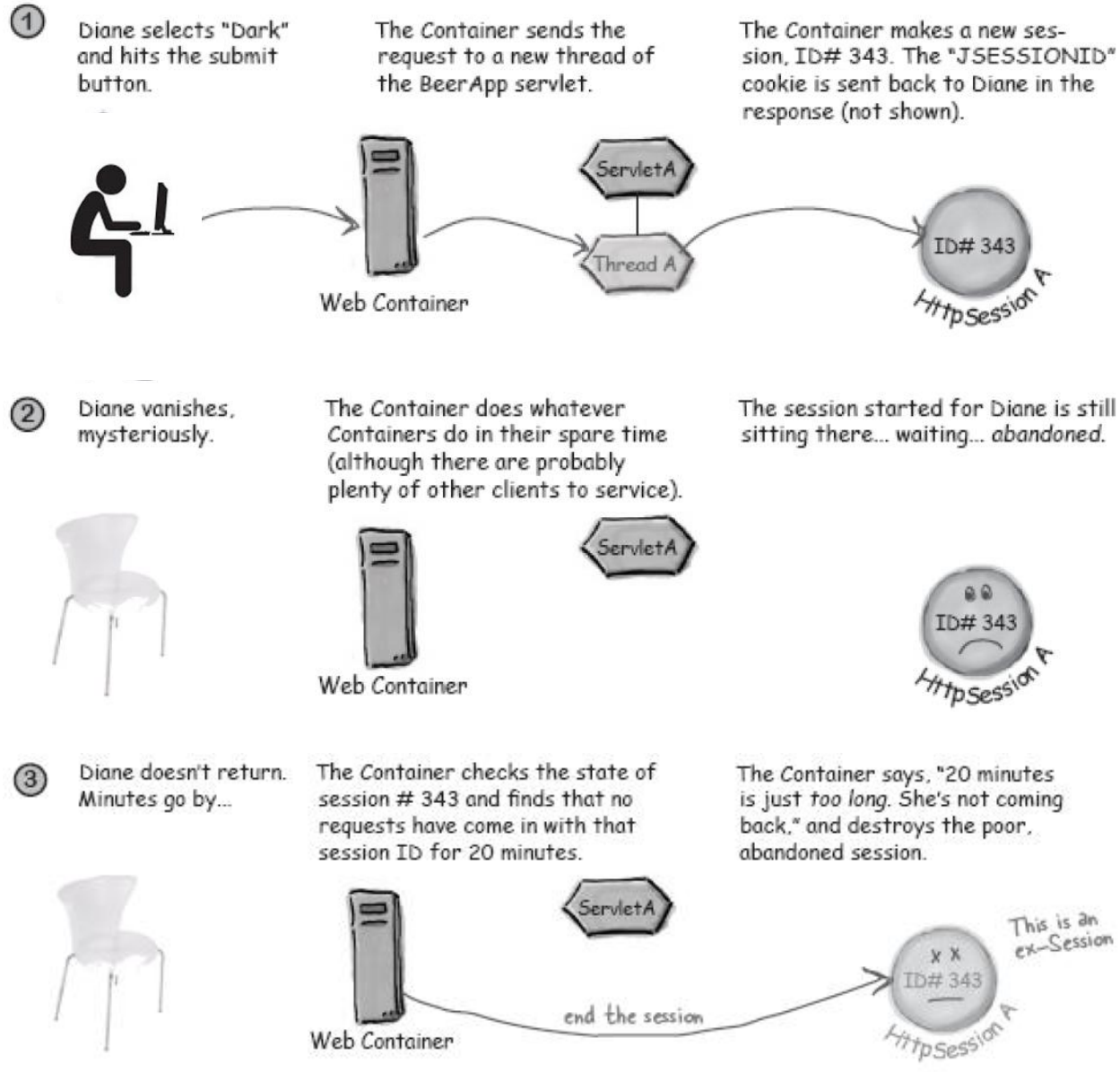
There's a special URL encoding method when we want to use a session with redirect.

response. encodeRedirectURL("some.jsp")

The only way to use URL rewriting is if ALL the pages that are part of a session are dynamically-generated! You can't hard-code session ID's, obviously, since the ID doesn't exist until runtime. URL encoding is the responsibility of HttpServletResponse.

When it's safe for a Container to destroy a session?

The client comes in, starts a session, then changes her mind and leaves the site.
Or the client comes in, starts a session, then her browser crashes.
Or the client comes in, starts a session, and then completes the session by making a purchase (shopping cart check-out).
Or her computer crashes.
The HTTP protocol doesn't have any mechanism for the server to know that the client is gone. We don't want sessions to live longer than necessary, because session-objects holds resources. The Container must recognize when a session has been inactive for too long. But "what too long really means!!!!".



Three ways a session can die:

- When It times out
- You call invalidate() on the session object
- The application goes down (crashes or is undeployed)

Configuring session timeout in the DD:

```
<web-app ...>
  <servlet> ... </servlet>
  <session-config>
    <session-timeout>15</session-timeout>
  </session-config>
</web-app>
```

The "15" is in minutes. This says if the client doesn't make any requests on this session for 15 minutes, kill it

Configuring a timeout in the DD has virtually the same effect as calling setMaxInactiveInterval() on every session that's created.

Setting session timeout for a specific session:

If you want to change the session-timeout value for a particular session instance (without affecting the timeout length for any other sessions in the app):

```
session.setMaxInactiveInterval(20*60);
```

Only the session on which you call the method is affected.

The argument to the method is in seconds, so this says if the client doesn't make any requests on the session for 20 minutes, kill it.*

A runtime exception (IllegalStateException) is thrown if we try to call any session specific method on the session AFTER the session becomes invalid.

Using Cookies:

Cookies were originally designed to help support session state, you can use custom cookies for other things.
A cookie is nothing more than a little piece of data (a name/value String pair) exchanged between the client and server. The server sends the cookie to the client, and the client returns the cookie when the client makes another request. One best thing about cookies is that the user doesn't have to get involved—the cookie exchange is automatic (assuming cookies are enabled on the client). By default, a cookie lives only as long as a session; once the client quits his browser, the cookie disappears. That's how the "JSESSIONID" cookie works. But you can tell a cookie to stay alive even AFTER the browser shuts down.

Everything you need to do with cookies has been encapsulated in the Servlet API in three classes: HttpServletRequest, HttpServletResponse, and Cookie.

Creating a new Cookie: Cookie cookie = new Cookie(“username”, name);

Setting how long a cookie will live on the client: cookie.setMaxAge(30*60);

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(“username”)) {
    String userName = cookie.getValue();
    out.println(“Hello “ + userName);
    break;
  }
}
```

Don’t confuse Cookies with headers!

When you add a header to a response, you pass the name and value Strings as arguments:

```
response.addHeader(“abc”, “123”);
```

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

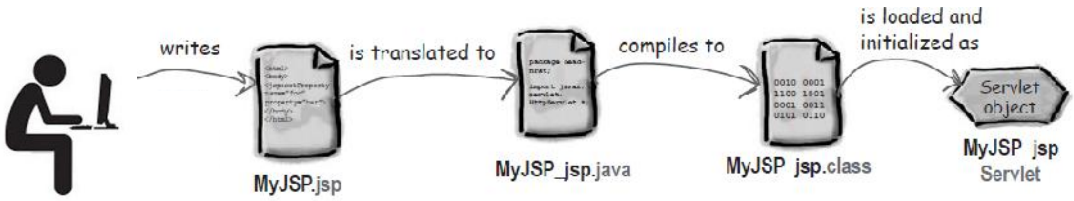
```
Cookie cookie = new Cookie(“name”, name);
response.addCookie(cookie);
```

There’s both a setHeader() and an addHeader() method (addHeader adds a new value to an existing header, if there is one, but setHeader replaces the existing value). But there’s NOT a setCookie() method. There’s only an addCookie() method!

Jsp, Servlet Part - II

Jsp is just a Servlet:

A JSP eventually becomes a full-fledged servlet running in our web app, where the servlet class is written for us, by the Container.
The Container takes what we've written in our JSP, translates it into a servlet class source (.java) file, then compiles that into a Java servlet class.
The Container loads the servlet class, instantiates and initializes it, makes a separate thread for each request, and calls the servlet's service() method.



Jsp element types:

There are four different JSP element types:

1. Scriptlet
2. Expression
3. Directive
4. Declaration

Scriptlet:
We can put regular old Java code in a JSP using a scriptlet - which just means Java code within a **<% ... %>** tag.

Directive:
A directive is a way for you to give special instructions to the Container at page translation time. Directives come in three flavors: **page**, **include**, and **taglib**.

To import a single package:

```
<%@ page import="abc.*" %>
<html>
 <body>
  The page count is:
  <%  out.println(Counter.getCount());  %>
 </body>
</html>
```

This is a page directive with an import attribute.
(Notice there's no semicolon at the end of a directive.)

Scriptlets are normal Java, so all statements in a scriptlet must end in a semicolon!

To import multiple packages:

```
<%@ page import="abc.*, java.util.*" %>
```

Use a comma to separate the packages.
The quotes go around the entire list of packages!

The Java code is between angle brackets with percent signs: <% and %>. But the directive adds an additional character to the start of the element—the @ sign. If you see JSP code that starts with <%@, you know it's a directive.

Expression:

Part of the whole point of JSP is to avoid println()! That's why there's a JSP expression element - it automatically prints out whatever you put between the tags.

The scriptlet code is between angle brackets with percent signs: <% and %>. But the expression adds an additional character to the start of the element—an equals sign (=). **Expression: <%= %>**

Scriptlet code:

```
<%@ page import="abc.*" %>
<html>
 <body>
  The page count is:
  <% out.println(Counter.getCount()); %>
 </body>
</html>
```

Expression code:

```
<%@ page import="abc.*" %>
<html>
 <body>
  The page count is now:
  <%= Counter.getCount() %>
 </body>
</html>
```

Expressions become the argument to an out.print(). The Container takes everything you type between the <%= and %> and puts it in as the argument to a statement that prints to the implicit response PrintWriter out.

When the Container sees this: <%= Counter.getCount() %>

It turns it into this: out.print(Counter.getCount());

If you did put a semicolon in your expression:

<%= Counter.getCount(); %>

That would be bad. It would mean this:

out.print(Counter.getCount();); ← This will never compile

Count page visits – example! ALL scriptlet and expression code lands in a service method. That means variables declared in a scriptlet are always LOCAL variables!

Declaration:

JSP declarations are for declaring members of the generated servlet class. That means both variables and methods! In other words, anything between the <%! and %> tag is added to the class outside the service method. That means you can declare both static variables and methods.

<%! int count = 0; %>

Variable Declaration

This JSP:

```
<html><body>
<%! int count=0; %>
The page count is now:
<%= ++count %>
</body></html>
```

Becomes this servlet:

```
public class basicCounter_jsp extends SomeSpecialHttpServlet {

  int count=0;

  public void _jspService(HttpServletRequest request,
    HttpServletResponse response)throws java.io.IOException {

      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.write("<html><body>");
      out.write("The page count is now:");
      out.print( ++count );
      out.write("</body></html>");
  }
}
```

This time, we're incrementing an instance variable instead of a local variable.

Method Declaration

This JSP:

```
<html>
<body>
<%! int doubleCount() {
      count = count*2;
      return count;
    }
%>
<%! int count=1; %>
The page count is now:
<%= doubleCount() %>
</body>
</html>
```

Becomes this servlet:

```
public class basicCounter_jsp extends SomeSpecialHttpServlet {

    int doubleCount() {
       count = count*2;
       return count;
    }
    int count=1;

    public void _jspService(HttpServletRequest request,
      HttpServletResponse response)throws java.io.IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body>");
        out.write("The page count is now:");
        out.print( doubleCount() );
        out.write("</body></html>");
    }
}
```

The method goes in just the way you typed it in your JSP.

It's Java, so no problem with forward-referencing (declaring the variable AFTER you used it in a method).

What the Container does with your JSP?

It looks at the directives, for information it might need during translation. Creates an HttpServlet subclass. For Tomcat 5, the generated servlet extends: **org.apache.jasper.runtime.HttpJspBase**

If there's a page directive with an import attribute, it writes the import statements at the top of the class file, just below the package statement.

For Tomcat 5, the package statement (which you don't care about) is: **package org.apache.jsp;**

If there are declarations, it writes them into the class file, usually just below the class declaration and before the service method. Tomcat 5 declares one static variable and one instance method of its own.

Builds the service method. The service method's actual name is _jspService(). It's called by the servlet superclass' overridden service() method, and receives the HttpServletRequest and HttpServletResponse. As part of building this method, the Container declares and initializes all the **implicit objects**. Combines the plain old HTML (called template text), scriptlets, and expressions into the service method, formatting everything and writing it to the PrintWriter response output. The generated code contains the three JSP lifecycle methods:

jspInit()
jspDestroy
_jspService()

When a Container translates the JSP into a servlet, it inserts a pile of implicit object declarations and assignments in the beginning of the service method. With implicit objects, we can write a JSP knowing that our code is going to be part of a servlet. In other words, we can take advantage of our servletness, even though we are not directly writing a servlet class ourself. All of the implicit objects map to something from the Servlet/JSP API.

API for the generated servlet:

1. jspInit() This method is called from the init() method. You can override this method.
2. jspDestroy() This method is called from the servlet's destroy() method. You can override this method as well.
3. _jspService() This method is called from the servlet's service() method, which means it runs in a separate thread for each request.

The Container passes the Request and Response objects to this method. You can't override this method! You can't do anything with this method yourself (except write code that goes inside it), and it's up to the Container vendor to take your JSP code and fashion the _jspService() method that uses it.

API		Implicit Object
JspWriter	_____	out
HttpServletRequest	_____	request
HttpServletResponse	_____	response
HttpSession	_____	session
ServletContext	_____	application
ServletConfig	_____	config
Throwable	_____	exception
PageContext	_____	pageContext
Object	_____	page

You can put two different types of comments in a JSP:

<!-- HTML comment -->

The Container passes this straight on to the client, where the browser interprets it as a comment.

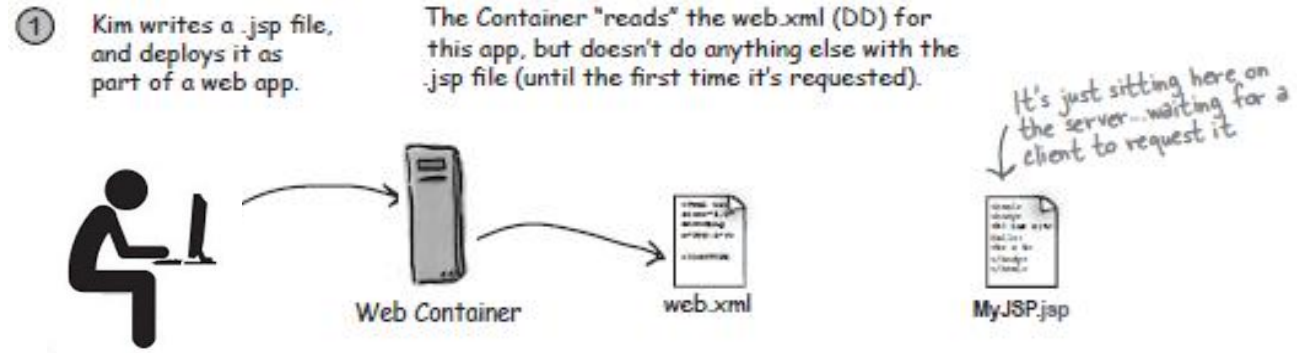
If you want comments to stay as part of the HTML response to the client, use an HTML comment. (Although the browser will hide them from the client's view)

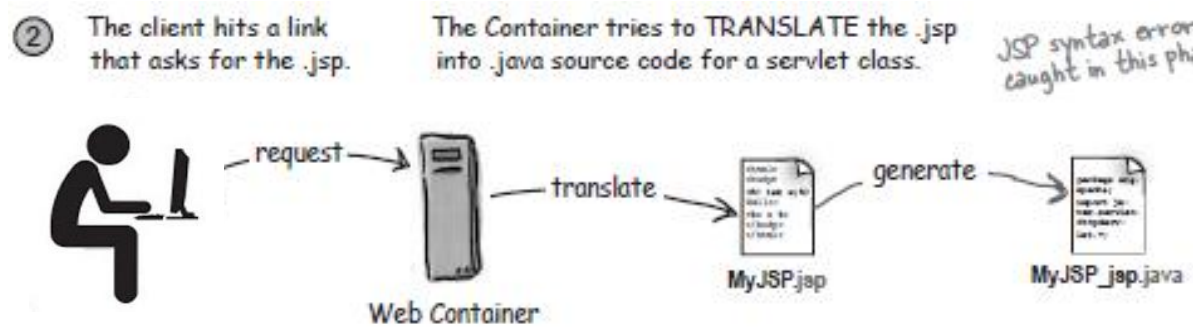
<%-- JSP comment --%>

These are for the page developers, and just like Java comments in a Java source file, they're stripped out of the translated page.

Lifecycle of a JSP:

You write the .jsp file. The Container writes the .java file for the servlet your JSP becomes.





③

The Container tries to COMPILE the servlet .java source into a .class file.

Java language/syntax errors are caught here.

④

The Container LOADS the newly-generated servlet class.

⑤

The Container instantiates the servlet and causes the servlet's jspInit() method to run.

The object is now a full-fledged servlet, ready to accept client requests.

⑤

The Container instantiates the servlet and causes the servlet's jspInit() method to run.

The object is now a full-fledged servlet, ready to accept client requests.

⑥

The Container creates a new thread to handle this client's request, and the servlet's _jspService() method runs.

Everything that happens after this is just plain old servlet request-handling.

Eventually the servlet sends a response back to the client (or forwards the request to another web app component).

When you deploy a web app with a JSP, the whole translation and compilation step happens only once in the JSP's life.

Once it's been 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. So the picture on the previous pages is for only *the first request.*

Initializing a Jsp:

How to Configuring servlet init parameters:

```
<web-app ...>
  <servlet>
    <servlet-name>MyTestInit</servlet-name>
    <jsp-file>/TestInit.jsp</jsp-file>
    <init-param>
      <param-name>email</param-name>
      <param-value>ikickedbutt@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyTestInit</servlet-name>
    <url-pattern>/TestInif.jsp</url-pattern>
  </servlet-mapping>
</web-app>
```

This is the only line that's different from a regular servlet. It basically says, "apply everything in this <servlet> tag to the servlet created from this JSP page..."

When you define a servlet for a JSP, you must also define a servlet mapping to the JSP page.

Overriding jspInit():

If you implement a jspInit() method, the Container calls this method at the beginning of this page's life as a servlet. It's called from the servlet's init() method, so by the time this method runs there is a ServletConfig and ServletContext available to the servlet. That means you can call getServletConfig() and getServletContext() from within the jspInit() method.

```
<%!
  public void jspInit() {
    ServletConfig sConfig = getServletConfig();
    String emailAddr = sConfig.getInitParameter("email");
    ServletContext ctx = getServletContext();
    ctx.setAttribute("mail", emailAddr);
  }
%>
```

Override the jspInit() method using a declaration.

You're in a servlet, so you can call your inherited getServletConfig() method!

This is EXACTLY what you'd do in a normal servlet.

Get a reference to the ServletContext and set an application-scope attribute.

Jsp Attribute Scopes:

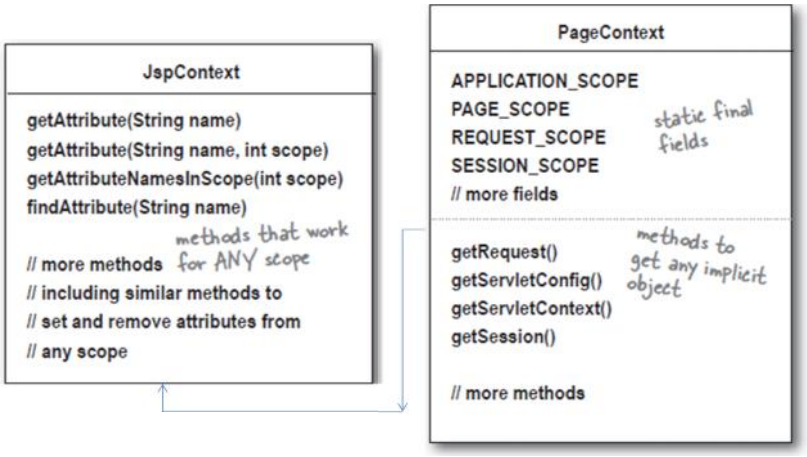
There are 4 attribute scopes available in jsp as opposed to Servlets. In addition to the standard *servlet scope - request*, *session*, and *application (context)* scopes, a JSP adds a fourth scope— *page scope* —that we get from a pageContext object.

Most of the time the page scope is useful in custom tag development. Where another non-server managed component plays the important role. The page-scope implicit object provides access to the other scoped variables.

	In a Servlet	**In a JSP** (using implicit objects)
Application	getServletContext().setAttribute("abc",obj);	application.setAttribute("abc",obj);
Request	request.setAttribute("abc",obj);	request.setAttribute("abc",obj);
Session	request.getSession().setAttribute("abc",obj);	session.setAttribute("abc",obj);
Page	Does not apply!	pageContext.setAttribute("abc",obj)

Using PageContext for attributes:

You can use a PageContext reference to get attributes from any scope.



Examples using pageContext to get and set attributes:

Setting a page-scoped attribute: <% Float one = new Float(42.5); %> <% pageContext.setAttribute("abc", one); %>	**Getting a page-scoped attribute:** <%= pageContext.getAttribute("abc") %>
Using the pageContext to set a session-scoped attribute: <% Float two = new Float(22.4); %> <% pageContext.setAttribute("abc", two, PageContext.SESSION_SCOPE); %>	
Using the pageContext to get an application-scoped attribute Email is: <%= pageContext.getAttribute("mail", PageContext.APPLICATION_SCOPE) %> Within a JSP, the code above is identical to: Email is: <%= application.**getAttribute("mail") %>**	

There are 2 overloaded getAttribute() methods you can call on pageContext: a one-arg that takes a String, and a two-arg that takes a String and an int. The one-arg version works just like all the others—it's for attributes bound to the pageContext object. But the two-arg version can be used to get an attribute from any of the four scopes.

Using the pageContext to find an attribute when you don't know the scope:

<%= pageContext.findAttribute("abc") %>

the method starts looking in other scopes, from most restricted to least restricted scope —

It looks: first in the **page context**, but if there's no "abc" attribute then, finds in the **request scope**, then **session scope**, then finally finds in the **application scope**. The first one it finds with that name wins.

Using <scripting-invalid>

By putting a <scripting-invalid> tag in the DD, one can make scripting invalid:

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

Standard Action & EL

Our MVC Architecture depends on Attribute Scopes.

Servlet (controller) code:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
                                        throws IOException, ServletException {
     String name = request.getParameter("userName");
     request.setAttribute("name", name);
     RequestDispatcher view = request.getRequestDispatcher("/result.jsp");
     view.forward(request, response);
}
```

JSP (view) code:

```
<html>
  <body>
    Hello <%= request.getAttribute("name")  %>
  </body>
</html>
```

But what if the attribute is not a String, but an instance of an existing class like Person?

Not just a Person, but a Person with a "name" property.

Person class has a getName() and setName() method pair, which in the JavaBean spec means Person has a property called "name".

The name of the property is what you get when you strip off the prefix "get" and "set", and make the first character after that lower case. so, getName/setName becomes **name.**

```
package abc;
class  Person {
   String  name;

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

   public String  getName(){
       return  name;
   }
}
```

Servlet code:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
                        throws IOException, ServletException {
    abc.Person p = new  abc.Person();
    p.setName("Evan");
    request.setAttribute("person", p);
    RequestDispatcher view = request.getRequestDispatcher("result.jsp");
    view.forward(request, response);
}
```

JSP code:

```
<html>
  <body>
    Person is: <%= request.getAttribute("person")  %>
  </body>
</html>
```

We need more code to get the Person's name:

Sending the result of getAttribute() to print/write statement doesn't give us what we want—it just runs the object's toString() method.

Class Person doesn't override its inherited Object.toString(), but we want to print the Person's *name.*

JSP code :

```
<html>
  <body>
    <% abc.Person p = (abc.Person) request.getAttribute("person"); %>
    Person is: <%= p.getName() %>
  </body>
</html>
```

Person is a JavaBean, so we'll use the bean-related standard actions:

Using standard actions, we can eliminate all the scripting code in our JSP and still print out the value of the person attribute's name property.

Remember name is not an attribute—only the person object is an attribute. The name property is simply the thing returned from a Person's getName() method.

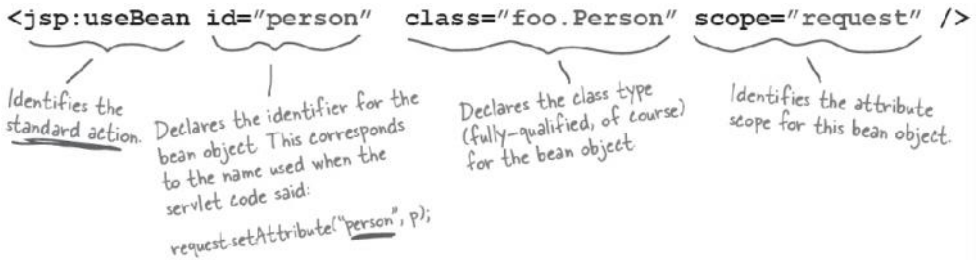
Without standard actions (using scripting):

```
<html>
  <body>
    <% abc.Person p = (abc.Person) request.getAttribute("person");  %>
    Person is: <%= p.getName() %>
  </body>
</html>
```

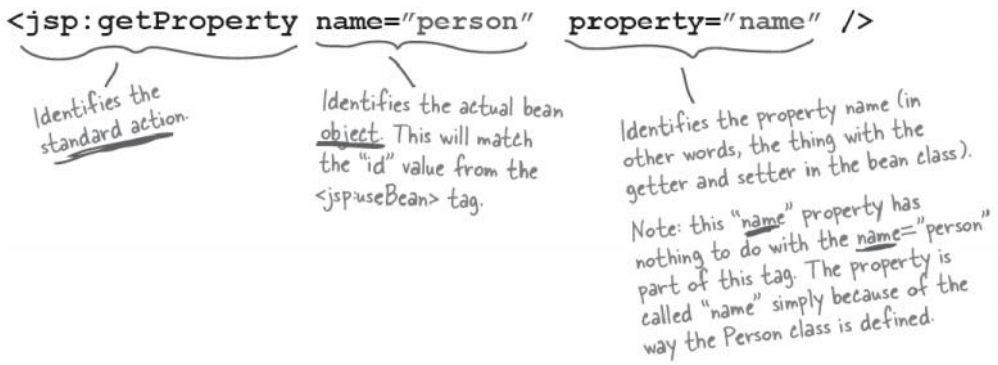
With standard actions (no scripting):

```
<html>
  <body>
    <jsp:useBean  id="person" class="abc.Person" scope="request" />
    Person created by servlet:
    <jsp:getProperty  name="person"  property="name" />
  </body>
</html>
```

Declare and initialize a bean attribute with <jsp:useBean>

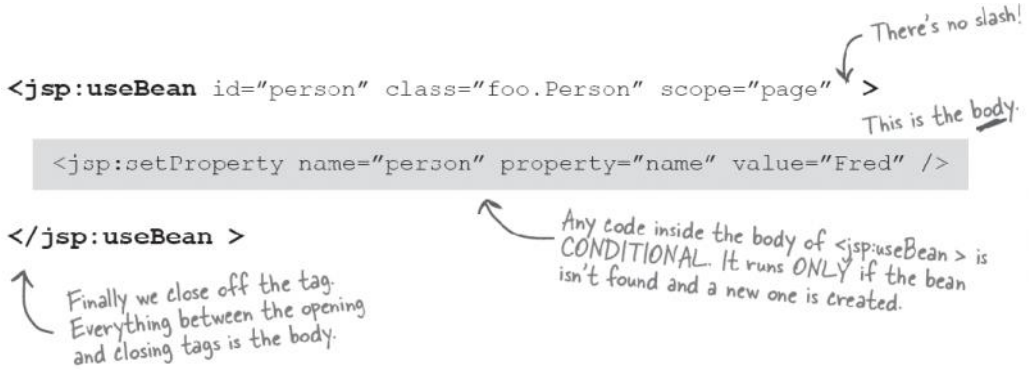


Get a bean attribute's property value with <jsp:getProperty>



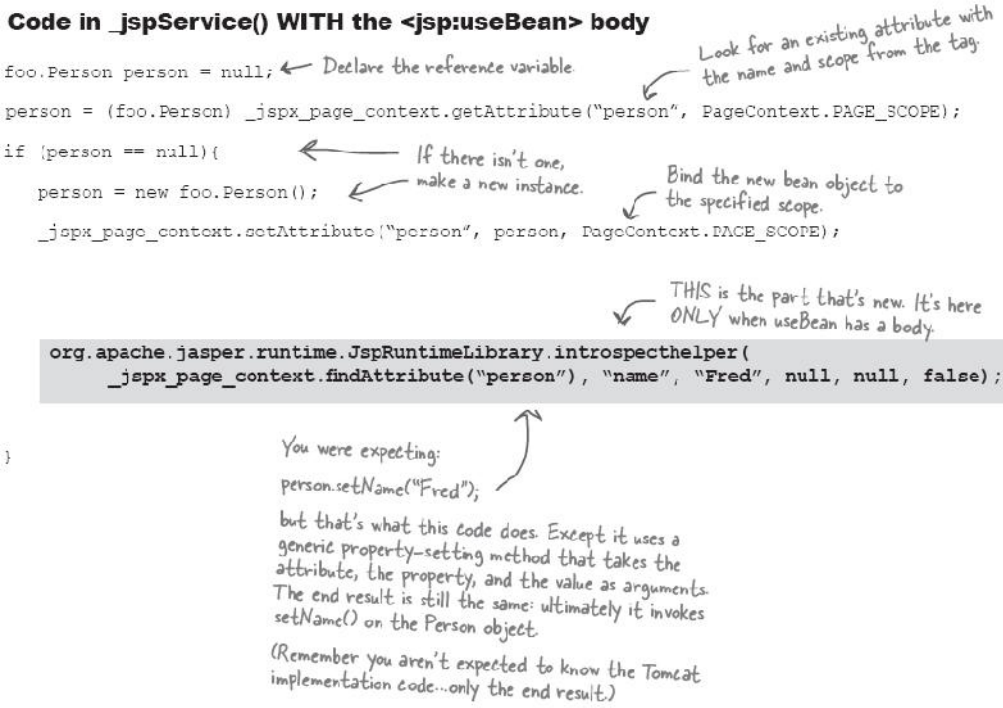
<jsp:useBean> can also create a bean!
If the <jsp:useBean> can't find an attribute object named "person", it creates one.
The Tag:
<jsp:useBean id="person" **class**="abc.Person" **scope**="request" **/>**
After translation - in the _jspService() method: *see the translation* in work directory.

We can use <jsp:setProperty>:
What if we don't WANT to have a bean that doesn't have its property values set!
<**jsp:useBean id**="person" **class**="abc.Person" **scope**="request" />
<**jsp:setProperty name**="person" **property**="name" **value**="Mohan" />
It means the JSP will reset the property name even if the bean already existed.
We want to set the property only when the bean is new.
<jsp:useBean> can have a body:
If we put our setter code (<jsp:setProperty>) inside the body of <jsp:useBean>, the property setting is conditional! In other words, the property values will be set only if a new bean is created. If an existing bean with that scope and id are found, the body of the tag will never run, so the property won't be reset from our JSP code.



Bean Law states that only the bean's public, no-arg constructor will be called. It means if we do not have a public no-arg constructor in our bean class, then the entire concept will fail. Generated servlet when <jsp:useBean> has a body: The Container puts the extra property-setting code inside the *if test.*

Generated servlet when <jsp:useBean> has a body:



Can you make polymorphic bean references?
When you write a <jsp:useBean>, the class attribute determines the class of the new object (if one is created).
It also determines the type of the reference variable used in the generated servlet.

The way it is NOW in the JSP :

```
<jsp:useBean  id="person"  class="abc.Person"  scope="page" />
```

Generated servlet:

```
abc.Person person = null;
// code to get the person attribute
if (person == null){
   person  =  new  abc.Person();
   . . .
```

The class attribute in the tag represents both the reference and object type.

But, what if we want the reference type to be different from the actual object type?
We'll change the Person class to make it abstract, and make a concrete subclass Employee. Imagine we want the reference type to be Person, and the new object type to be Employee.

```
package abc;

public abstract class Person {
   private String name;

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

   public String getName() {
      return name;
   }
}
```

```
package  abc;

public class Employee extends Person {
    private int empID;

    public void setEmpID(int empID) {
       this.empID = empID;
    }

    public int getEmpID() {
       return empID;
    }
}
```

Adding a type attribute to <jsp:useBean>:
With the changes we just made to the Person class, we're in trouble if the attribute can't be found:

Our original JSP :

```
<jsp:useBean  id="person"  class="abc.Person"  scope="page" />
```

Has this result :

```
java.lang.InstantiationException: abc.Person
```

Because the Container tries to:

```
new  abc.Person();
```

Person is now abstract!
Obviously You can't make one,
but the Container still tries,
based on the class attribute
in the tag

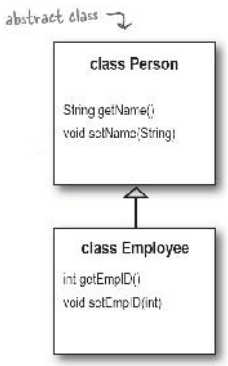
We need to make the reference variable type Person, and the object an instance of class Employee.
Adding a '*type*' attribute to the tag lets us do that.
Type can be a class type, abstract type, or an interface—anything that you can use as a declared reference type for the class type of the bean object.

Our new JSP with a type:

```
<jsp:useBean id="person" type="abc.Person" class="abc.Employee" scope="page" />
```

Generated servlet :

```
abc.Person person  =  null;
// code to get the person attribute
if (person == null){
  person  =  new  abc.Employee();
...
```



You can't violate Java typing rules, of course. It means the class must be a subclass or concrete implementation of the type.

Using type without class:

What happens if we declare a type, but not a class? Does it matter if the type is abstract or concrete?

JSP:

<jsp:useBean id=”person” type=”abc.Person” scope=”page”/>

-Result if the person attribute already exists in “page” scope:

It works perfectly.

- Result if the person attribute does NOT exist in “page” scope:

java.lang.InstantiationException: bean person not found within scope

* If type is used without class, the bean must already exist.

* If class is used (with or without type) the class must NOT be abstract, and must have a public no-arg constructor.

What if we change the type to “abc.Employee”? Will it use the type for both the reference AND the object type?

NO! It never works. If the Container discovers that the bean doesn’t exist, and it sees only a type attribute without a class, it knows that, you haven’t told it what to make a new instance of!

The scope attribute defaults to “page”:

If you don’t specify a scope in either the <jsp:useBean> or <jsp:getProperty> tags, the Container uses the default of “page”.

This

<jsp:useBean id=”person” class=”abc.Employee” scope=”page” />

Is the same as this

<jsp:useBean id=”person” class=”abc.Employee” />

Don’t confuse type with class!

Check out this code:

<jsp:useBean id=”person” type=”abc.Employee” class=”abc.Person” />

Remember:

type == reference type

class == object type

Or

In other words:

type is what you DECLARE (can be abstract)

class is what you INSTANTIATE (must be concrete)

type x = new class()

Going straight from the request to the JSP without going through a servlet...

Imagine this is our form:

```
<html>
<body>
  <form action=“TestBean.jsp”>
    name: <input type=“text” name=“userName”>
    ID#: <input type=“text” name=“userID”>
    <input type=”submit”>
  </form>
</body>
</html>
```

We know we can do it with a combination of standard actions and scripting:

<jsp:useBean id=“person” type=“abc.Person” class=“abc.Employee”/>

<% person.setName(request.getParameter(“userName”)); %>

We can even do it with scripting INSIDE a standard action:

<jsp:useBean id=“person” type=“abc.Person” class=“abc.Employee”>

<jsp:setProperty name=“person” property=“name” value=“<%= request.getParameter(“userName”) %>” />

</jsp:useBean>

The param attribute to the rescue:

We can send a request parameter straight into a bean, without scripting, using the *param* attribute.

Inside Test.jsp:

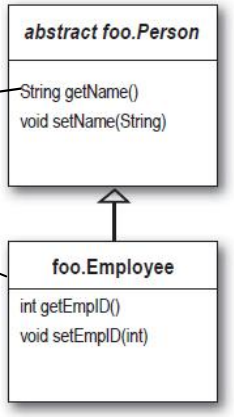
```
<jsp:useBean id="person" type="abc.Person" class="abc.Employee">
    <jsp:setProperty name="person" property="name" param="userName" />
</jsp:useBean>

<html>
<body>
  <form action="Test.jsp">
     name: <input type="text" name="userName">
     ID#: <input type="text" name="userID">
     <input type="submit">
  </form>
</body>
</html>
```

The param value "userName" comes from the name attribute of the form's input field.

We can do it even better:

We have to make sure our form input field name (which becomes the request parameter name) is the same as the property name in our bean. Then in the <jsp:setProperty> tag, we don't have to specify the param attribute.
If we name the property but don't specify a value or param, we're telling the Container to get the value from a request parameter with a matching name. If we change the HTML so that the input field name matches the property name:

```
<html>
<body>
<form action="TestBean.jsp">
    name: <input type="text" name="name">
    ID#: <input type="text" name="empID">
   <input type="submit">
</form>
</body>
</html>
```



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.

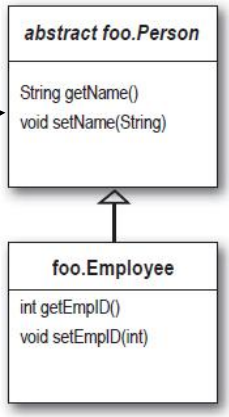
We get to do this:

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

It gets even BETTER ...

If we make all the request parameter names match the bean property names. The person bean, which is an instance of abc.Employee actually has two properties—name and empID.
Now we will change the HTML again:

```
<form action="TestBean.jsp">
  name: <input type="text" name="name">
  ID#: <input type="text" name="empID">
  …
```

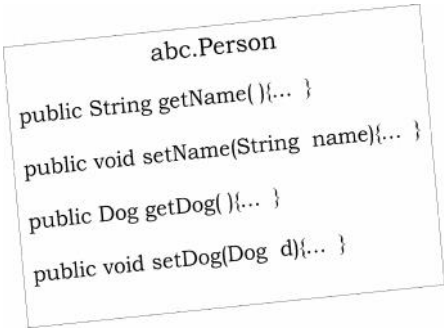


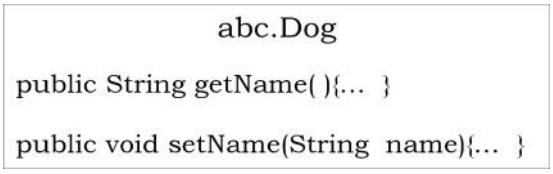
We get to do this:

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

Expression Language(EL)

Bean tags convert primitive properties automatically:
String and primitives are converted automatically.
The <jsp:setProperty> action takes the String request parameter, converts it to an int, and passes that int to the bean's setter method for that property.
What if the property is something OTHER than a String or primitive?
If one of the property is of type Object and has it's own properties:





If we want to print the name of the Person's Dog:

Servlet code:

```
abc.Person p = new abc.Person();
p.setName("mohan");
abc.Dog d = new abc.Dog();
d.setName("Spike");
p.setDog(d);
request.setAttribute("person", p);
```

Now display the property of the property:

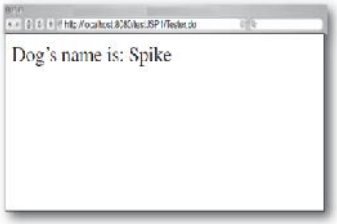
Without standard actions (using scripting) :

```
<html>
  <body>
    <%= ((abc.Person) request.getAttribute("person")).getDog().getName() %>
  </body>
</html>
```

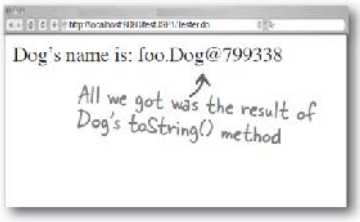
With standard actions (no scripting)

```
<html>
 <body>
   <jsp:useBean id="person" class="abc.Person" scope="request" />
   Dog's name is: <jsp:getProperty name="person" property="dog" />
 </body>
</html>
```

What we WANT



What we GOT



The JSP Expression Language (EL) was added to the JSP 2.0 spec, which helps us to save time.

JSP code without scripting, using EL :

```
<html>
  <body>
    Dog's name is: ${person.dog.name}
  </body>
</html>
```

Here this: ${person.dog.name}

Replaces this: <%= ((abc.Person) request.getAttribute("person")).getDog().getName() %>

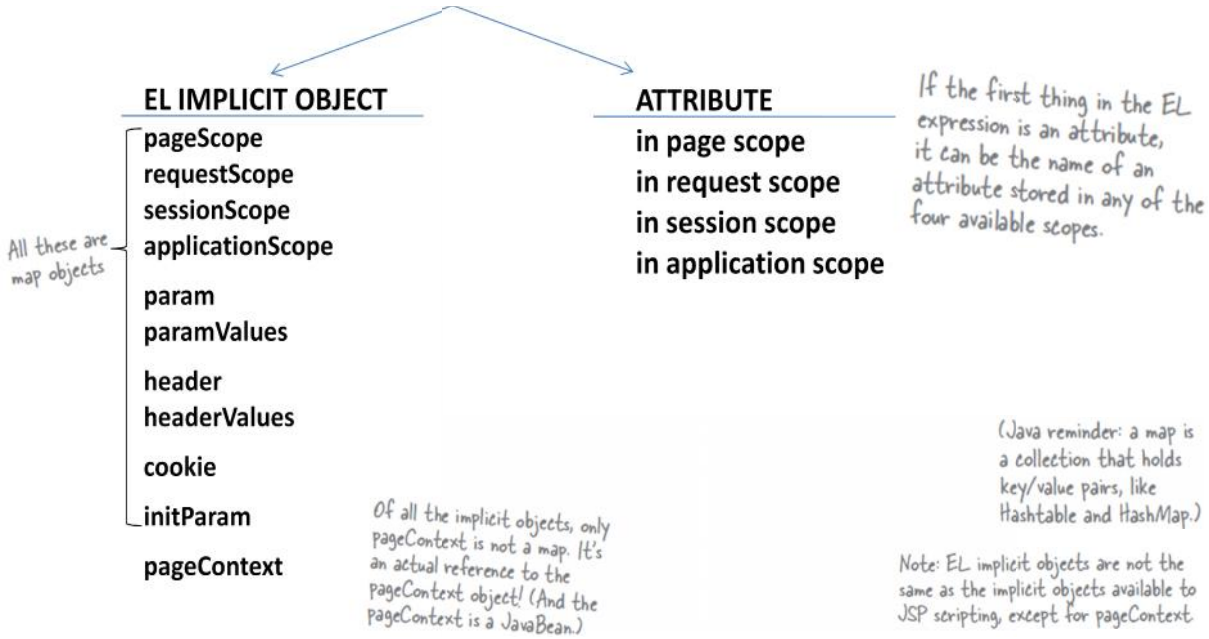
EL makes it easy to print nested properties... in other words, properties of properties!

EL expressions are always within curly braces, and prefixed with the dollar sign.

${person.name}

The first named variable in the expression is either an **implicit object** or an **attribute**.

${ **firstThing .** secondThing }



Using the dot (.) operator to access properties and map values:

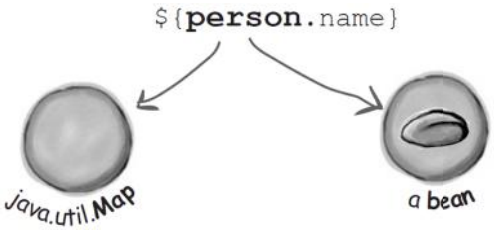
The first variable is either an implicit object or an attribute, and the thing to the right of the dot is:

Either a **map key** (if the first variable is a map)

Or a **bean property** if the first variable is an attribute that's a JavaBean.

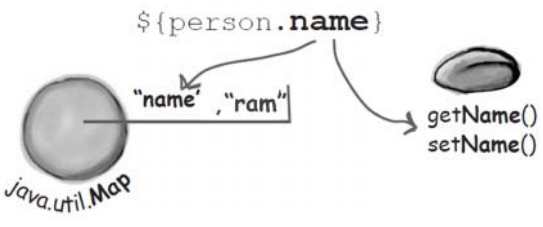
1

If the expression has a variable followed by a dot, the left-hand variable MUST be a Map or a bean.



2

The thing to the right of the dot MUST be a Map key or a bean property.



3 And the thing on the right must follow normal Java naming rules for identifiers.

${person.name}

* Must start with a letter, _, or $.
* After the first character, you can include numbers.
* Can't be a Java keyword.

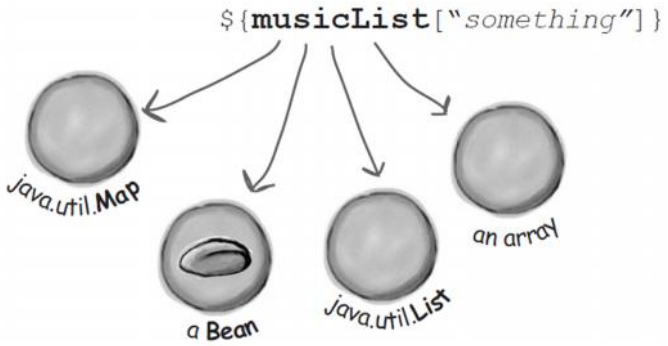
The pageContext implicit object is a bean—it has getter methods. All other implicit objects are Maps.
If the object is a bean but the named property doesn't exist, then an exception is thrown.

The [ ] operator is like the dot only way better:
The simple dot operator version works because person is a bean, and name is a property of person.
But what if person is an array? Or what if person is a List? Or what if name is something that can't be expressed with the normal Java naming rules?
But the [ ] operator is a lot more powerful and flexible...
This: ${person["name"]}
Is the same as this: ${person.name}

The [ ] gives us more options...
With the [ ], the thing on the left can also be a List or an array (of any type).
That also means the thing on the right can be a number, or anything that resolves to a number, or an identifier that doesn't fit the Java naming rules.
For example, you might have a Map key that's a String with dots in the name ("xyz.abc.trouble").

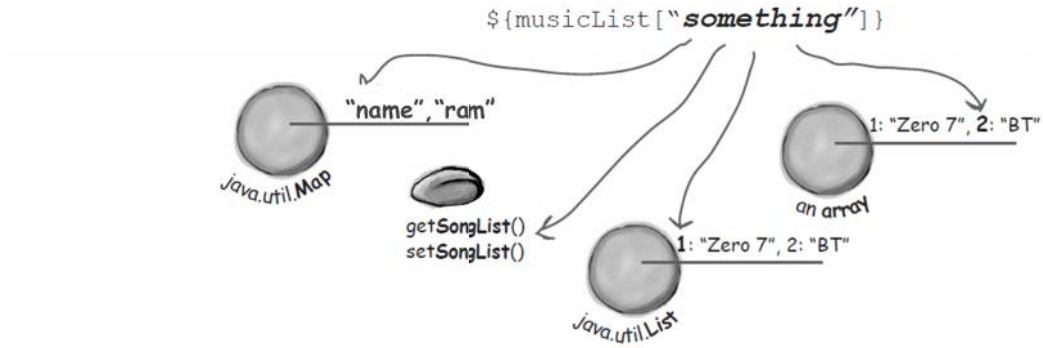
1

If the expression has a variable followed by a bracket [ ], the left-hand variable can be a Map, a bean, a List, or an array.



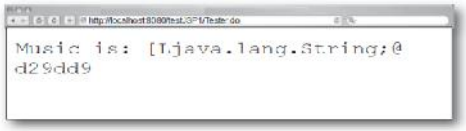
2

If the expression has a variable followed by a bracket [ ], the left-hand variable can be a Map, a bean, a List, or an array.

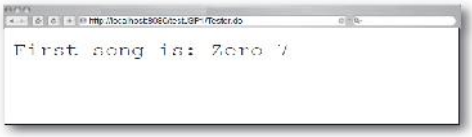


Using the [ ] operator with an array:
In a Servlet:
String[] favoriteMusic = {"Zero 7", "Tahiti 80", "BT", "Frou Frou"}; request.setAttribute("musicList", favoriteMusic);
In a JSP:

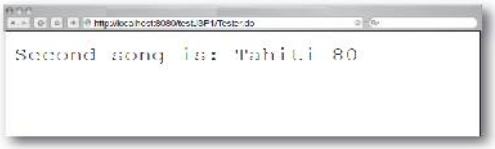
Music is: **${musicList}**



First song is: **${musicList[0]}**



Second song is: **${musicList[“1”]}**



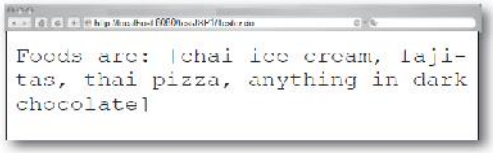
The EL for accessing an array is the same as the EL for accessing a List.
Remember EL is not Java! In EL, the [ ] operator is NOT the array access operator.
[ ] has no specific name according to the spec.

In a Servlet:

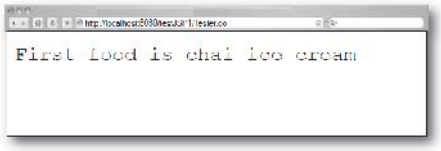
```
java.util.ArrayList favoriteFood = new java.util.ArrayList();
favoriteFood.add(“chai ice cream”);
favoriteFood.add(“fajitas”);
favoriteFood.add(“thai pizza”);
favoriteFood.add(“anything in dark chocolate”);
request.setAttribute(“favoriteFood”, favoriteFood);
```

In a JSP:

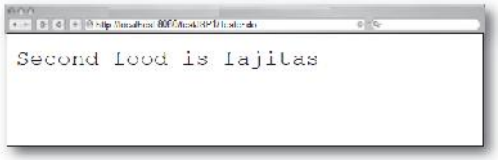
Foods are: **${favoriteFood}**



First food is **${favoriteFood[0]}**



Second food is **${favoriteFood[“1”]}**



If the thing to the left of the bracket is an array or a List, and the index is a String literal, the index is coerced to an int. This would not work:${favoriteFood[“one”]}
Because “one” can’t be turned into an int. We’ll get an error if the index can’t be coerced.

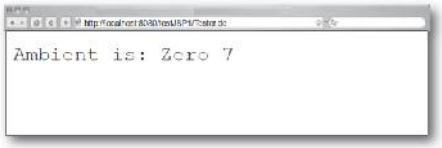
We ask for the key or property name, and we get back the value of the key or property:

In a Servlet :

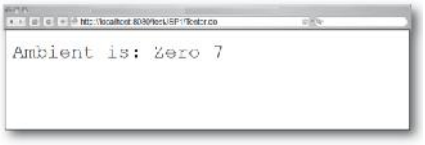
```
java.util.Map musicMap = new java.util.HashMap();
musicMap.put(“Ambient”, “Zero 7”);
musicMap.put(“Surf”, “Tahiti 80”);
musicMap.put(“DJ”, “BT”);
musicMap.put(“Indie”, “Travis”);
request.setAttribute(“musicMap”, musicMap);
```

In a JSP :

Ambient is: **${musicMap.Ambient}**



Ambient is: **${musicMap[“Ambient”]}**



String literals are evaluated:
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.
Music is: ${musicMap[Ambient]}
Find an attribute named "Ambient". Use the VALUE of that attribute as the key into the Map, or return null.

In a servlet
java.util.Map musicMap = new java.util.HashMap();
musicMap.put("Ambient", "Zero 7");
musicMap.put("Surf", "Tahiti 80");
musicMap.put("DJ", "BT");
musicMap.put("Indie", "Frou Frou");
request.setAttribute("musicMap", musicMap);
request.setAttribute("Genre", "Ambient");

This *DOES work in a JSP*

Music is ${musicMap[**Genre]}** —evaluates to→ Music is ${musicMap[**"Ambient"]}**
Because there is a request attribute named "Genre" with
a value of "Ambient", and "Ambient" is a key into musicMap.

This does *NOT work in a JSP (given the servlet code)*

Music is ${musicMap[**"Genre"]}** —doesnot change→ Music is ${musicMap[**"Genre"]}**

Because there is no key in musicMap named "Genre". With the quotes around it, the Container didn't try to evaluate it and just assumed it was a literal key name.

We can use nested expressions inside the brackets:
We nest expressions to any arbitrary level. We can put a complex expression inside a complex expression inside a... (it keeps going). And the expressions are evaluated from the inner most brackets out.

In a servlet	**This DOES work in a JSP**

java.util.Map musicMap = new java.util.HashMap();
musicMap.put("Ambient", "Zero 7");
musicMap.put("Surf", "Tahiti 80");
musicMap.put("DJ", "BT");
musicMap.put("Indie", "Frou Frou");
request.setAttribute("musicMap", musicMap);
String[] musicTypes = {"Ambient", "Surf", "DJ", "Indie"};
request.setAttribute("MusicType", musicTypes);
	Music is **${musicMap[MusicType[0]]}**
	Music is **${musicMap["Ambient"]}** becomes
	Music is **Zero 7**

You can't do ${abc.1}:

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.

This:	**But this:**
${musicMap.Ambient}	${musicList["1"]}
Is the same as this:	**Can not be turned into this:**
${musicMap["Ambient"]}	${musicList.1}

EL renders raw text, including HTML:

If, currentTip => "Some Tip"

This:	**Becomes:**
 Tip of the Day: ${pageContent.currentTip} 	 Tip of the Day: **Some Tip**

The same is true for JSP expression tags:	And for the jsp:getProperty standard action:
 Tip of the Day: <%= pageContent.getCurrentTip() %> 	 Tip of the Day: <jsp:getProperty name='pageContent' property='currentTip' />

Here the portion of the tip is being sent in the output stream, but the web browser is simply rendering it as raw HTML.

The EL implicit objects:

El implicit objects are not the same as the Jsp Implicit Objects (except for one, pageContext).

pageScope
requestScope
sessionScope
applicationScope
— A Map of the scope attributes.

param
paramValues
— Maps of the request parameters.

header
headerValues
— Maps of the request headers.

cookie
— This is a Map of cookies.

initParam
— A Map of the context init parameters (NOT servlet init parameters!)

pageContext
— An actual reference to the pageContext object, which is a bean. Look in the API for the PageContext getter methods.

Request parameters in EL:

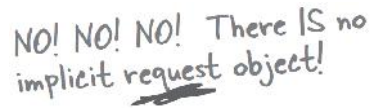
The param implicit object is fine when you know you have only one parameter for that particular parameter name. Use paramValues when you might have more than one parameter value for a given parameter name.

In the HTML form :	**In Jsp :**
<form action="TestBean.jsp"> Name: <input type="text" name="name" /> ID#: <input type="text" name="empID" /> First food: <input type="text" name="food" /> Second food: <input type="text" name="food" /> <input type="submit" /> </form>	Request param name is: ${param.name} Request param empID is: ${param.empID} Request param food is: ${param.food} First food request param: ${paramValues.food[0]} Second food request param: ${paramValues.food[1]} Request param name: ${paramValues.name[0]}

What if we want to get Host information :

Host is: ${header["host"]}

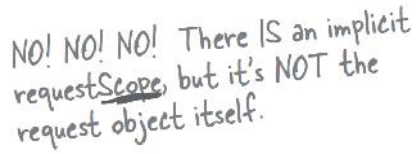
Host is: ${header.host}



This won't work:

Method is: ${request.method}

Method is: ${requestScope.method}



Then How to do it???

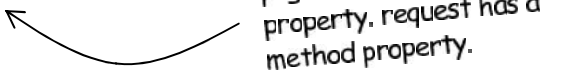
The requestScope is NOT the request object:

The implicit requestScope is just a Map of the request scope attributes, not the request object itself!

Use requestScope to get request ATTRIBUTES, not request PROPERTIES. For request properties, you need to go through pageContext. Ex.:

What we want (the HTTP method) is a property of the request object, not an attribute at request scope.

Method is: **${pageContext.request.method}**



Don't confuse the Map scope objects with the objects to which the attributes are bound.

It's so easy to think that, applicationScope is a reference to ServletContext, since that's where application-scoped attributes are bound. You can't treat it like a ServletContext, so don't expect to get ServletContext properties back from the applicationScope implicit object!

Scope implicit objects for the rescue:

If all you need is to print the name of a person, and you really don't care what scope the person is in

${**person.name}**

Or, if you're worried about a potential naming conflict, you can be explicit about which person you want:

${**requestScope.person.name}**

There is one more reason we needed scope implicit object:

request.setAttribute("person", p);

Here scoped-attribute-name is in String format, which may contain any character. It may not be following the Java-Legal-Identifier rule.

request.setAttribute("abc.person", p);

Which may cause a trouble:

${abc.person.name}

But we should be thankful to scoped-Implicit-Object:
${requestScope[“abc.person”].name}

EL operators:
The most useful EL arithmetic, relational, and logical operators.

Arithmetic (5)

Addition:	+
Subtraction:	-
Multiplication:	*
Division:	/ *and* **div**
Remainder:	% *and* **mod**

By the way... you CAN divide by zero in EL—you get INFINITY, not an error.

But you CANNOT use the Remainder operator against a zero—you'll get an exception.

Relational (6)

Equals:	== *and* **eq**
Not equals:	!= *and* **ne**
Less than:	< *and* **lt**
Greater than:	> *and* **gt**
Less than or equal to:	<= *and* **le**
Greater than or equal to:	>= *and* **ge**

Logical (3)

AND:	&& *and* **and**
OR:	|| *and* **or**
NOT:	! *and* **not**

Note: Don’t use EL reserved words as identifiers!

There are few more EL reserved words:

true	*a boolean literal*
false	*the OTHER boolean literal*
null	*It means... null*
instanceof	*(this is reserved for “the future”)*
empty	*an operator to see if something is null or empty (eg. ${empty A}) returns true if A is null or empty (you'll see this in action a little later in the chapter)*

EL handles null values gracefully:
“It’s better to show a partial, incomplete page than to show the user an error page.”
EL is null-friendly. It handles 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”.

EL functions:

It's an easy way to write a simple EL expression that calls a static method in a plain old Java class that we write. Whatever the method returns is used in the expression. It does take some more work to configure things, but *functions* give us a lot more... functionality.

Imagine you want your JSP to roll dice:

1. Write a Java class with a public static method.

This is just a plain old Java class. The method MUST be public and static, and it can have arguments. It should (but isn't required to) have a non-void return type. After all, the whole point is to call this from a JSP and get something back that you can use as part of the expression or to print out.

Put the class file in the /WEB-INF/classes directory structure (matching the appropriate package directory structure, just like you would with any other class).

2. Write a Tag Library Descriptor (TLD) file:

For an EL function, the TLD provides a mapping between the Java class that defines the function and the JSP that calls the function.

That way, the function name and the actual method name can be different.

The TLD says, "This is the Java class, this is the method signature for the function (including return type) and this is the name we'll use in EL expressions". In other words, the name used in EL doesn't have to be the same as the actual method name, and the TLD is where you map that.

Put the TLD file inside the /WEB-INF directory. Name it with a .tld extension.

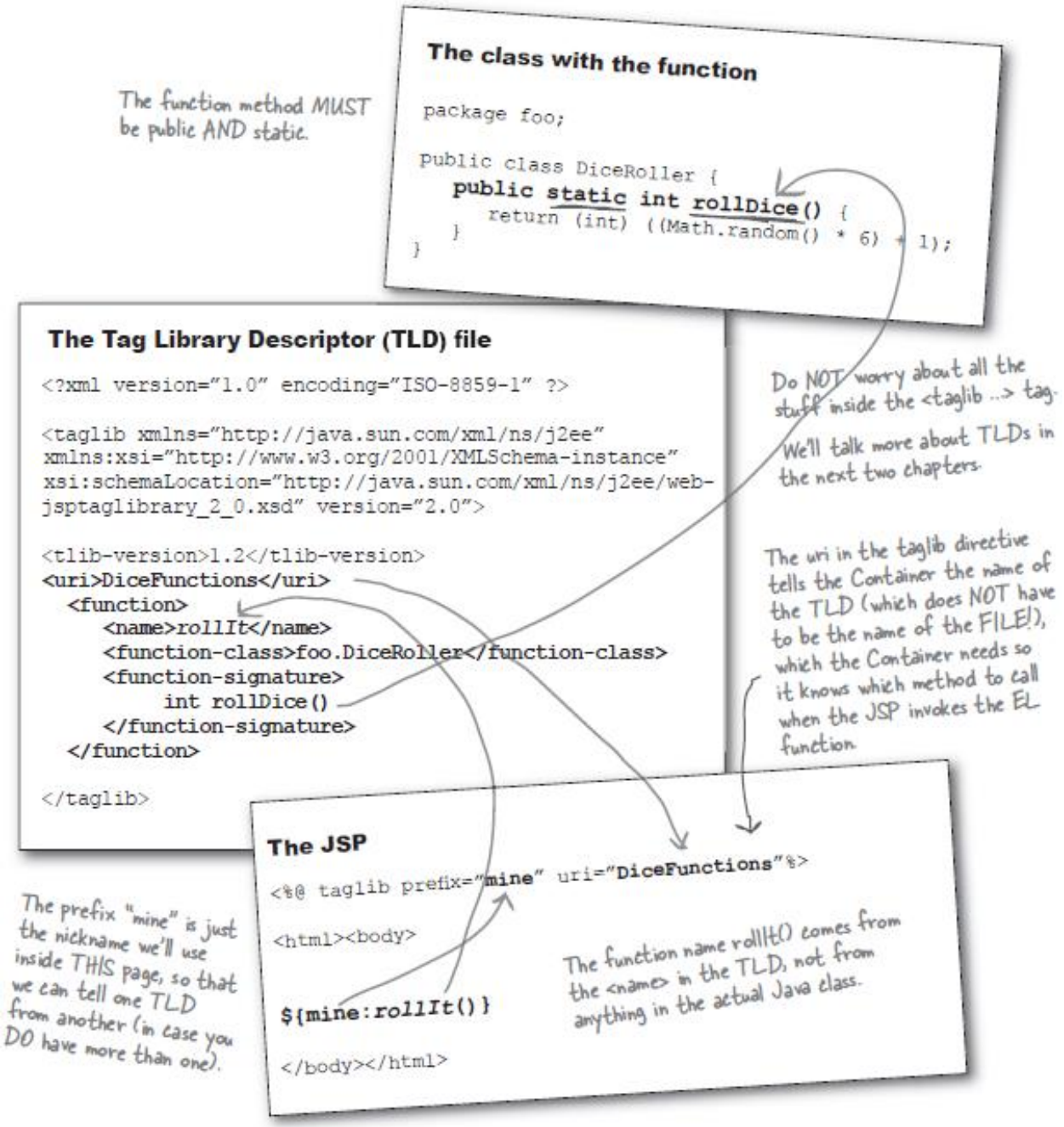
3.Put a taglib directive in your JSP:

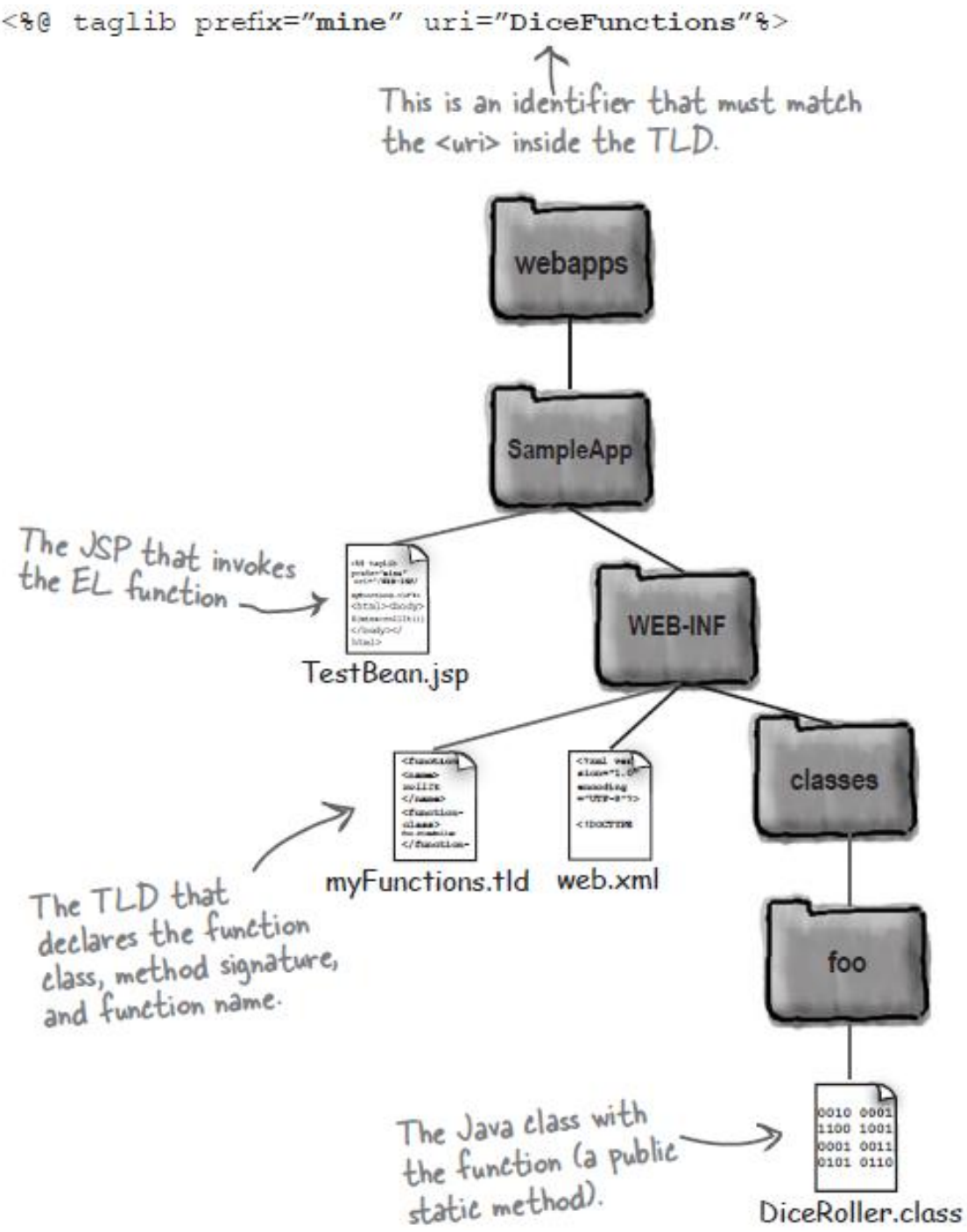
The taglib directive tells the Container, "I'm going to use this TLD, and in the JSP, when we want to use a function from this TLD, we are going to prefix it with this name..." In other words, it lets you define the namespace.

You can use functions from more than one TLD, and even if the functions have the same name, that's OK. The taglib directive is kind of like giving all your functions fully-qualified names. You invoke the function by giving both the function name AND the TLD prefix. The prefix can be anything you like.

4.Use EL to invoke the function:

This is the easy part. You just call the function from an expression using ${prefix:name()}.

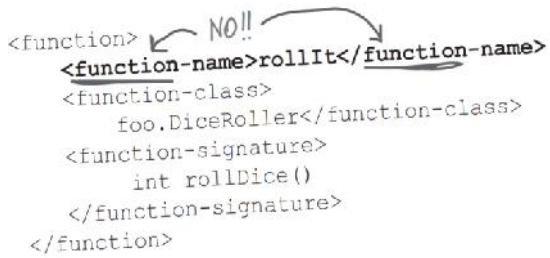

The class with the function
package foo;
Public class DiceRoller {
public static int rollDice() {
return (int) ((Math.random() * 6) + 1);
}
}
The function method MUST be public AND static.
The Tag Library Descriptor (TLD) file
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" version="2.0">
<tlib-version>1.2</tlib-version>
<uri>DiceFunctions</uri>
<function>
<name>rollIt</name>
<function-class>foo.DiceRoller</function-class>
<function-signature>
int rollDice()
</function-signature>
</function>
</taglib>
Do NOT worry about all the stuff inside the <taglib ...> tag.
We'll talk more about TLDs in the next two chapters.
The uri in the taglib directive tells the Container the name of the TLD (which does NOT have to be the name of the FILE!), which the Container needs so it knows which method to call when the JSP invokes the EL function.
The JSP
<%@ taglib prefix="mine" uri="DiceFunctions"%>
<html><body>
${mine:rollIt()}
</body></html>
The prefix "mine" is just the nickname we'll use inside THIS page, so that we can tell one TLD from another (in case you DO have more than one).
The function name rollIt() comes from the <name> in the TLD, not from anything in the actual Java class.



The class with the function (the public static method) must be available to the web app just like servlet, bean & listener classes. That means somewhere in WEB-INF/classes…

Put the TLD file somewhere under WEB-INF, and make sure the taglib directive in the JSP includes a uri attribute that matches the <uri> element in the TLD.

The relationships between the class, the TLD, and the JSP. Most importantly, remember that the METHOD name does NOT have to match the FUNCTION name. What you use in EL to invoke the function must match the <name> element in the <function> declaration in the TLD. The element for <function-signature> is there to tell the Container which method to call when the JSP uses the <name>. And the only place the class name appears (besides the class declaration itself) is in the <function-class> element. Did you notice that everything in the <function> tag has the word <function> in it EXCEPT for the <name> tag? So, don't be fooled by this:



Remember:
If you're calling an EL function that doesn't return anything, then you're calling it just for its side effects! Given the goal for EL is to reduce the amount of logic in a JSP (a JSP is supposed to be the VIEW!).

When the app is deployed, the Container searches through WEB-INF and its subdirectories (or in JAR files within WEB- INF/lib) looking for .tld files. When it finds one, it reads the URI and creates a map that says, "The TLD with this URI is actually this file at this location...".

In the TLD, specify the fully-qualified class name (unless it's a primitive) for each argument. A function that takes a Map would be:

<function-signature> int rollDice(java.util.Map) </function-signature>

And call it with: ${mine:rollDice(aMapAttribute)}

Reusable template pieces:
Web pages in an application contains several repeating blocks like: Header, Footer, Menu etc.
There's a mechanism for handling this in a JSP—it's called include.

We write our JSP in the usual way, except that instead of putting the reusable stuff explicitly into the JSP we're authoring, we instead tell the Container to include the other file into the existing page, at the location we select. It's kind of like saying:

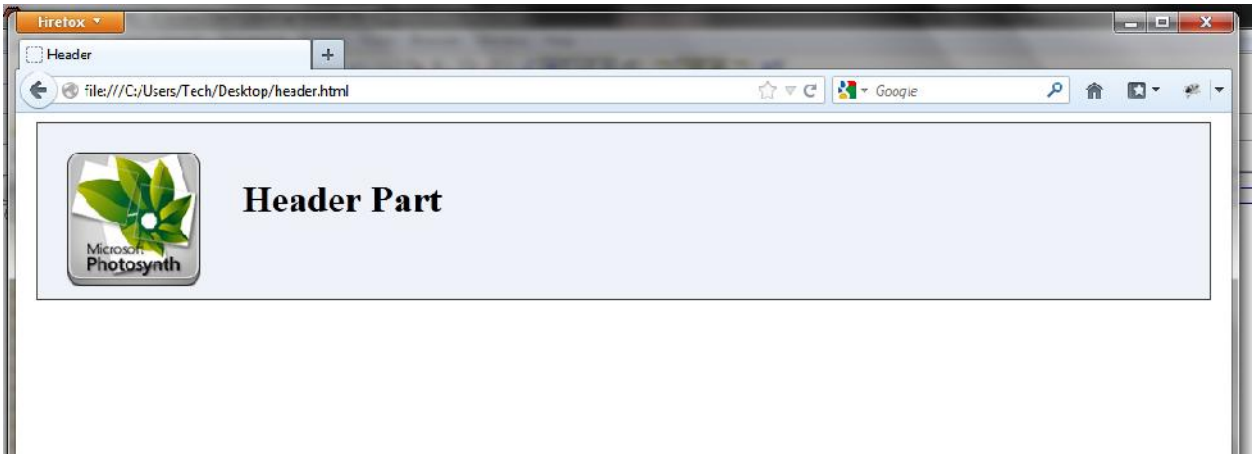
```
<html>
 <body>
   <!-- insert the header file here -->
   Welcome to our site...
   more stuff here...
   <!-- insert the footer file here -->
 </body>
</html>
```

Now we'll discuss two different include mechanisms: the include directive and the <jsp:include/> standard action.

Standard header file ("Header.jsp"):

```
<html><body>
   <img src="images/some-image.jpg" > <br>
   <em><strong>Header Part</strong></em> <br>
</body></html>
```

This is what we want on EVERY page.



The include directive:

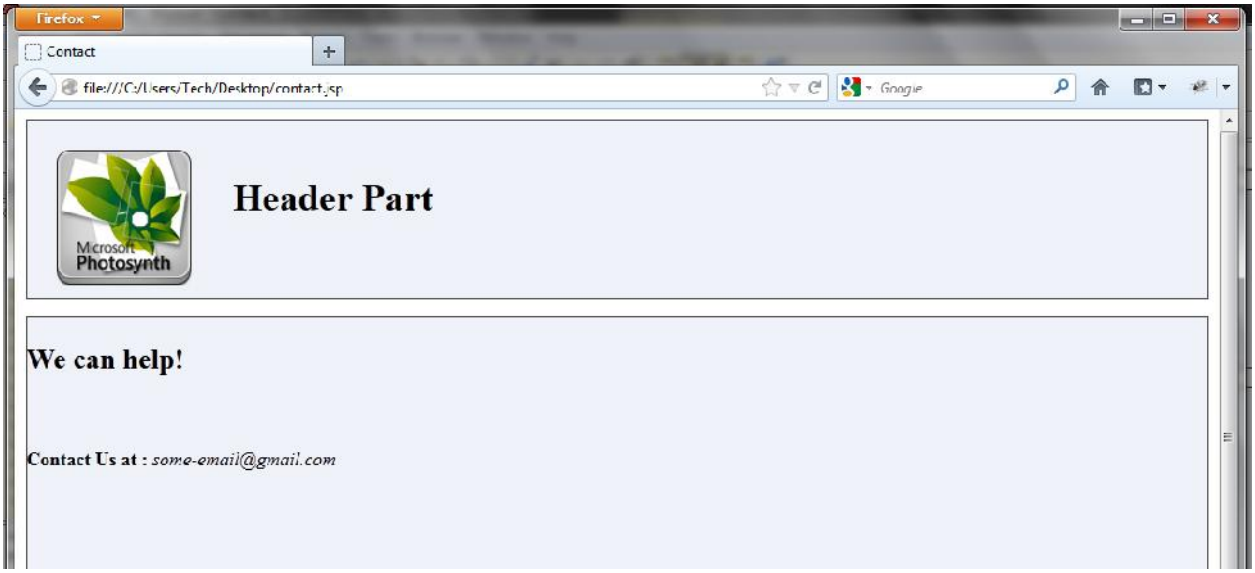
The include directive tells the Container one thing:

"copy everything in the included file and paste it into this file, right **here**..."

A JSP from the web app ("Contact.jsp")

```
<html><body>
  <%@ include file="Header.jsp"%>

 <br>
  <em>We can help.</em> <br><br>
  Contact us at: ${initParam.mainEmail}
</body></html>
```

This says "Insert the complete Header.jsp file into this point in THIS page, then keep going with the rest of this JSP..."

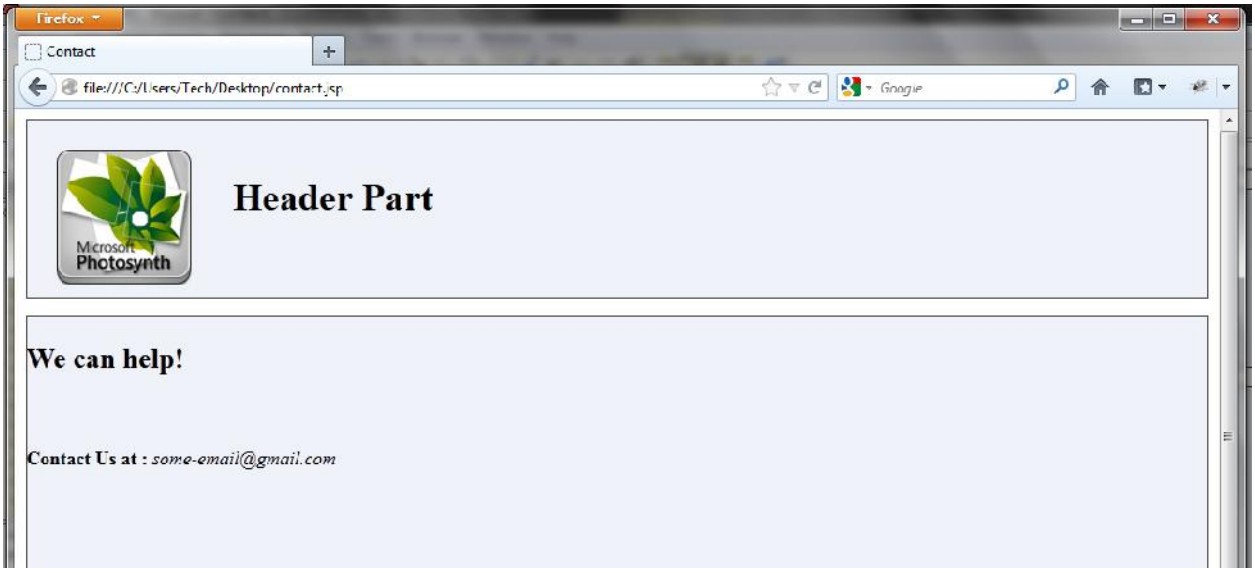


The <jsp:include> standard action:

A JSP from the web app ("Contact.jsp")

```
<html><body>
  <jsp:include page="Header.jsp" />

 <br>
  <em>We can help.</em> <br><br>
  Contact us at: ${initParam.mainEmail}
</body></html>
```

This says "Insert the complete Header.jsp file into this point in THIS page, then keep going with the rest of this JSP..."



The Difference:
With the include directive, there is NO difference between you opening your JSP page and pasting in the contents of "Header.jsp". Except the Container does it at translation time for you, so that you don't have to duplicate the code everywhere.

You can write all your pages with an include directive, and the Container will go through the trouble of copying the header code into each JSP before translating and compiling the generated servlet.

But <jsp:include> is a completely different story. Rather than copying in the source code from "Header.jsp", the include standard action inserts the response of "Header.jsp", at runtime.

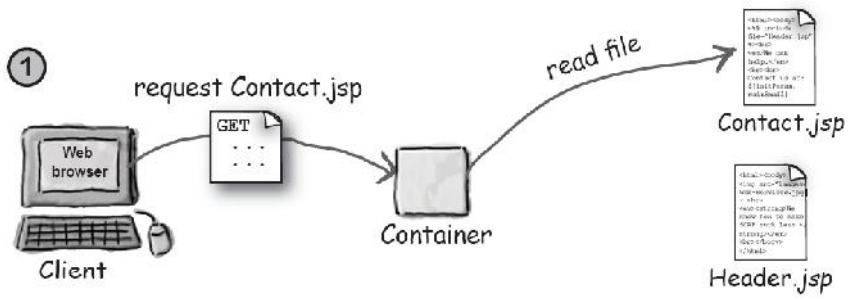
The key to <jsp:include> is that the Container is creating a RequestDispatcher from the page attribute and applying the include() method. The dispatched/included JSP executes against the same request and response objects, within the same thread.

The include directive inserts the source of "header.jsp", at translation time.

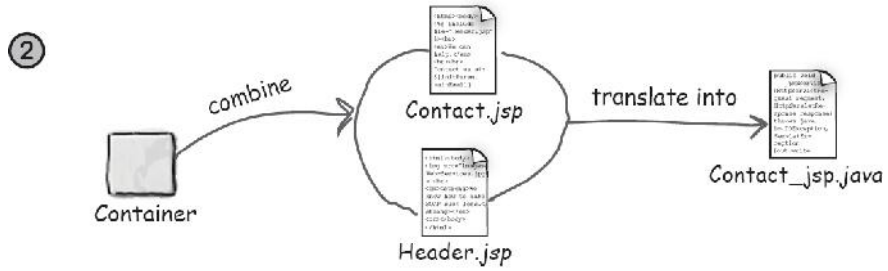
But the <jsp:include /> standard action inserts the response of "header.jsp" at runtime.

There is an extra performance hit with every <jsp:include>. With the directive, on the other hand, the hit happens only once when the including page is translated. So if you are pretty sure that once you go to production, the directive might be the way to go. The spec. doesn't force Containers to check any changes made in any jsp page. So to make your web-app compatible with other servers implementations, you need <jsp:include />.

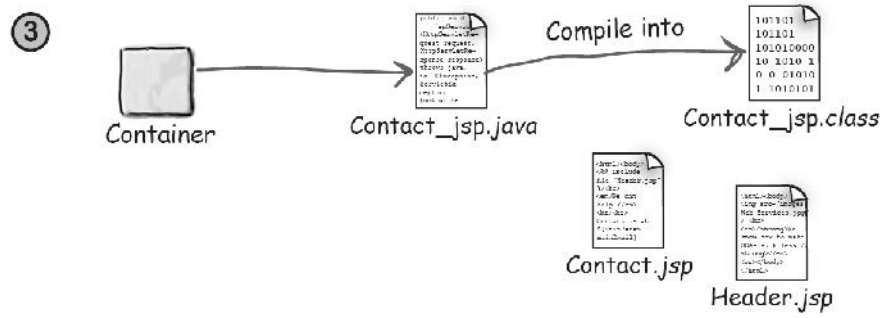
The include directive at first request:
With the include directive, the Container has a lot of work to do, but only on the first request. From the second request on, there's no extra runtime overhead.



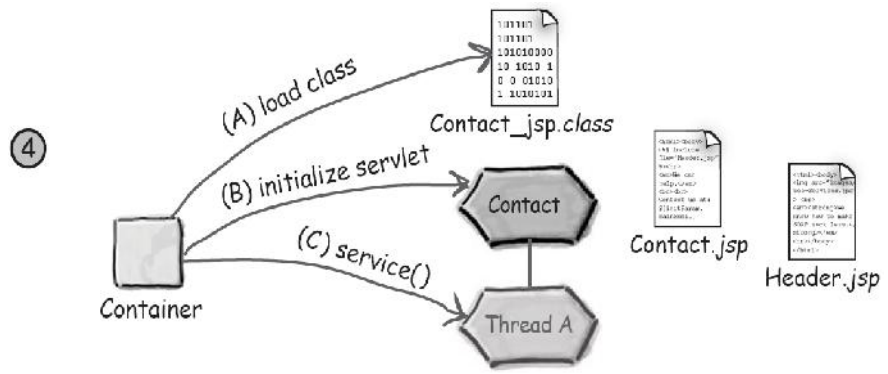
The client makes a request for Contact.jsp, which has not been translated. The Container reads the Contact.jsp page to start the translation process.



The container sees the include directive, and combines the source code of Header.jsp and Contact.jsp, and creates/translates that into a Java source file for the generated servlet.



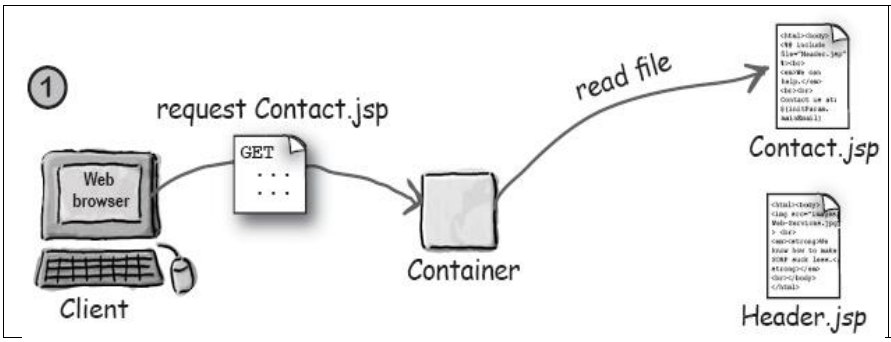
The Container compiles the translated source file into a servlet class. It's just like any other servlet at this point, and the previous step never has to happen again, unless Contact.jsp changes (or, if your Container is smart and can tell that the included *Header*.jsp has changed).



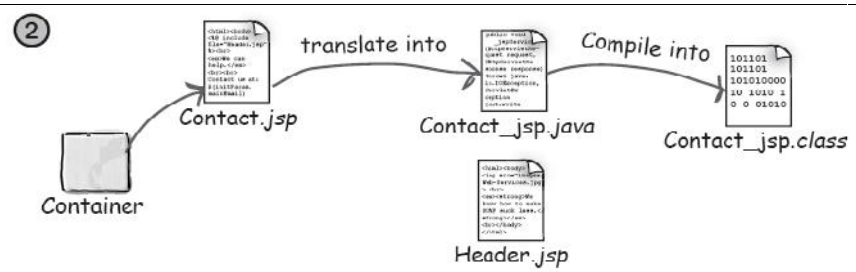
To complete the request, the Container loads the newly-compiled class, initializes a servlet (instantiates the servlet then calls init() on the new object), allocates a thread for the request, and calls the _jspService() method. From the second request on, the Container does only step (C): allocates a thread and calls the _jspService() method.

The <jsp:include> standard action at first request:

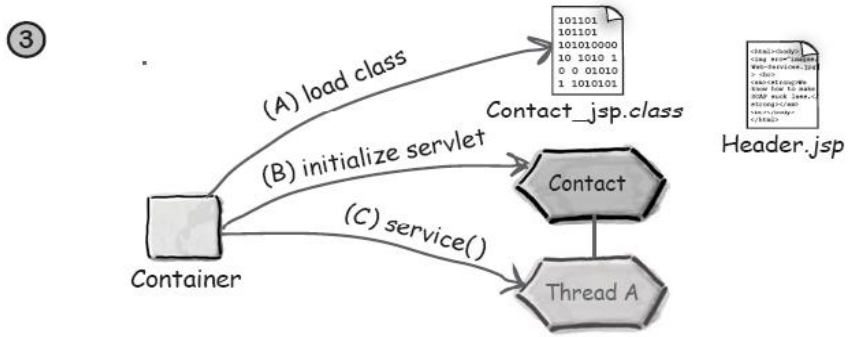
With the include standard action, there is less work at translation time, and more work with each request, especially if the included file is a JSP.



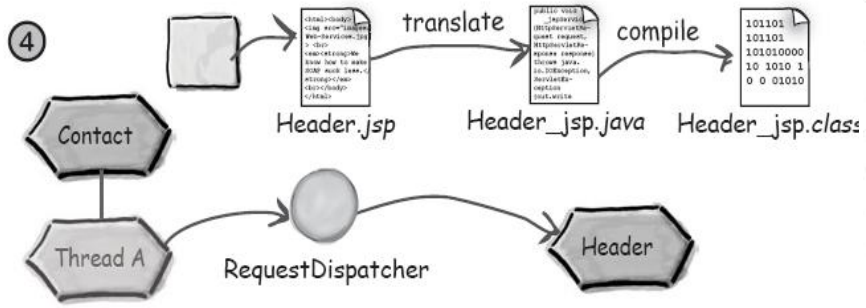
The client makes a request for Contact.jsp, which has not been translated. The Container reads the Contact.jsp page to start the translation process.



The container sees the include standard action, and uses that to insert a method call in the generated servlet code that—at runtime—will dynamically combine the response from Header.jsp into the response from Contact.jsp. The Container generates servlets for both JSP files. (This is not dictated by the spec, so we're showing only an example of how it *could* work.)



The Container compiles the translated source file into a servlet class. It's just like any other servlet at this point. The generated servlet class file is loaded into the Container's JVM and is initialized. Next, the Container allocates a thread for the request and calls the JSP's _jspService() method.



The Contact servlet hits the method that does the dynamic include, and something vendor-specific happens! All we care about is that the response generated by the Header servlet is combined with the response from the Contact servlet (at the appropriate place). (NOT SHOWN: at some point the Header.jsp is translated and compiled, then the generated servlet class is loaded and initialized.)

Note:

1. The directive attribute is file but the standard action attribute is page!

<%@ include file="Header.jsp" %>

<jsp:include page="Header.jsp" />

The include directive is used only at translation time (as with all directives). And when translating, the Container cares only about files—.jsp to .java, and .java to .class. But the <jsp:include /> standard action, as with all standard actions, is executed at request time, when the Container cares about pages to be executed.

2. We can put the page directive anywhere in the page, although by convention most people put page directives at the top. But it really matters where we put the include directive, because it tells the Container exactly WHERE to insert the source from the included file!

3. The included page can be a static(.html) page or a dynamic(.jsp) page.

If it is a dynamic page then There are a few limitations:

An included page CANNOT change the response status code or set headers (which means it can't call, say, addCookies()).

You won't get an error if the included JSP tries to do things it can't—you just won't get what you asked for.

4. Don't put opening and closing HTML and Body tags within your reusable pieces! Design and write your layout template chunks (like headers, navigation bars etc.) assuming they will be included in some other page.

Customizing the included content with <jsp:param>:

What if we want to customize part of the header?

Suppose, a context-sensitive subtitle that's part of the header, but that changes depending on the page?

We have few options: Put the subtitle information into the main page, as, say, the first thing in our page after the include for the header. Or, pass the subtitle information as a new request parameter to the included page!

JSP that does the include:

Look... no closing slash!

```
<html><body>
 <jsp:include  page="Header.jsp" >
  <jsp:param  name="subTitle"  value="Header Title" />
 </jsp:include>
 <br>
 <em>You need help! Contact Us:</em> <br><br>
 Contact us at: ${initParam.mainEmail}
</body></html>
```

<jsp:include> can have a BODY, so that you add (or replace) request parameters that the included thing can use.

Note: this idea of params doesn't make any sense with the include directive (which is not dynamic), so it applies ONLY to the <jsp:include> standard action.

The included header that USES the new param ("Header.jsp"):

```
<img src="images/Some-image.jpg" > <br>
<em><strong>${param.subTitle}</strong></em> <br>
```

To the included file, the param set with <jsp:param> is just like any OTHER request parameter. Here we're using EL to get it.

The <jsp:forward> standard action:

What if I wanted to forward from one JSP to another? If the client gets to my page and hasn't logged in, I want to send him to a different page...

We can forward from one JSP to another. Or from one JSP to a servlet. Or from one JSP to any other resource in our web app.

Note: We don't usually want to do this in production, because if you're using MVC, the View is supposed to be the View! And the View has no business doing control logic. In other words, it shouldn't be the View's job to figure out if the guy is logged in or not—someone else should have made that decision (the Controller), before deciding to forward to the View.

A conditional forward...

We want to first check that the userName parameter isn't null. If it's not, no problem—finish the response.

But if the userName parameter is null, we want to stop right here and turn the whole request over to something else—like a different JSP that will ask for the userName.

JSP with a conditional forward (Hello.jsp)

```
<html><body>
  Welcome to our page!
  <% if (request.getParameter("userName") == null) { %>
     <jsp:forward  page="HandleIt.jsp" />
  <% } %>
  Hello ${param.userName}
</body></html>
```

Try an example to find out how it runs!

With <jsp:forward>, the buffer is cleared BEFORE the forward:
When a forward happens, the resource to which the request is forwarded starts with a clear response buffer! In other words, anything written to the response before the forward happens is thrown out.

Using JSTL:

Before we can use JSTL, we need to put two files, "jstl.jar" & "standard.jar" into the WEB-INF/lib directory of our web app.
We also need to use taglib directive before we start using JSTL tags in our Jsp.

```
<%@ taglib  prefix="c"  uri="http://java.sun.com/jsp/jstl/core"  %>
```

JSTL is used in combination with EL.
Use the <c:out /> tag:

ServletCode:

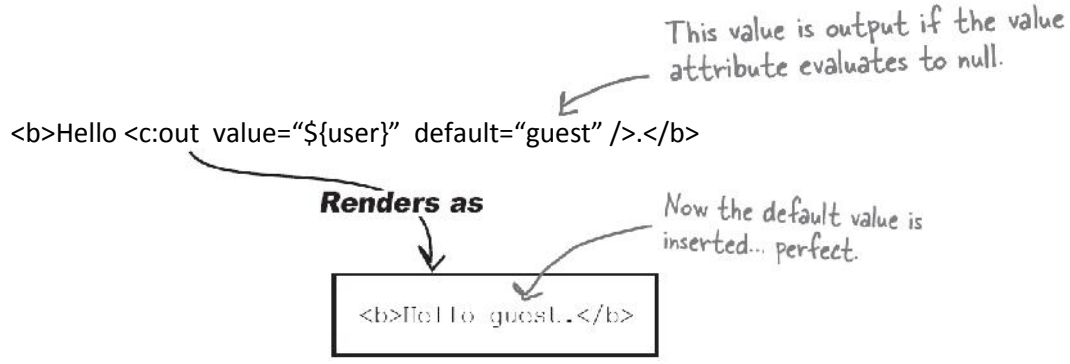
```
User u = new User();
request.setAttribute("user", u);
```

Jsp Code:

```
<c:out  value="${user.name}" />
```

Tag <c:out /> simply works like scripting-expression or EL, means simply prints the output into the response.

<c:out> provides a default attribute:
Add a default attribute, and provide the value you want to print if your expression evaluates to null:

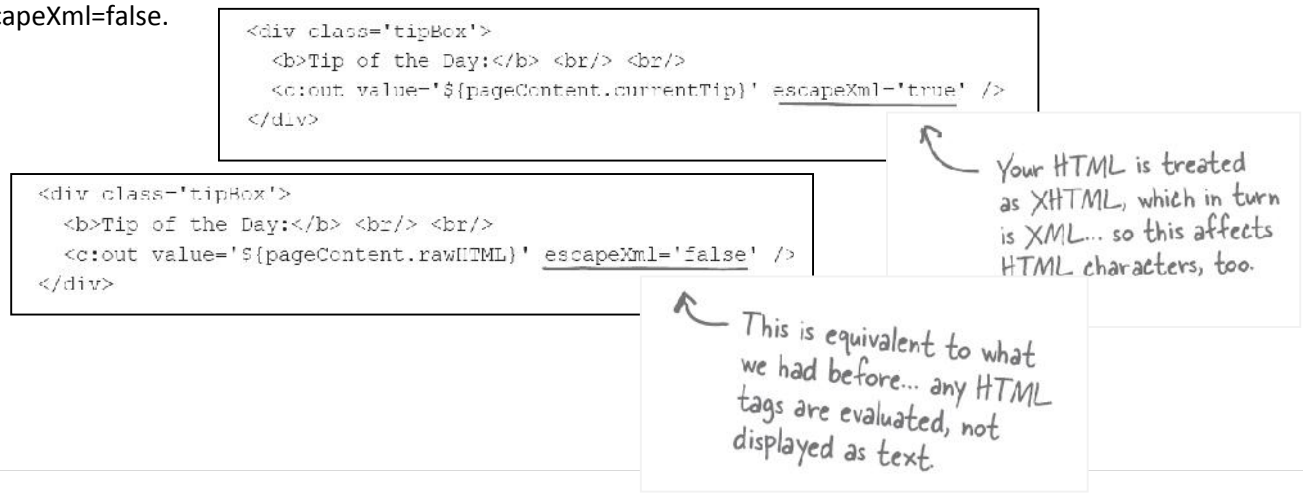


Or, you can do it this way:

```
<b>Hello <c:out value="${user}">guest</c:out></b>
```

<c:out> provides an escapeXml attribute:

If an expression, passed to value attribute, result to some html code, then the html code will further be parsed, and rendered. But if you want to use the html code literally then the attribute escapeXml of c:out tag is for the rescue. By default the escapeXml is set to true. If you want to override the default behavior then you need to set escapeXml=false.



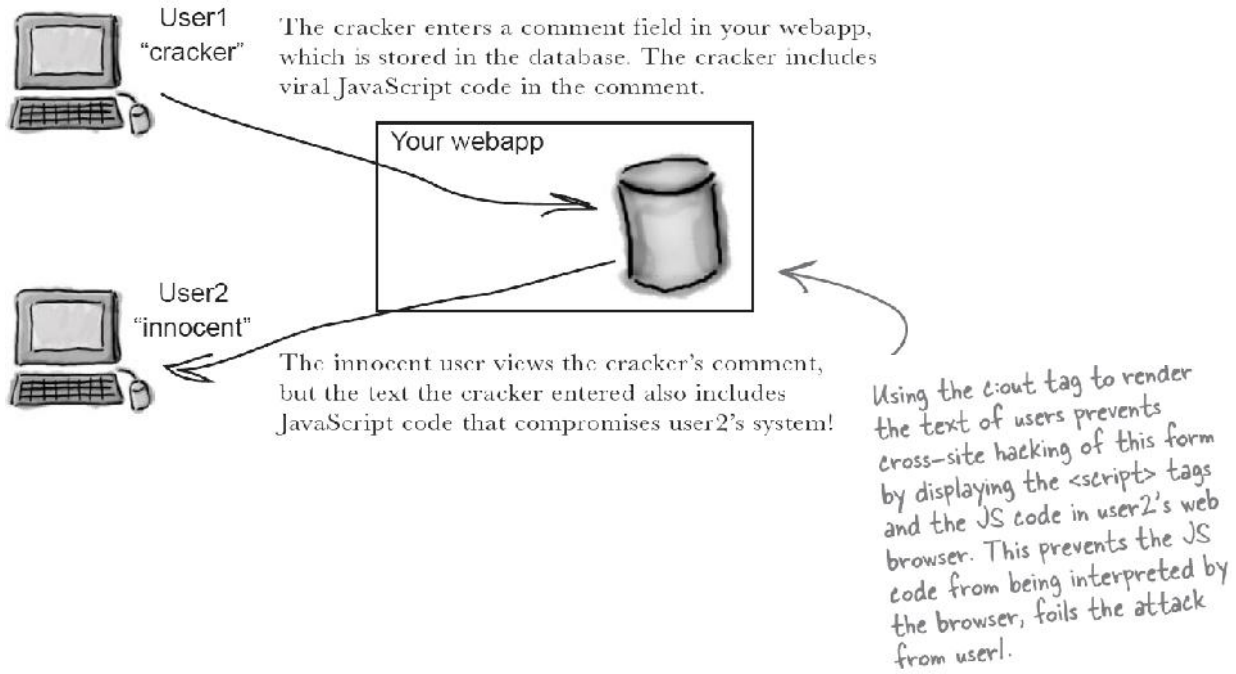
```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <c:out value='${pageContent.currentTip}' />
</div>
```

This is actually identical in functionality to this.

There are only five characters that require escaping:
<, >, &, and the two quote symbols,single and double ". All of these are converted into the equivalent HTML entities. For example, < becomes <, & becomes &, and so on.

Character	Character Entity Code
<	<
>	>
&	&
'	'
"	"

Using EL every where is discouraged. Because it may cause a serious security risk called: **cross-site hacking or cross-site scripting.** It is preferable and very secure to call EL along with the JSTL.
The attack is sent from one user to another user's web browser using your webapp as the delivery mechanism:



Looping without scripting:
Suppose you have collection of values and we want to iterate over it to pick one by one and populate table records or pull-down menu. We know we can't hard code the values!
We use "collection" here to mean either an array or Collection or Map or comma-delimited String.

Servlet code

```
...
String[] movieList = {"Amelie", "Return of the King", "Mean Girls"};
request.setAttribute("movieList", movieList);
```

Inside a Jsp Code:

```
<table>
  <c:forEach var="movie" items="${movieList}" >
    <tr>
      <td>${movie}</td>
    </tr>
  </c:forEach>
</table>
```

Loops through the entire array (the "movieList" attribute) and prints each element in a new row. (This table has just one column per row.)

The <c:forEach> tag maps nicely into a for loop—the tag repeats the body of the tag for each element in the collection.

The <c:forEach> tag

The variable that holds each ELEMENT in the collection. Its value changes with each iteration.

```
<c:forEach var="movie" items="${movieList}" >
    ${movie}
</c:forEach>
```

The actual thing to loop over (array, Collection, Map, or a comma-delimited String).

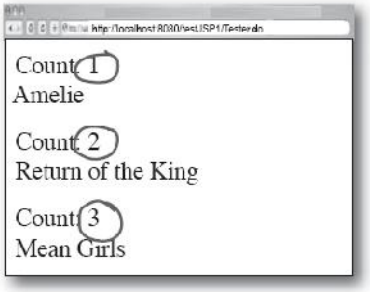
```
String[] items = (String[]) request.getAttribute("movieList");
for (int i = 0; i < items.length; i++) {
    String movie = items[i];
    out.println(movie);
}
```

Getting a loop counter with the optional varStatus attribute

varStatus makes a new variable that holds an instance of javax.servlet.jsp.jstl.core.LoopTagStatus.

```
<table>
  <c:forEach var="movie" items="${movieList}" varStatus="movieLoopCount" >
    <tr>
      <td>Count: ${movieLoopCount.count}</td>
    </tr>
    <tr>
      <td>${movie} <br><br></td>
    </tr>
  </c:forEach>
</table>
```

Helpfully, the LoopTagStatus class has a count property that gives you the current value of the iteration counter. (Like the "i" in a for loop.)



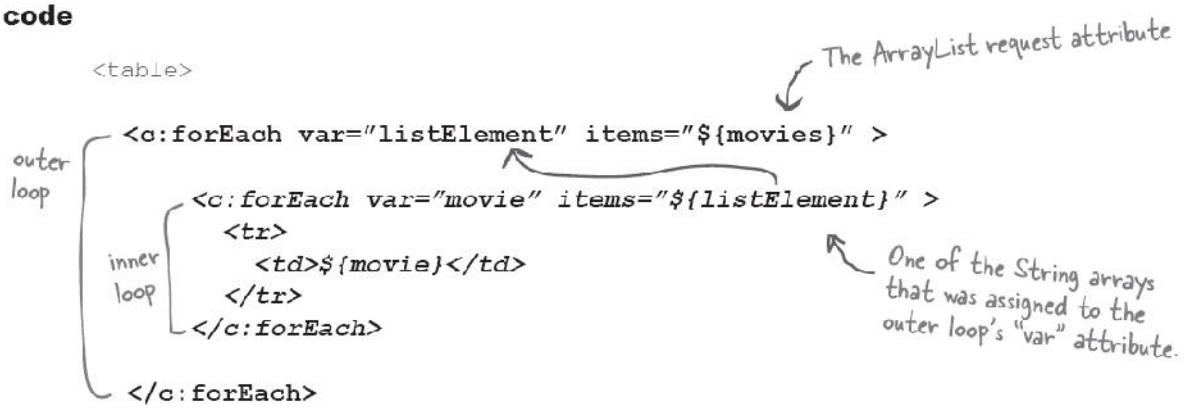
We can even nest <c:forEach>:

What if we have something like a collection of collections? An array of arrays? We can nest <c:forEach> tags for more complex scenario.

Servlet code

String[ ] movies1 = {“Matrix”, “Kill Bill”, “Boondock”};
String[ ] movies2 = {“Amelie”, “The King”, “Joe”};
java.util.List movieList = new java.util.ArrayList();
movieList.add(movies1);
movieList.add(movies2);
request.setAttribute(“movies”, movieList);

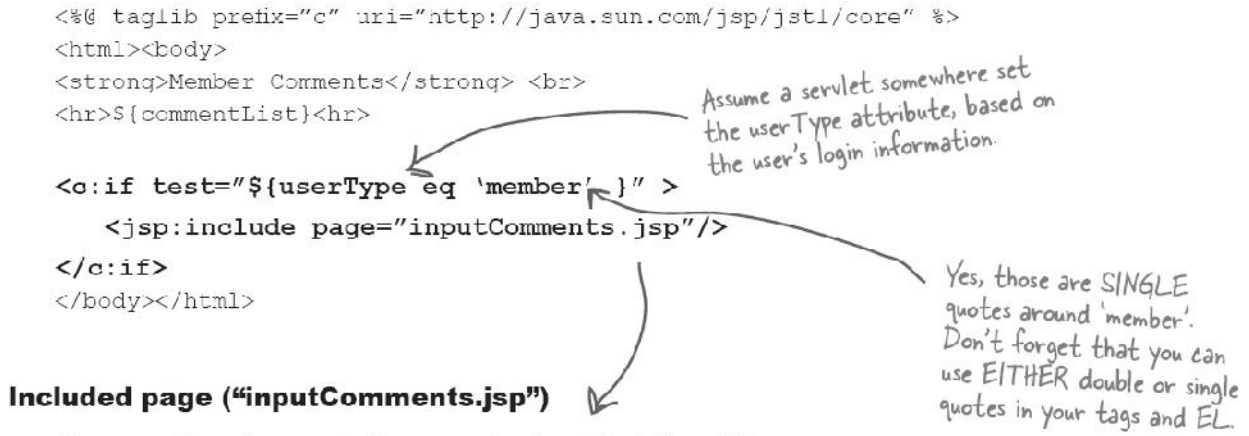
JSP code



Conditional Behavior:

If you want to perform one thing when certain condition is true, then you need to wrap such conditional code inside <c:if> tag:

JSP code



Included page (“inputComments.jsp”)

```
<form action="commentsProcess.jsp" method="post">
Add your comment: <br>
<textarea name="input" cols="40" rows="10"></textarea> <br>
<input name="commentSubmit" type="button" value="Add Comment">
</form>
```

The <c:choose> tag and its partners <c:when> and <c:other wise>:

```
<c:choose>
   <c:when test="${user == 'mohan'}">
         <!-- Mohan specific code part. -->
   </c:when>
   <c:when test="${user == 'sohan'}">
         <!-- Sohan specific code part. -->
   </c:when>
   <c:when test="${user == 'rahul'}">
         <!-- Rahul specific code part. -->
   </c:when>
   <c:otherwise>
         <!--Guest specific code. -->
   </c:otherwise>
</c:choose>
```

No more than one of these four bodies (including the <c:otherwise>) will run.

If none of the <c:when> tests are true, the <c:otherwise> runs as a default.

The <c:set />:

The <jsp:setProperty> tag can set the property of a bean.

The <c:set /> tag helps to do more: 1. Helps to set a value in a Map. 2. Helps to make a new entry in a Map? 3. Helps to create a new request-scoped attribute?	Set comes in two flavors: **var** and **target**. The var version is for setting attribute variables. The target version is for setting bean properties or Map values. Each of the two flavors comes in two variations: with or without a body. The <c:set> body is just another way to put in the value.

Setting an attribute variable *var* with <c:set>

① With NO body

If there's NOT a session-scoped attribute named "userLevel", this tag creates one (assuming the value attribute is not null).

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

The scope is optional; var is required. You MUST specify a value, but you have a choice between putting in a value attribute or putting the value in the tag body

value doesn't have to be a String...

```
<c:set var="Fido" value="${person.dog}" />
```

If ${person.dog} evaluates to a Dog object, then "Fido" is of type Dog.

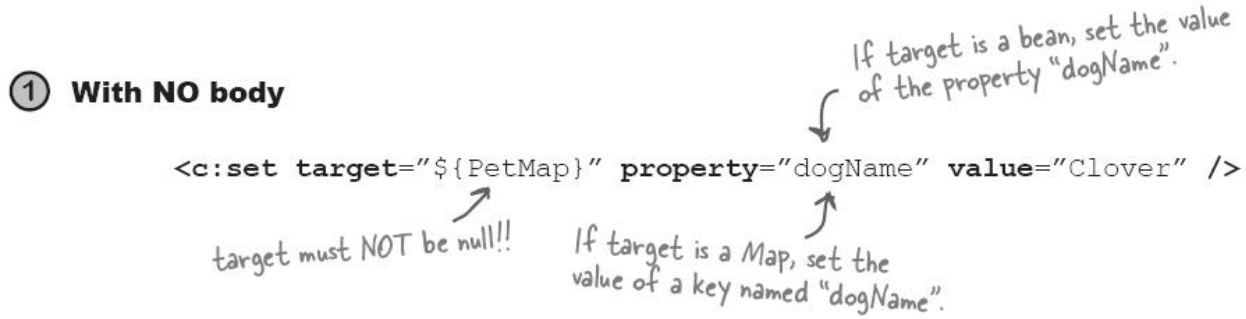
② WITH a body

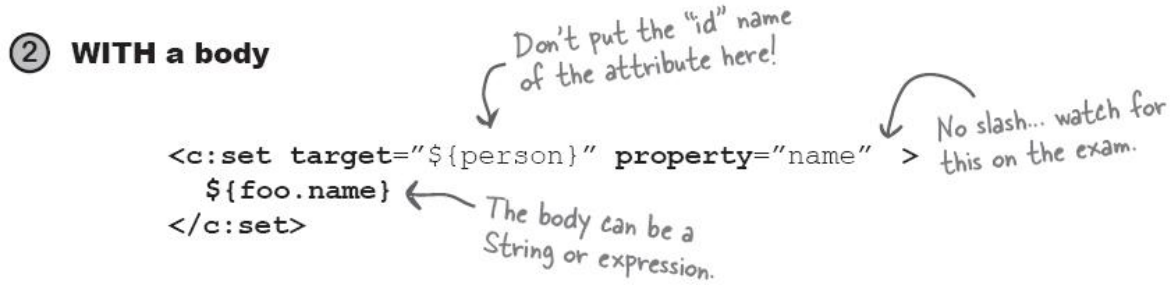
```
<c:set var="userLevel" scope="session" >
        Admin, Manager, Moderator
</c:set>
```

The body is evaluated and used as the value of the variable.

If the value evaluates to null, the variable will be **removed**!
In previous example if ${person.dog} evaluates to null (meaning there is no person, or person's dog property is null, then if there IS a variable attribute with a name "Fido", that attribute will be removed!
If you don't specify a scope, it will start looking at page, then request, etc.

Using <c:set> with beans and Maps:
This flavor of <c:set> (with its two variations—with and without a body) works for only two things: bean properties and Map values.
We can't use it to add things to lists or arrays. It's simple—
we give it the object (a bean or Map), the property/key name, and the value.





The "target" must evaluate to the object!
In the <c:set> tag, the "target" attribute in the tag seems like it should work just like "id" in the <jsp:useBean>. Even the "var" attribute in the other version of <c:set> takes a String literal that represents the name of the scoped attribute. But, it doesn't work this way with "target"! 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 or a scripting expression (<%= %>).

Rules regarding <c:set />:
1. You can never have BOTH the "var" and "target" attributes in a <c:set>.
2. "Scope" is optional, but if you don't use it the default is *page scope.*
3. If the "value" is null, the attribute named by "var" will be removed!
4. If the attribute named by "var" does not exist, it'll be created, but only if "value" is not null.
5. If the "target" expression is null, the Container throws an exception.
6. 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.*
7. If the "target" expression is not a Map or a bean, the Container throws an exception.
8. If the "target" expression is a bean, but the bean does not have a property that matches "property", the Container throws an exception. Remember that the EL expression ${bean.notAProperty} will also throw an exception.

If you don't use the optional "scope" attribute in the tag, then the tag will only look in the page scope space. It means you will just have to know exactly which scope you are dealing with.
Using <c:remove> tag:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>

   <c:set var="userStatus" scope="request" value="Brilliant" />

   userStatus: ${userStatus} <br>

   <c:remove var="userStatus" scope="request" />

   userStatus is now: ${userStatus}

</body></html>
```

The var attribute MUST be a String literal! It can't be an expression!!

The scope is optional, but if you leave it out then the attribute is removed from ALL scopes.

Using <c:import>:

① The include *directive*

```
<%@ include file="Header.html" %>
```

Static: adds the content from the value of the *file* attribute to the current page at ***translation*** time.

② The <jsp:include> *standard action*

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

Dynamic: adds the content from the value of the *page* attribute to the current page at ***request*** time.

Unlike the other two includes, the <c:import> url can be from outside the web Container!

③ The <c:import> JSTL tag

```
<c:import url="http://www.google.com/pkj/index.html" />
```

Dynamic: adds the content from the value of the *URL* attribute to the current page, at ***request*** time. It works a lot like <jsp:include>, but it's more powerful and flexible.

The directive was originally intended for static layout templates, like HTML headers. In other words, a "file".
The <jsp:include> was intended more for dynamic content coming from JSPs, so they named the attribute "page" to reflect that.
The attribute for <c:import> is named for exactly what you give it—a url!
The first two "includes" can't go outside the current Container, but <c:import> can.

<c:param> tag:

Jsp page where we will insert the import -

```
<%@ taglib prefi x=”c” uri=”http://java.sun.com/jsp/jstl/core” %>
...
<c:import url=”Header.jsp” >
    <c:param name=”subTitle” value=”You are in Home Page.” />
</c:import>
…
```

Header.jsp Which we want to include:

```
<img src=”images/Web-Image.jpg” > <br>
<h1>${param.subTitle}</h1>
```

<c:url> for hyperlink encoding:

The Container will, automatically, fall back to URL rewriting if it doesn't get a cookie from the client. It generally happens when cookies disabled at client side. But with servlets, you still have to tell the Container to "append the jsessionid to the end of this particular URL..." for each URL where it matters. We can do it from a JSP, using the <c:url> tag.

URL rewriting from a servlet:

```
<a  href='<%= response.encodeURL("service.do") %>' >click</a>
```

Add the extra session ID info to this URL.

URL rewriting from a JSP:

```
<a  href="<c:url value='/inputComments.jsp' />">Click here</a>
```

This adds the jsessionid to the end of the "value" relative URL (if cookies are disabled).

If we're a web developer, you need to know that URLs often need to be encoded. URL encoding means replacing the unsafe/reserved characters with other characters, and then the whole thing is decoded again on the server side.

For example, spaces aren't allowed in a URL, but you can substitute a plus sign "+" for the space. The problem is, <c:url> does NOT automatically encode your URLs!

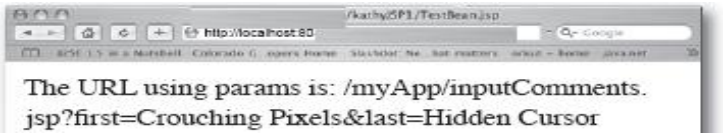
Using <c:url> with a query string

Remember, the <c:url> tag does URL rewriting, but *not* URL encoding!

```
<c:set var="last" value="Hidden Cursor" />
<c:set var="first" value="Crouching Pixels"/>

<c:url value="/inputComments.jsp?first=${first}&last=${last}" var="inputURL" />

The URL using params is: ${inputURL} <br>
```

Use the optional "var" attribute when you want access to this value later...

Output:



Using <c:param> in the body of <c:url>:
This solves our problem! Now we get both URL rewriting and URL encoding.

```
<c:url value="/inputComments.jsp" var="inputURL" >
  <c:param name="firstName" value="${first}" />
  <c:param name="lastName" value="${last}" />
</c:url>
```

Now we're safe, because <c:param> takes care of the encoding.

Now the URL looks like this:
/myApp/inputComments.jsp?firstName=Crouching+Pixels&lastName=Hidden+Cursor

Make your own error pages:
We can design a custom page to handle errors, then use the page directive to configure it.

The designated ERROR page ("errorPage.jsp") :

```
<%@ page isErrorPage="true" %>
<html><body>
  <h1>Sorry ! Some Problem Preventing you to view this page.</h1>
</body></html>
```

The BAD page that throws an exception ("badPage.jsp") :

```
<%@ page errorPage="errorPage.jsp" %>
<html><body>
   About to be bad...
   <% int x = 10/0; %>
</body></html>
```

Tells the Container, "If something goes wrong here, forward the request to errorPage.jsp".

Custom Error Pages:
To avoid showing the error page with stack trace to the client, we can make our own friendly & attractive error-response-pages.

The designated ERROR page ("errorPage.jsp")

```
<%@ page isErrorPage="true" %>
<html><body>
  <h1>Sorry for inconvenience. </h1>
  <img src="images/error-image.jpg">
</body></html>
```

Confirms for the Container, "Yes, this IS an officially-designated error page."

The BAD page that throws an exception ("badPage.jsp")

```
<%@ page errorPage="errorPage.jsp" %>
<html><body>
  About to be bad... <% int x = 10/0; %>
</body></html>
```

Tells the Container, "If something goes wrong here, forward the request to errorPage.jsp".

We can declare error pages in the DD for the entire web app, and we can even configure different error pages for different exception types, or HTTP error code types (404, 500, etc.).
The Container uses <error-page> configuration in the DD as the default, but if a JSP has an explicit errorPage page directive, the Container uses the directive.
We can declare error pages in the DD based on either the <exception-type> or the HTTP status <error-code> number. That way we can show the client different error pages specific to the type of the problem that generated the error.

Declaring a catch-all error page:

This applies to everything in our web app—not just JSPs. We can override it in individual JSPs by adding a page directive with an *errorPage attribute.*

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

If we configure more specific exception after the catch-all exception then the catch-all handler will run.

Declaring an error page based on an HTTP status code:

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

The <location> MUST be relative to the web-app root/context, which means it MUST start with a slash. (This is true regardless of whether the error page is based on <error-code> or <exception-type>.)

This configures an error page that's called only when the status code for the response is "404" (file not found).

Error pages get an extra object: exception

An error page is essentially the JSP that handles the exception, so the Container gives the page an extra object for the exception. In a scriptlet, we can use the implicit object exception, and from a JSP, we can use the EL implicit object ${pageContext.exception}. The object is type java.lang.Throwable, so in a script we can call methods, and with EL we can access the stackTrace and message properties.

A more explicit ERROR page ("errorPage.jsp"):

```
<%@ page isErrorPage="true" %>
<html><body>
  <h1>Sorry for inconvenience. </h1>
  We encountered a ${pageContext.exception} on the server.<br>
  <img src="images/error-image.jpg">
</body></html>
```

Note: the exception implicit object is available ONLY to error pages with an explicitly-defined page directive:

<%@ page isErrorPage="true" %>

In other words, configuring an error page in the DD is not enough to make the Container give that page the implicit exception object!

The <c:catch> tag:

If we have a page that invokes a risky tag, but we think we can recover, then, we can do a kind of try/catch using the <c:catch> tag, to wrap the risky tag or expression.

If we don't, and an exception is thrown, our default error handling will be invoke in and the user will get the error page declared in the DD.

The <c:catch> serves as both the try and the catch—there's no separate try tag.

We wrap the risky EL or tag calls or whatever in the body of a <c:catch>, and the exception is caught right there.

Once the exception occurs, control jumps to the end of the <c:catch> tag body.

```
<%@ taglib prefi x="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page errorPage="errorPage.jsp" %>
<html><body>
   About to do a risky thing: <br>
   <c:catch>
      <% int x = 10/0; %>
   </c:catch>
   If you see this, we survived.
</body></html>
```

In a real Java try/catch, the catch argument is the exception object.
But with web app error handling, remember, only officially-designated error pages get the exception object.
To any other page, the exception just isn't there. So this does not work:

```
<c:catch>
    Inside the catch...
    <% int x = 10/0; %>
</c:catch>
Exception was: ${pageContext.exception}
```

Won't work because this isn't an official error page, so it doesn't get the exception object.

Using the "var" attribute in <c:catch> :
Use the optional var attribute if you want to access the exception after the end of the <c:catch> tag. It puts the exception object into the page scope, under the name you declare as the value of var.

```
<%@ taglib prefi x="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page errorPage="errorPage.jsp" %>
<html><body>
  About to do a risky thing: <br>
  <c:catch var="myException">
      Inside the catch...
      <% int x = 10/0; %>
  </c:catch>
  <c:if test="${myException != null}">
      There was an exception: ${myException.message} <br>
  </c:if> We survived.
</body></html>
```

Now there's an attribute myException, and since it's a Throwable, it has a "message" property (because Throwable has a getMessage() method).

In a regular Java try/catch, once the exception occurs, the code below that point in the try block never executes—control jumps directly to the catch block.
With the <c:catch> tag, once the exception occurs, two things happen:

1) If we used the optional "var" attribute, the exception object is assigned to it.
2) Flow jumps to below the body of the <c:catch> tag.

```
<c:catch>
  Inside the catch...
  <% int x = 10/0; %>
  After the catch...
</c:catch>
We survived.
```

Control

We will never see this!

There is simply no way to use any information about the exception within the <c:catch> tag body.

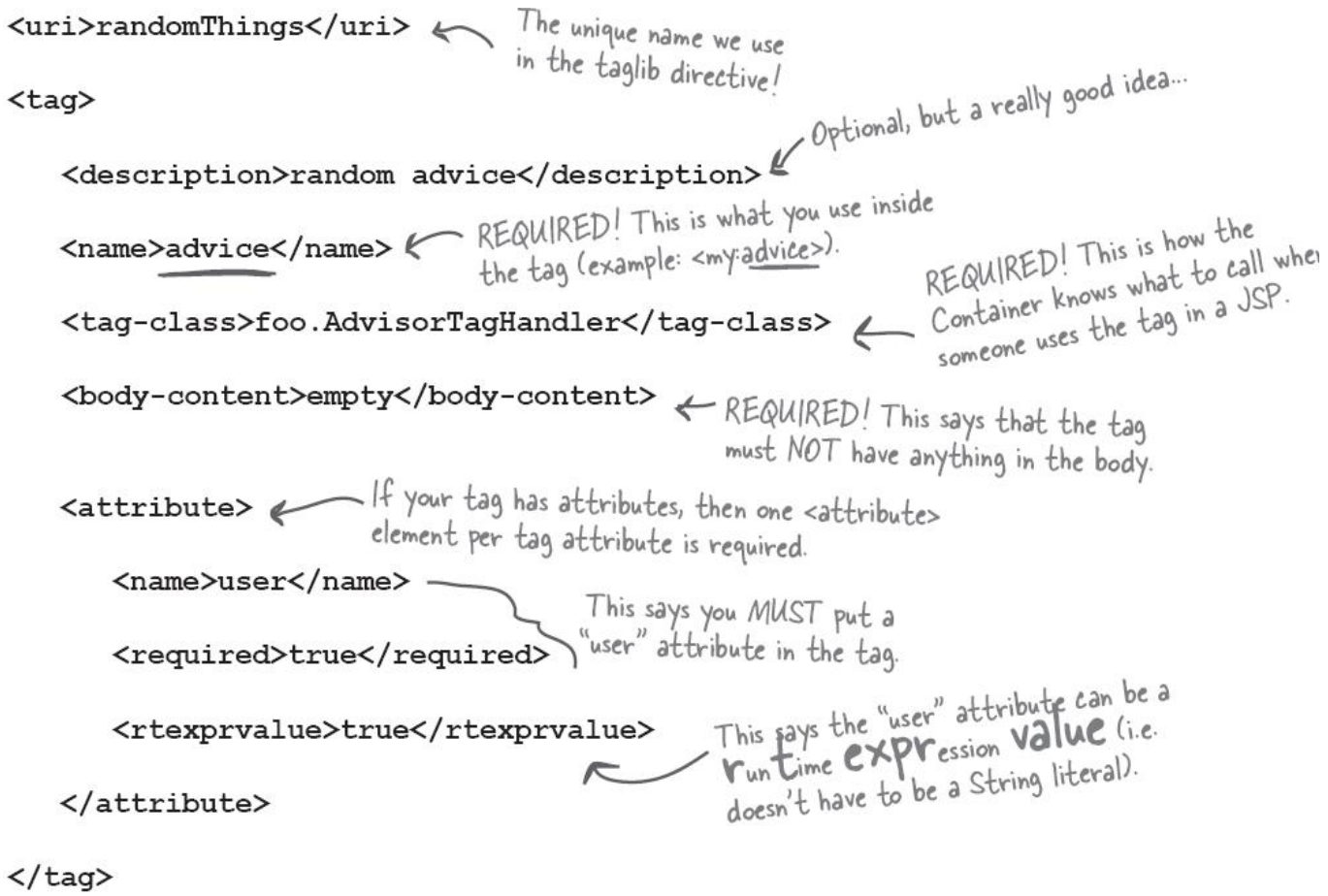
Custom Tag:

Using a tag library that's not from the JSTL:
We can also create our own tag and write tag handler which will be invoked when we put such custom tag in JSP page.
The tag name and syntax:
The tag has a name. In <c:set>, the tag name is set, and the prefix is c. You can use any prefix you want, but the name comes from the TLD.
The syntax includes things like:

- required and optional attributes,
- whether the tag can have a body
  (and if so, what you can put there),
- the type of each attribute, and
- whether the attribute can be an expression (vs. a literal String).

The library URI:
The URI is a unique identifier in the Tag Library Descriptor (TLD). When we put it in the tag directive It tells the Container how to identify the TLD file within the web app, which the Container needs in order to map the tag name used in the JSP to the Java code that runs when we use the tag.

The tag <tag>Inside TLD:



The TLD elements for the advice tag

```
<taglib ...>
...
<uri>randomThings</uri>
<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>

   <attribute>
      <name>user</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>

</tag>
</taglib ...>
```

This is the same tag you saw on the previous page, but without the annotations.

Each library you use in a page needs its own taglib directive with a unique prefix.

JSP that uses the tag

```
<html><body>

<%@ taglib prefix="mine" uri="randomThings"%>

Advisor Page<br>

<mine:advice user="${userName}" />

</body></html>
```

The uri matches the <uri> element in the TLD.

It's OK to use EL here, because the <rtexprevalue> in the TLD is set to "true" for the user attribute. (Assume the "userName" attribute already exists.)

The TLD says the tag can't have a body, so we made it an empty tag (which means the tag ends with a slash).

Java class that does the tag work

```
package foo;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;

public class AdvisorTagHandler extends SimpleTagSupport {

    private String user;

    public void doTag() throws JspException, IOException {
       getJspContext().getOut().write( "Hello " + user + " <br>" );
       getJspContext().getOut().write( "Your advice is: " + getAdvice() );
    }

    public void setUser(String user) {
       this.user=user;
    }

    String getAdvice() {
       String[] adviceStrings = {"That color's not working for you.",
             "You should call in sick.", "You might want to rethink that haircut."};
       int random = (int) (Math.random() * adviceStrings.length);
       return adviceStrings[random];
    }
}
```

SimpleTagSupport implements things we need in custom tags.

The Container calls doTag() when the JSP invokes the tag using the name declared in the TLD.

The Container calls this method to set the value from the tag attribute. It uses JavaBean property naming conventions to figure out that a "user" attribute should be sent to the setUser() method.

Our own internal method.

Understanding TLD:

The TLD describes two main things: custom tags, and EL functions.
We used one when we made the dice rolling function, but we had only a <function> element in the TLD.
Using the custom "advice" tag:
The "advice" tag is a simple tag that takes one attribute—the user name—and prints out a piece of random advice.
This simple tag handler extends SimpleTagSupport, and implements two key methods: doTag(), the method that does the actual work, and setUser(), the method that accepts the attribute value.

More on < rtexprvalue>:
The <rtexprvalue> is especially important because it tells you whether the value of the attribute is evaluated at translation or runtime.
If the <rtexprvalue> is false, or the <rtexprvalue> isn't defined, you can use only a String literal as that attribute's value!

If you see this:

```
<attribute>
   <name>rate</name>
   <required>true</required>
   <rtexprvalue>false</rtexprvalue>
</attribute>
```

or

```
<attribute>
   <name>rate</name>
   <required>true</required>

</attribute>
```

If there's no <rtexprvalue> the default value is false.

Then we know THIS WON'T WORK!

```
<html>
  <body>
     <%@ taglib prefi x="my" uri="myTags"%>
     <my:handleIt rate="${currentRate}" />
  </body>
</html>
```

NO! This must NOT be an expression... it must be a String literal.

<rtexprvalue> is not just for EL expressions:

We can use three kinds of expressions for the value of an attribute (or tag body) that allows runtime expressions.

1. EL expressions:
   <mine:advice user="${userName}" />
2. Scripting expressions:
   <mine:advice user='<%= request.getAttribute("username") %>' />
3. <jsp:attribute> standard actions:

```
<mine:advice>
   <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```

<jsp:attribute> lets you put attributes in the BODY of a tag, even when the tag body is explicitly declared "empty" in the TLD!!

The <jsp:attribute> is simply an alternate way to define attributes to a tag.
The key point is, there must be only one <jsp:attribute> for each attribute in the enclosing tag. So if we have a tag that normally takes three attributes in the tag, then inside the body we'll now have three <jsp:attribute> tags, one for each attribute. Also notice that the <jsp:attribute> has an attribute of its own, name, where we specify the name of the outer tag's attribute for which we're setting a value.

Think of the TLD as the API for the custom tag. You have to know how to call it and what arguments it needs.
The tag handler developer creates the TLD to tell both the Container and the JSP developer how to use the tag.

What can be in a tag body:
A tag can have a body only if the <body-content> element for this tag is not configured with a value of empty.

```
<body-content>empty</body-content>
```
The tag must NOT have a body.

```
<body-content>scriptless</body-content>
```
The tag must NOT have scripting elements (scriptlets, scripting expressions, and declarations), but it CAN have template text and EL and custom and standard actions.

```
<body-content>tagdependent</body-content>
```
The tag body is treated as plain text, so the EL is NOT evaluated and tags/actions are not triggered.

```
<body-content>JSP</body-content>
```
The tag body can have anything that can go inside a JSP.

THREE ways to invoke a tag that can't have a body:

Each of these are acceptable ways to invoke a tag configured in the TLD with: <body-content>empty</body-content>.

① An *empty* tag

```
<mine:advice user="${userName}" />
```
When you put a slash in the opening tag, you don't use a closing tag.

② A tag with *nothing* between the opening and closing tags

```
<mine:advice user="${userName}"> </mine:advice>
```
We have an opening and closing tag, but NOTHING in between.

③ A tag with only <jsp:attribute> tags between the opening and closing tags

```
<mine:advice>
   <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```
The <jsp:attribute> tag is the ONLY thing you can put between the opening and closing tags of a tag with a <body-content> of empty! It's just an alternate way to put the attributes in, but <jsp:attribute> tags don't count as "body content".

Before JSP 2.0 the Deployment Descriptor (web.xml) had to tell the Container where the TLD file with a matching <uri> was located.

The OLD (before JSP 2.0) way to map a taglib uri to a TLD file

```
<web-app>
...
  <jsp-config>
    <taglib>
      <taglib-uri>randomThings</taglib-uri>
      <taglib-location>/WEB-INF/myFunctions.tld</taglib-location>
    </taglib>
  </jsp-config>
</web-app>
```

In the DD, map the <uri> in the TLD to an actual path to a TLD file.

The NEW (JSP 2.0) way to map a taglib uri to a TLD file

No <taglib> entry in the DD!

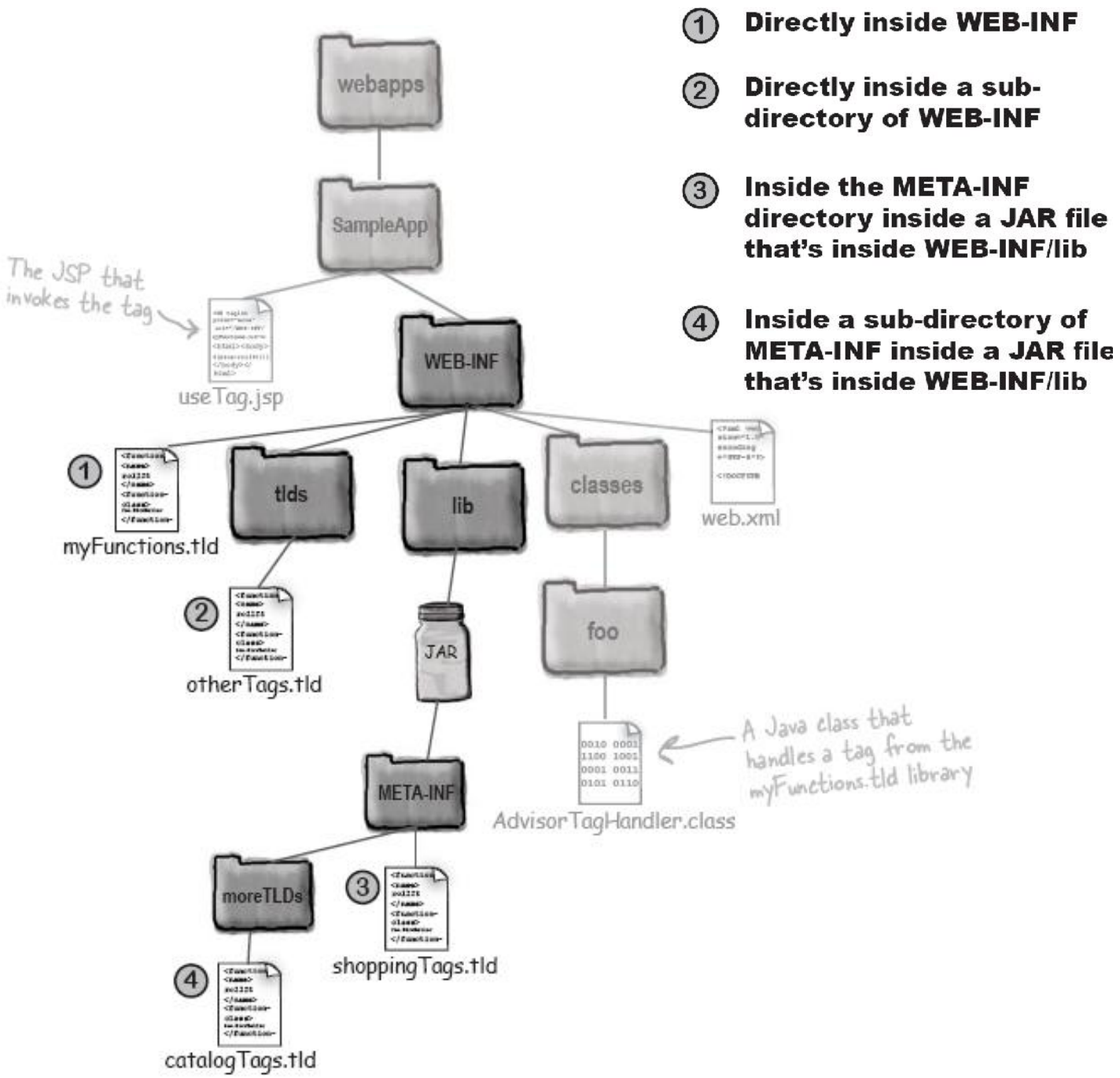
The taglib <uri>:

The <uri> element in the TLD is a unique name for the tag library. It does not need to represent any actual location (path or URL, for example). It simply has to be a name—the same name we use in the taglib directive.
The uri attribute value in the taglib directive for core library is neither an actual URL to any web resource, nor a path to the resource packaged in the application folder:

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

It's just the convention Sun uses for the uri, to help ensure that it's a unique name. All that matters is that the <uri> in the TLD and the uri in the taglib directive match! We can't, have two TLD files in the same web app, with the same <uri>. 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.

As long as you put the TLD in a place the Container will search, the Container will find the TLD and build a map for that tag library. If you do specify an explicit <taglib-location> in the DD (web.xml), and when the JSP 2.0 Container begins to build the <uri>-to-TLD map, the Container will look first in your DD to see if we've made any <taglib> entries, and if we have, it'll use those to help construct the map. So the next step is for us to see where the Container looks for TLDs, and also where it looks for the tag handler classes declared in the TLDs.

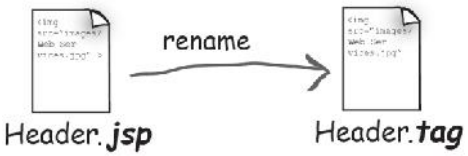

1 Directly inside WEB-INF
2 Directly inside a sub-directory of WEB-INF
3 Inside the META-INF directory inside a JAR file that's inside WEB-INF/lib
4 Inside a sub-directory of META-INF inside a JAR file that's inside WEB-INF/lib
webapps
SampleApp
The JSP that invokes the tag
useTag.jsp
WEB-INF
1
myFunctions.tld
tlds
lib
classes
web.xml
2
otherTags.tld
JAR
foo
A Java class that handles a tag from the myFunctions.tld library
AdvisorTagHandler.class
META-INF
moreTLDs
3
shoppingTags.tld
4
catalogTags.tld

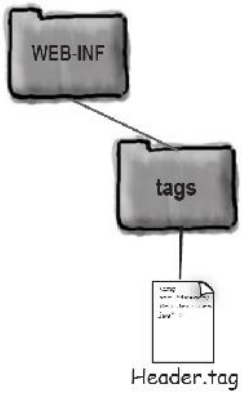
Tag Files:

With Tag Files, we can invoke reusable content using a custom tag instead of the generic <jsp:include> or <c:import>.
Tag Files let page developers create custom tags, without having to write a complicated Java tag handler class.

Simplest way to make and use a Tag File

1. Take an included file (like "Header.jsp") and rename it with a .tag extension.





2. Put the tag file (Header.tag) in a directory named "tags" inside the "WEB-INF" directory.

3. Put a taglib directive (with a tagdir attribute) in the JSP, and invoke the tag.

```
<%@ taglib prefix="abc" tagdir="/WEB-INF/tags" %>
<html><body>
    <myTags:Header />
```

Use the "tagdir" attribute in the taglib directive, instead of the "uri" TLDs for tag libraries.

The name of the tag is simply the name of the tag file! (minus the .tag extension)

Sending parameters to a tag file:

Instead of sending parameters to a tag file, we send attributes, which is opposed to the include mechanism we learnt earlier, where we send parameters, which is not a good idea. Because conceptually 'parameter' is used to represent information sent by client not programmer specified info., which must be an attributes.

Invoking the tag from the JSP:

Before (using <jsp:param> to set a request parameter):

```
<jsp:include page="Header.jsp">
    <jsp:param name="subTitle" value="ParamValue" />
</jsp:include>
```

After (using a Tag with an attribute)

```
<myTags:Header subTitle="AttribValue" />
```

Using the attribute in the Tag File:

Before (using a request param value) -

```
<h1>${param.subTitle}</h1>
```

After (using a Tag File attribute) -

```
<h1>${subTitle}</h1>
```

This is inside the actual Tag File (in other words, the included file).

All tag attributes have TAG scope.
Once the tag is closed, the tag attributes go out of scope!
This will NOT work:

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>
  <myTags:Header subTitle="AttribValue" />
  <br>
  ${subTitle}
</body></html>
```

This won't work! The attribute is out of scope

Tag Files use the attribute directive:

Inside the Tag File
(Header.tag)

This means the attribute is not optional.

It can be a String literal OR an expression.

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

<img src="images/Web-Services.jpg" > <br>
<em><strong>${subTitle}</strong></em> <br>
```

Inside the JSP that uses the tag

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>
<myTags:Header subTitle="We take the String out of SOAP" />

<br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

Tag Body:

Imagine we have a tag attribute that might be as long as, a paragraph. Sticking that in the opening tag could get ugly. We can either put content in the body of the tag. Now we'll take the subTitle attribute out of the tag, and instead make it the body of the <myTags:Header> tag.

Inside the Tag File *(Header.tag)*

We no longer need the attribute directive!

```
<img src="images/Web-Services.jpg" > <br>
<em><strong><jsp:doBody/></strong></em> <br>
```

This says, "Take whatever is in the body of the tag used to invoke this tag file, and stick it here."

Inside the JSP that uses the tag

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

<myTags:Header>
     Paragraph text here... Paragraph text here... Paragraph text here...
     Paragraph text here... Paragraph text here... Paragraph text here...
     Paragraph text here... Paragraph text here... Paragraph text here...
</myTags:Header>

<br>
```

Now we just give the tag a body, instead of putting all this as the value of an attribute in the opening tag.

Declaring body-content for a Tag File:

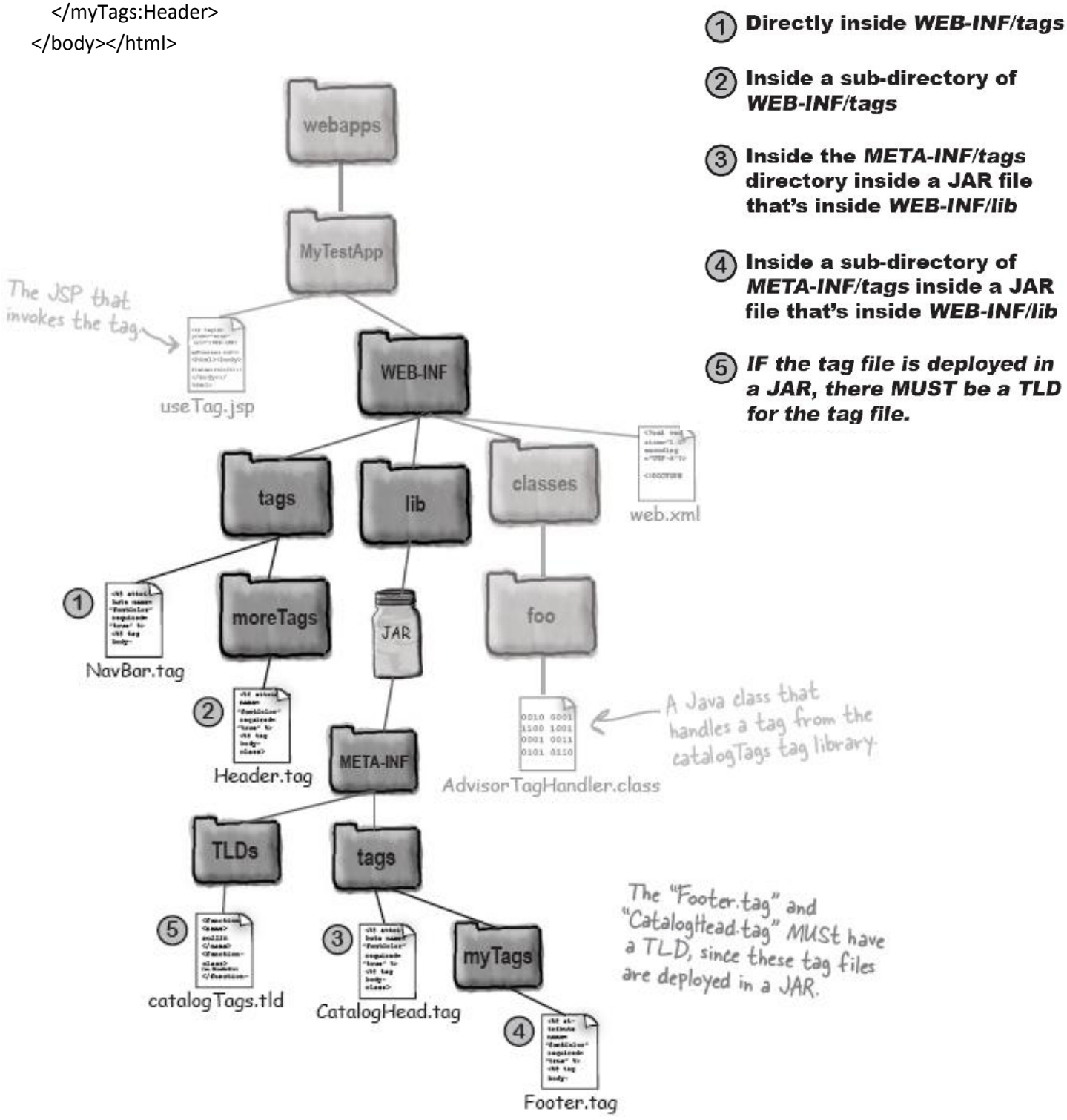
The tag directive is used to declare body-content type for a Tag File.
For a custom tag, the <body-content> element inside the <tag> element of a TLD is mandatory! But a Tag File does not have to declare tag directive because default value of the attribute body-content scriptless.
A value of scriptless means we can't have scripting elements. And scripting elements, are scriptlets (<% ... %>), expressions (<%= ... %>), and declarations (<%! ... %>).
You can not use scripting code in the body of a Tag File tag!
The body-content of a Tag File defaults to "scriptless", so you don't have to declare body-content unless you want one of the other two options: "empty" (nothing in the tag body) or "tagdependent" (treats the body as plain text).

Inside the Tag File with a tag directive (Header.tag):

```
<%@ attribute name="fontColor" required="true" %>
<%@ tag body-content="tagdependent" %>
<font color="${fontColor}"><jsp:doBody/></font>
```

Inside the JSP that uses the tag:

```
<%@ taglib  prefix="myTags"  tagdir="/WEB-INF/tags" %>
<html><body>
  <myTags:Header  fontColor="#660099">
        Paragraph text here... Paragraph text here... Paragraph text here...
        Paragraph text here... Paragraph text here... Paragraph text here...
        Paragraph text here... Paragraph text here... Paragraph text here...
  </myTags:Header>
</body></html>
```



We can do scripting in a Tag File, but we can't do scripting inside the body of the tag used to invoke the Tag File. Because a Tag File is going to be part of a Jsp.
Tag File can use the implicit request and response objects, the normal EL implicit objects are always there as well, and we can access to a JspContext as well.
Tag File don't have a ServletContext, but it uses a JspContext instead of a ServletContext. Tag files implement the tag functionality with another page (using JSP). Tag handlers implement the tag functionality with a special Java class.
On one hand the EL functions are nothing more than static methods, a tag handler class has access to tag attributes, the tag body, and even the page context so it can get scoped attributes and the request and response.
Tag handlers come in two flavors: Simple and Classic.
Classic tags were all we had in the previous version of JSP, but with JSP 2.0, a new and much simpler model was added. We still use classic tags because we might get a chance to work on older code where it was used.

Access Tag Body:

If the tag needs a body, the TLD <body-content> needs to reflect that, and we need a special statement in the doTag() method.

The JSP that uses the tag:

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
  Simple Tag 2:
  <myTags:simple2>
     This is the body
  </myTags:simple2>
</body></html>
```

The tag handler class:

```
  package  abc;
   //All imports…
  public class SimpleTagTest2 extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
      getJspBody().invoke(null);
    }
  }
```

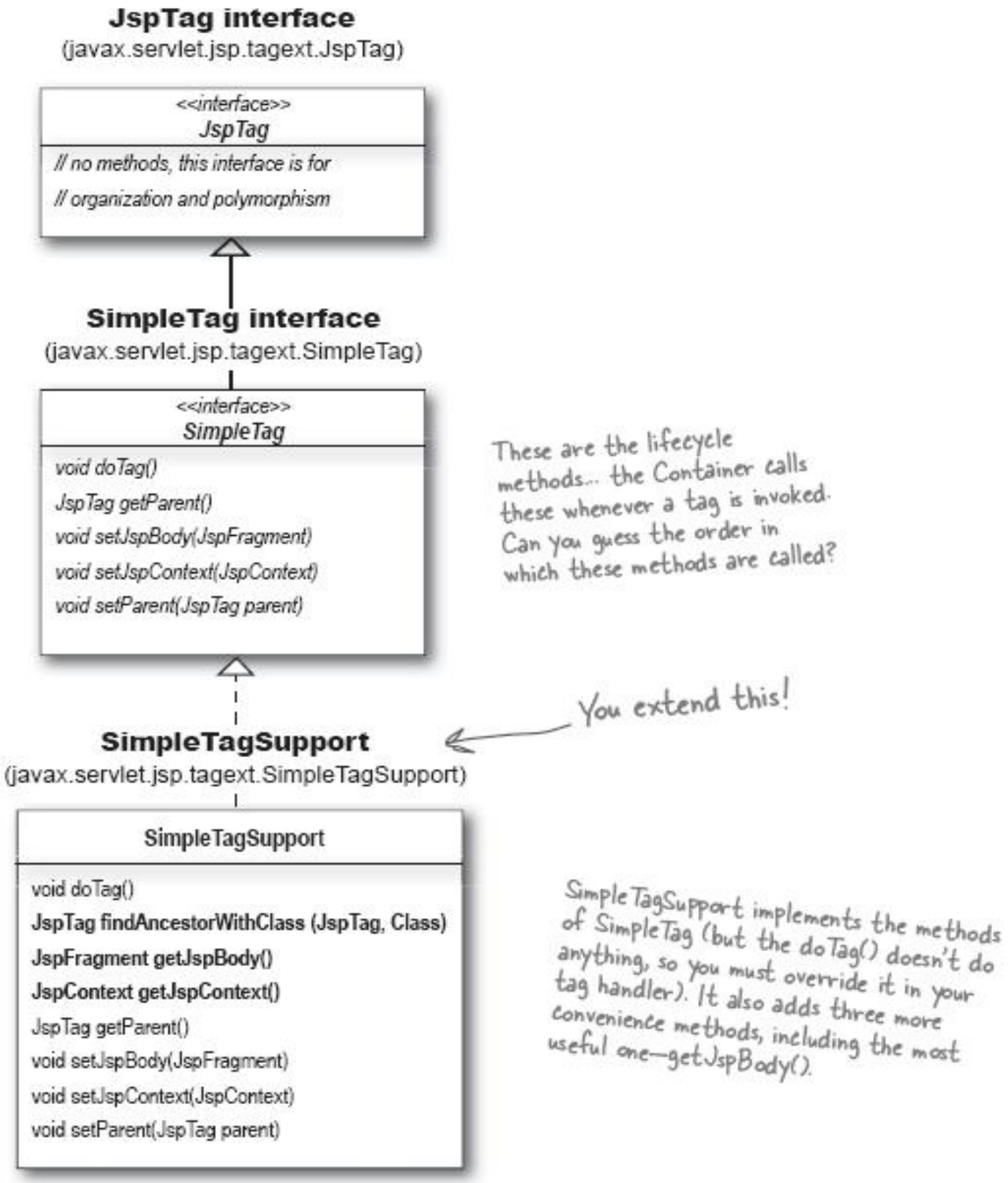
This says, "Process the body of the tag and print it to the response". The null argument means the output goes to the response rather than some OTHER writer you pass in.

The TLD for the tag:

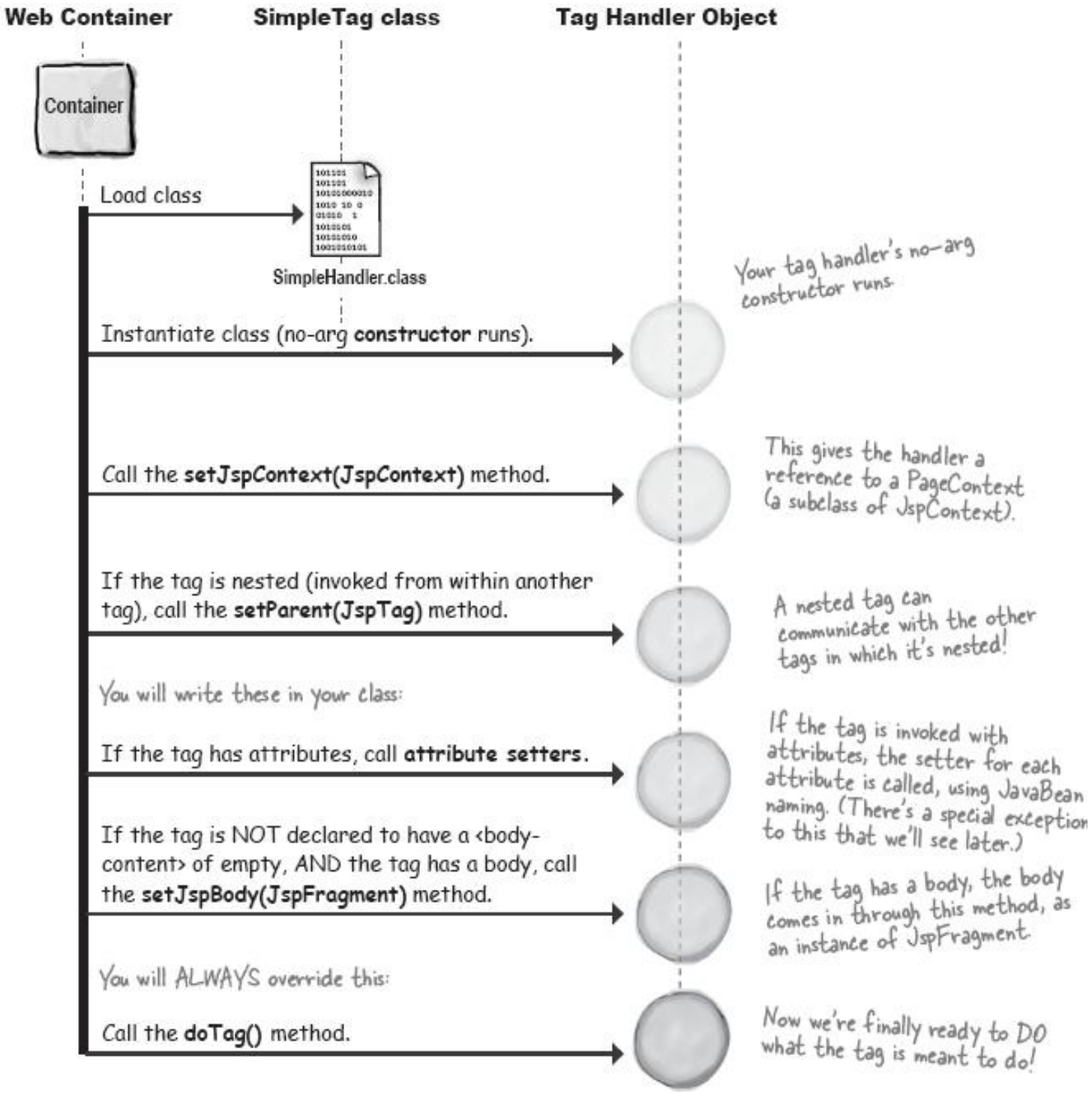
```
. . .
<tag>
   <description>marginally better use of a custom tag</description>
   <name>simple2</name>
   <tag-class>abc.SimpleTagTest2</tag-class>
   <body-content>scriptless</body-content>
</tag>
```

This says the tag can have a body, but the body cannot have scripting (scriptlets, scripting expressions, or declarations).

The Simple tag API:



The life of a Simple tag handler:



When a JSP invokes a tag, a new instance of the tag handler class is instantiated, and when the doTag() method completes, the handler object goes away.

What if the tag body uses an expression?

Imagine you have a tag with a body that uses an EL expression for an attribute. Now imagine that the attribute doesn't exist at the time you invoke the tag! In other words, the tag body depends on the tag handler to set the attribute.

The JSP tag invocation

```
<myTags:simple3>
   Message is: ${message}
</myTags:simple3>
```

At the point where the tag is invoked, "message" is NOT a scoped attribute! If you took this expression out of the tag, it would return null.

The tag handler doTag() method

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

The tag handler sets an attribute and THEN invokes the body.

A tag with dynamic row data:

In this example, the EL expression in the body of the tag represents a single value in a collection, and the goal is to have the tag generate one row for each element in the collection.
It's simple—the doTag() method simply does the work in a loop, invoking the body on each iteration of the loop.

The JSP tag invocation

```
<table>
  <myTags:simple4>
    <tr><td>${movie}</td></tr>
  </myTags:simple4>
</table>
```

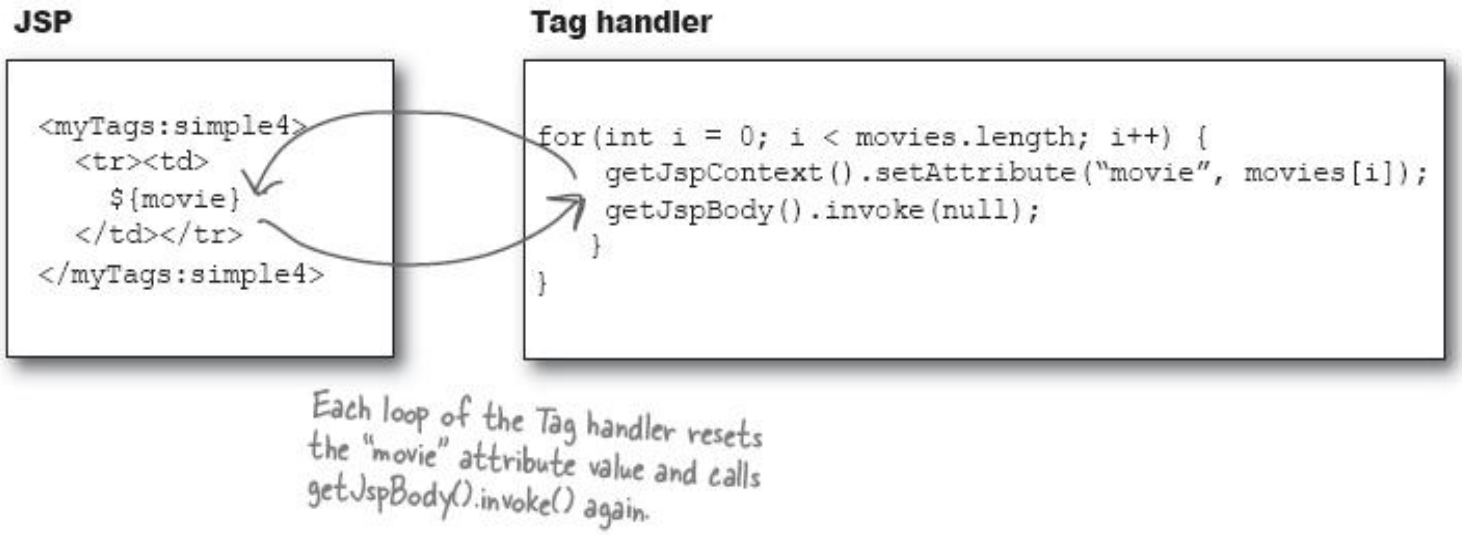
The movie attribute doesn't exist at the time the tag is invoked. It will be set by the tag handler, and the body will be called repeatedly.

The tag handler doTag() method

```
String[] movies = {"Monsoon Wedding", "Saved!", "Fahrenheit 9/11"};

public void doTag() throws JspException, IOException {
   for(int i = 0; i < movies.length; i++) {
      getJspContext().setAttribute("movie", movies[i]);
      getJspBody().invoke(null);
   }
}
```

Set the attribute value to be the next element in the array.

Invoke the body again.



What exactly IS a JspFragment?

A JspFragment is an object that represents JSP code. Its sole purpose in life is to be invoked. In other words, it's something that's 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.

The crucial thing we must remember about JspFragment is that it must not contain any scripting elements!
It can contain template text, standard and custom actions, and EL expressions, but no scriptlets, declarations, or scripting expressions.

Since it is an object, we can even pass the fragment around to other helper objects. And those objects, in turn, can get information from it by invoking the JspFragment's other method—getJspContext().

Once we've got a context, we can ask for attributes. So the getJspContext() method is really a way for the tag body to get information to other objects. Most of the time, we'll use JspFragment simply to output the body of the tag to the response. We might, however, want to get access to the contents of the body.

Notice that JspFragment doesn't have an access method like getContents() or getBody(). We can write the body to something, but we can't directly get the body. If we do want access to the body, we 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 we want the body to be written to the response output, pass null to the invoke method. Most of the time, that's what we'll do. But if we want access to the actual contents of the body, we can pass in a Writer, then use that Writer to process the body in some way.

SkipPageException:

Imagine we're in a page that invokes the tag, and the tag depends on specific request attributes (that it gets from the JspContext available to the tag handler).
Now imagine the tag can't find the attributes it needs, and that the tag knows the rest of the page will never work if the tag can't succeed. What do we do?
Imagine if an unavailability of a required resource or any other reasons the tag throws a JspException from inside a tag-handler, but that would kill the page... but what if it's only the rest of the page that won't work?
If we wanted to display the page before the custom-tag and wanted to skip the rest of the page after the custom tag, then we must manually throw the SkipPageException from within the tag-handler class. Everything in the doTag() method up to the point of the SkipPageException still shows up in the response. But after the exception, anything still left in either the tag or the page won't be evaluated.

In the JSP

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
About to invoke a tag that throws SkipPageException <br>
<myTags:simple6/>

<br>Back in the page after invoking the tag.

</body></html>
```

This doesn't print out!

In the tag handler

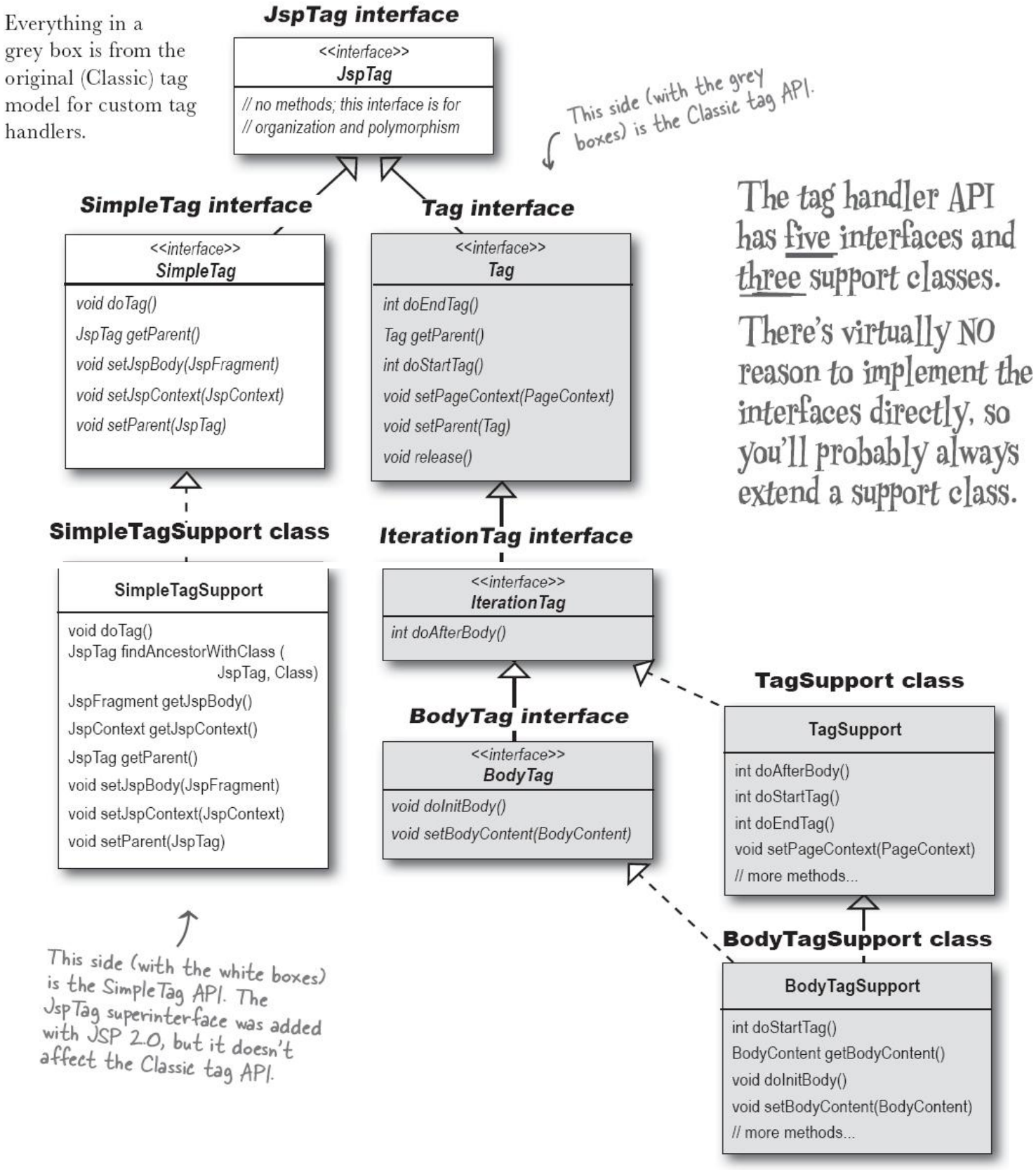
```
public void doTag() throws JspException, IOException {
   getJspContext().getOut().print("Message from within doTag().<br>");
   getJspContext().getOut().print("About to throw a SkipPageException");
   if (thingsDontWork) {
     throw new SkipPageException();
   }
}
```

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. SimpleTag handlers are never reused! Each tag handler instance takes care of a single invocation. A SimpleTag handler object will always be initialized before any of its methods are called.

Classic tag handlers:

We don't ever have write a Classic tag handler but probably need to at least read the source code for a Classic tag handler. You might be called on to maintain or refactor a Classic tag handler class.

Tag handler API :



A Classic tag handler Example:

A JSP that invokes a Classic tag:

```
<%@ taglib prefix="mine" uri="ClassicTags" %>
<html><body>
   Classic Tag One:<br>
   <mine:classicOne />
</body></html>
```

The TLD <tag> element for the Classic tag:

```
<tag>
    <description>A use of a Classic tag</description>
    <name>classicOne</name>
    <tag-class>abc.Classic1</tag-class>
    <body-content>empty</body-content>
</tag>
```

The Classic tag handler:

```
package abc;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class Classic1 extends TagSupport {
        public int doStartTag() throws JspException {
                JspWriter out = pageContext.getOut();
                try {
                        out.println("classic tag output");
                } catch(IOException ex) {
                        throw new JspException("IOException- " + ex.toString());
                }
                return SKIP_BODY;
        }
}
```

The methods declare JspException, but NOT an IOException! (The SimpleTag doTag() declares IOException.)

One More Example:

A JSP that invokes a Classic tag:

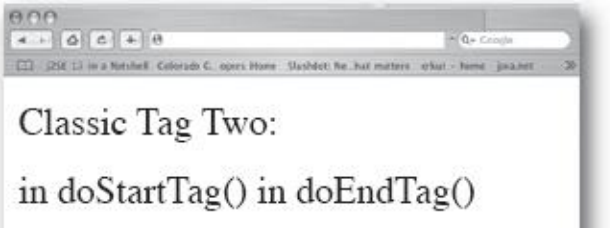
```
<%@ taglib prefix="mine" uri="ClassicTags" %>
<html><body>
   Classic Tag Two:<br>
   <mine:classicTwo />
</body></html>
```

Here in this example we will override two methods: doStartTag() and doEndTag():
The point of doEndTag() is that it's called after the body is evaluated. Assume the TLD here is virtually identical to the previous one, except for some of the names. The tag is declared to have no attributes, and an empty body.

```
public class Classic2 extends TagSupport {
    JspWriter out;

    public int doStartTag() throws JspException {
        out = pageContext.getOut();
        try {
          out.println("in doStartTag()");
        } catch(IOException ex) {
           throw new JspException("IOException- " + ex.toString());
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
       try {
          out.println("in doEndTag()");
        } catch(IOException ex) {
           throw new JspException("IOException- " + ex.toString());
        }
      return EVAL_PAGE;
  }
}
```

This says, "Don't evaluate the body if there is one—just go straight to the doEndTag() method."

This says, "Evaluate the rest of the page" (as opposed to SKIP_PAGE, which would be just like throwing a SkipPageException from a SimpleTag handler).



Comparing Simple vs. Classic:

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.

The JSP that uses the tag

```
<%@ taglib prefix="myTags" uri="myTags" %>
<html><body>
  <myTags:simpleBody>
     This is the body
  </myTags:simpleBody>
</body></html>
```

A SimpleTag handler class

```
// package and imports
public class SimpleTagTest extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().print("Before body.");
        getJspBody().invoke(null);
        getJspContext().getOut().print("After body.");
    }
}
```

This causes the body to be evaluated.

A Classic tag handler that does the same thing

```
// package and imports
public class ClassicTest extends TagSupport {
   JspWriter out;

   public int doStartTag() throws JspException {
      out = pageContext.getOut();
      try {
        out.println("Before body.");
      } catch(IOException ex) {
         throw new JspException("IOException- " + ex.toString());
      }
      return EVAL_BODY_INCLUDE;
    }

   public int doEndTag() throws JspException {
      try {
        out.println("After body.");
      } catch(IOException ex) {
         throw new JspException("IOException- " + ex.toString());
      }
      return EVAL_PAGE;
   }
}
```

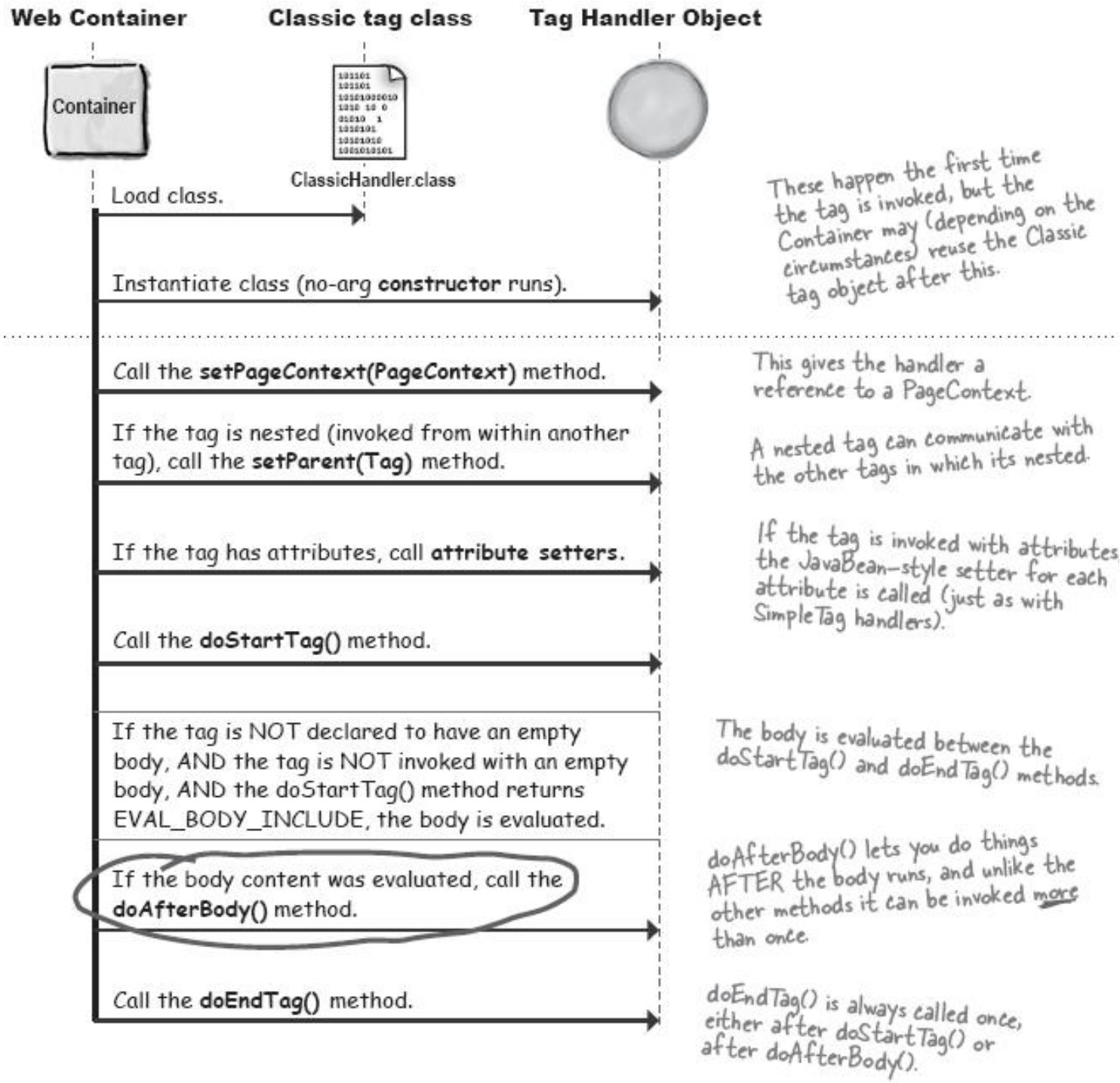
THIS is what causes the body to be evaluated in a Classic tag handler!

Classic tags have a different lifecycle:

With classic tags, there's a doStartTag() and a doEndTag(). And that brings up an interesting problem—when and how is the body evaluated? There's no doBody() method, but there is a doAfterBody() method that's called after the body is evaluated and before the doEndTag() runs.

The Classic lifecycle depends on return values:

The doStartTag() and doEndTag() methods return an int. That int tells the Container what to do next. With doStartTag(), the question the Container asks is, "Should I evaluate the body?" (assuming there is one, and assuming the TLD doesn't declare the body as empty). With doEndTag(), the Container asks, "Should I keep evaluating the rest of the calling page?" The return values are represented by constants declared in the Tag and IterationTag interfaces.



Possible return values when you extend TagSupport

doStartTag()

SKIP_BODY

EVAL_BODY_INCLUDE

doAfterBody()

SKIP_BODY

EVAL_BODY_AGAIN

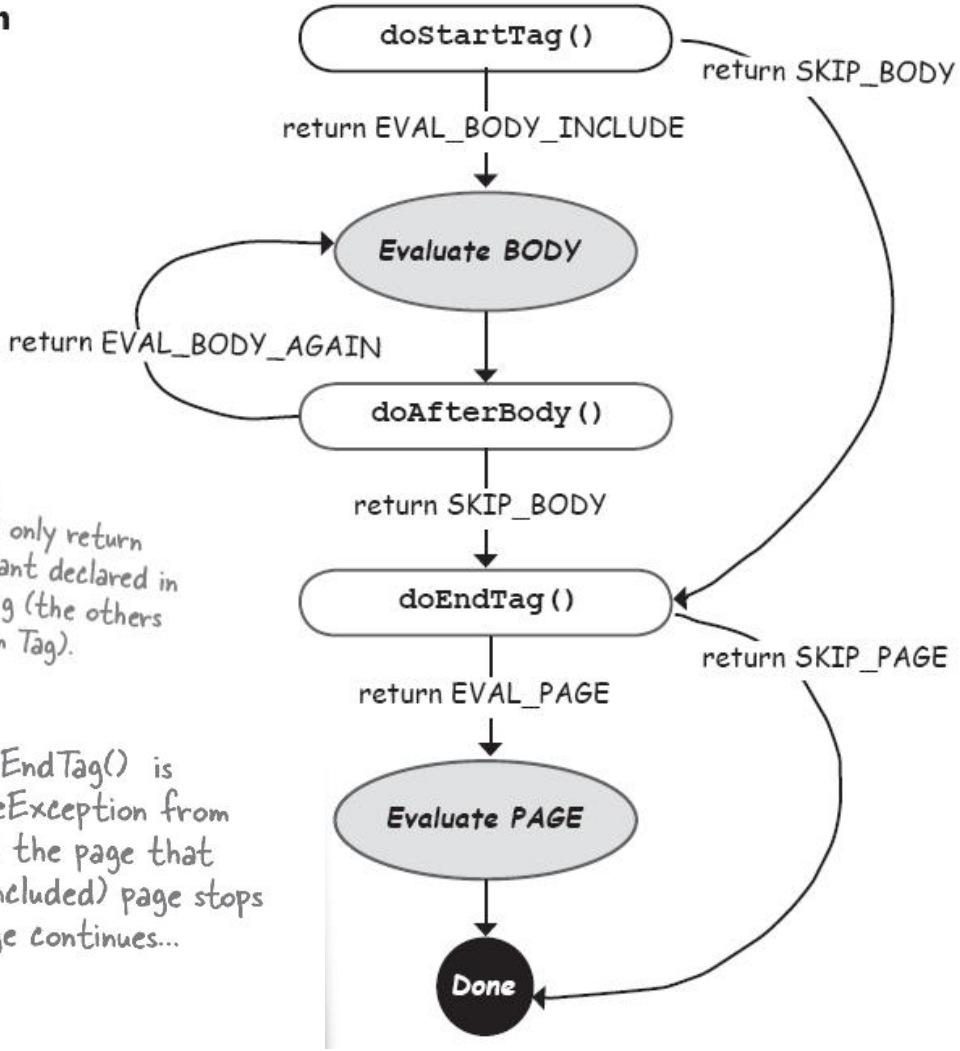
This is the only return value constant declared in IterationTag (the others are all from Tag).

doEndTag()

SKIP_PAGE

EVAL_PAGE

Returning SKIP_PAGE from doEndTag() is exactly like throwing a SkipPageException from a Simple tag! If a page included the page that invoked the tag, the current (included) page stops processing, but the including page continues...



Default return values from TagSupport:

If you don't override the TagSupport lifecycle methods that return an integer, be aware of the default values the TagSupport method implementations return. The TagSupport class assumes that your tag doesn't have a body (by returning SKIP_BODY) from doStartTag()), and that if you DO have a body that's evaluated, you want it evaluated only once (by returning SKIP_BODY from doAfterBody()). It also assumes that you want the rest of the page to evaluate (by returning EVAL-PAGE from doEndtag()).

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 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.

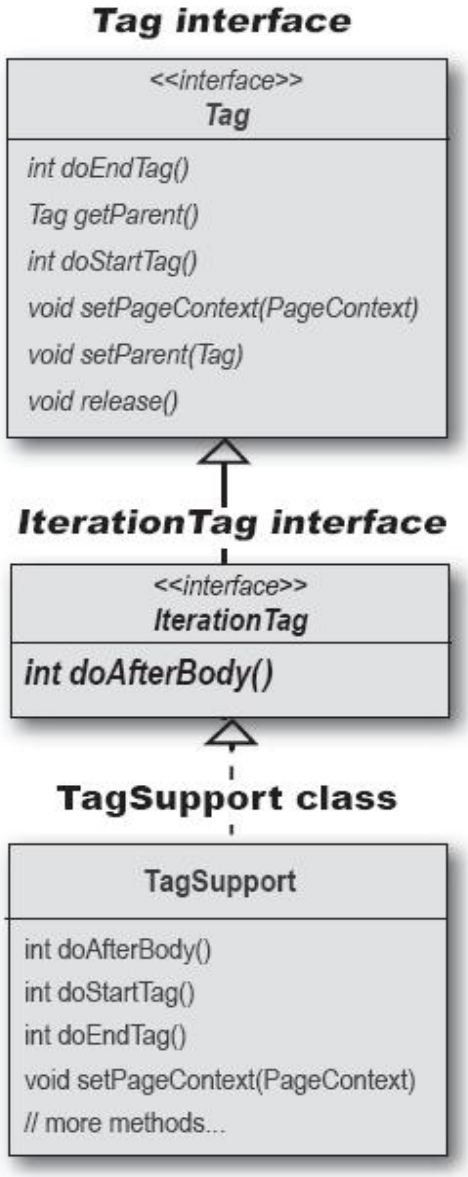
It also assumes that you always want the rest of the page to be evaluated.

doStartTag() and doEndTag() run exactly once.

You must override doStartTag() if you want the tag body to be evaluated!!

IterationTag lets us repeat the body:

When we write a tag handler that extends TagSupport, we get all the life cycle 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.
But with doAfterBody(), our return value tells the Container whether it should repeat the body again (EVAL_BODY_AGAIN) or call the doEndTag() method (SKIP_BODY).



The Container can reuse Classic tag handlers!

Remember—Classic Tag is completely different from SimpleTag handlers which are definitely not reused. That means you have to be very careful about instance variables—you should reset them in doStartTag().
The Tag interface does have a release() method, but that's called only when the tag handler instance is about to be removed by the Container. So don't assume that release() is a way to reset the tag handler's state in between tag invocations!

What if we need access to the body contents?

We found that most of the time the lifecycle methods from the Tag and IterationTag interfaces, as provided by TagSupport, are enough. Between the three key methods (doStartTag(), doAfterBody(), and doEndTag()), we can do anything. We don't have direct access to the contents of the body. If we need access to the actual body contents, so that we can, use it in an expression or perhaps filter or alter it in some way, then extend BodyTagSupport instead of TagSupport, and we'll have access to the BodyTag interface methods.