SCWCD 1.4 Study Guide

IBA JV

Mikalai Zaikin

<NZaikin[at]iba.by>

March 2005

Revision History		
Revision 0.8	14 Mar 2005	MZ
Small Servlet API error correction. Added EJB-related tags descriptions. Proposed Final Draft 5.		
Revision 0.7	15 Nov 2004	MZ
JSP Custom Tag API correction. Proposed Final Draft 4.		
Revision 0.6	7 May 2004	MZ
Proposed Final Draft 3.		
Revision 0.5	5 Apr 2004	MZ
Proposed Final Draft 2.		
Revision 0.4	19 Jan 2004	MZ
Proposed Final Draft 1.		
Revision 0.3	15 Jan 2004	MZ
Chapters 9-10 updates.		
Revision 0.2	12 Jan 2004	MZ
Chapters 1-8 updates.		
Revision 0.1	22 Dec 2003	MZ
Initial release.		

Abstract

The purpose of this document is to help in preparation for exam CX-310-081 (Sun Certified Web Component Developer using the J2EE Platform 1.4).

This document should not be used as the only study material for SCWCD 1.4 test. It covers all objective topics, but it is not enough. I tried to make this document as much accurate as possible, but if you find any error, please let me know.

Table of Contents

purpose of the method and the technical characteristics of the HTTP Method protocol, list triggers that might cause a Client (usually a Web browser) to use the method; and identify the HttpServlet method that corresponds to the HTTP Method.

Using the HttpServletRequest interface, write code to retrieve HTML form parameters from the request, retrieve HTTP request header information, or retrieve cookies from the request.

Using the HttpServletResponse interface, write code to set an HTTP response header, set the content type of the response, acquire a text stream for the response, acquire a binary stream for the response, redirect an HTTP request to another URL, or add cookies to the response.

Describe the purpose and event sequence of the servlet life cycle: (1) servlet class loading, (2) servlet instantiation, (3) call the init method, (4) call the service method, and (5) call destroy method.

2. The Structure and Deployment of Web Applications

Construct the file and directory structure of a Web Application that may contain (a) static content, (b) JSP pages, (c) servlet classes, (d) the deployment descriptor, (e) tag libraries, (f) JAR files, and (g) Java class files; and describe how to protect resource files from HTTP access.

Describe the purpose and semantics for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

Construct the correct structure for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

Explain the purpose of a WAR file and describe the contents of a WAR file, how one may be constructed.

3. The Web Container Model

For the ServletContext initialization parameters: write servlet code to access initialization parameters; and create the deployment descriptor elements for declaring initialization parameters.

For the fundamental servlet attribute scopes (request, session, and context): write servlet code to add, retrieve, and remove attributes; given a usage scenario, identify the proper scope for an attribute; and identify multi-threading issues associated with each scope.

Describe the Web container request processing model; write and configure a filter; create a request or response wrapper; and given a design problem, describe how to apply a filter or a wrapper.

Describe the Web container life cycle event model for requests, sessions, and web applications; create and configure listener classes for each scope life cycle; create and configure scope attribute listener classes; and given a scenario, identify the proper attribute listener to use.

Describe the RequestDispatcher mechanism; write servlet code to create a request dispatcher; write servlet code to forward or include the target resource; and identify and describe the additional request-scoped attributes provided by the container to the target resource.

4. Session Management

Write servlet code to store objects into a session object and retrieve objects from a session object.

Given a scenario describe the APIs used to access the session object, explain when the session object was created, and describe the mechanisms used to destroy the session object, and when it was destroyed.

Using session listeners, write code to respond to an event when an object is added to a session, and write code to respond to an event when a session object migrates from one VM to another.

Given a scenario, describe which session management mechanism the Web container could employ, how cookies might be used to manage sessions, how URL rewriting might be used to manage sessions, and write servlet code to perform URL rewriting.

5. Web Application Security

Based on the servlet specification, compare and contrast the following security mechanisms: (a) authentication, (b) authorization, (c) data integrity, and (d) confidentiality.

In the deployment descriptor, declare a security constraint, a Web resource, the transport guarantee, the login configuration, and a security role.

Compare and contrast the authentication types (BASIC, DIGEST, FORM, and CLIENT-CERT); describe how the type works; and given a scenario, select an appropriate type.

6. The JavaServer Pages (JSP) Technology Model

Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c)

Preface

Exam Objectives

Chapter 1. The Servlet Technology Model

For each of the HTTP Methods (such as `GET`, `POST`, `HEAD`, and so on) describe the purpose of the method and the technical characteristics of the HTTP Method protocol, list triggers that might cause a Client (usually a Web browser) to use the method; and identify the `HttpServlet` method that corresponds to the HTTP Method.

The `HttpServlet` abstract subclass adds additional methods beyond the basic `Servlet` interface that are automatically called by the `service` method in the `HttpServlet` class to aid in processing HTTP-based requests. These methods are (HTTP 1.1):

- `doGet` for handling HTTP `GET` requests.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a `GET` request.

Overriding this method to support a `GET` request also automatically supports an HTTP `HEAD` request. A `HEAD` request is a `GET` request that returns NO BODY in the response, only the request header fields. When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DisplayServlet extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws IOException, ServletException {
                resp.setContentType("text/html");
                PrintWriter out = resp.getWriter();
                out.println("<html><head><title>Display Information");
                out.println("</title></head><body>");
                out.println("Hello, World");
                out.println("</body></html>");
        }
}
```

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

The `GET` method should be safe, that is, without any side effects for which users are held responsible. For example, most form queries have no side effects. If a client request is intended to change stored data, the request should use some other HTTP method (for example, `POST` method).

The `GET` method should also be idempotent, meaning that it can be safely repeated. Sometimes making a method safe also makes it idempotent. For example, repeating queries is both safe and idempotent, but buying a product online or modifying data is neither safe nor idempotent.

GET method purpose.

The `GET` method means retrieve whatever information (in the form of an entity) is identified by the `Request-URI`. If the `Request-URI` refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

In short, this method should be used for getting (retrieving) data only. It should not be used for storing data in DB.

GET method technical characteristics.

Query string or form data during this method is simply appended to the URL as name-value pairs separated with '&'.

```
name1=value1&name2=value2&name3=value3
```

Query length is limited (it depends on servlet container's plaform, but usually should not exceed 1024 bytes). Users can see data in the browser's address bar.

```
http://some-server.com/some-script?name1=value1&name2=value2&name3=value3
```

Only ASCII (text) data can be sent to server with `GET` method.

Easy to bookmark.

GET method triggers.

The web browser sends an HTTP `GET` request when:

- The user types a URL in the browser's address bar.
- The user clicks a link.
- Retrieve a resource which was defined in `src` (image) or `href` (cascade style sheet) attributes.

  ```
  <img src="image.gif">
  ```

  ```
  <link href="style.css" rel="stylesheet" type="text/css">
  ```

- The user (or JavaScript) submits a form that specifies attribute `method="GET"`:

```
<form action="/servlet/display" method="GET">
        First Name: <input type="text" name="firstName"><p>
        Last Name: <input type="text" name="lastName"><p>
        <input type="submit" value="Display">
</form>
```

  - The user (or JavaScript) submits a form that specifies NO `method` attribute (forms use method `GET` BY DEFAULT):

```
<form action="/servlet/display">
        First Name: <input type="text" name="firstName"><p>
        Last Name: <input type="text" name="lastName"><p>
        <input type="submit" value="Display">
</form>
```

- `doPost` for handling HTTP `POST` requests.

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a `POST` request. The HTTP `POST` method allows the client to send data of UNLIMITED length to the Web server a single time and is useful when posting information such as credit card numbers.

When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DisplayServlet extends HttpServlet {
        public void doPost(HttpServletRequest req, HttpServletResponse resp
                        throws IOException, ServletException {
                resp.setContentType("text/html");
                PrintWriter out = resp.getWriter();
                out.println("<html><head><title>Display Information");
                out.println("</title></head><body>");
                out.println("Hello, World");
                out.println("</body></html>");
        }
}
```

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

This method does not need to be either safe or idempotent. Operations requested through `POST` can have side effects for which the user can be held accountable, for example, updating stored data or buying items online.

POST method purpose.

The `POST` method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the `Request-URI` in the `Request-Line`. `POST` is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources;
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.

In short, this method should be used for posting newgroups messages, submitting long data fields to a database (such as a SQL insert of lengthy string), or sending binary files to server.

POST method technical characteristics.

Sends information to the server such as form fields, large text bodies, and key-value pairs.

Hides form data from users because it is not passed as a query string, but in the message body.

Sends UNLIMITED length data as part of its HTTP request body.

For sending ASCII (text) or binary data.

Disallows bookmarks.

POST method triggers.

The web browser sends an HTTP `POST` request when:

- The user (or JavaScript) submits a form that specifies attribute `method="POST"`:

```
<form action="/servlet/display" method="POST">
        First Name: <input type="text" name="firstName"><p>
        Last Name: <input type="text" name="lastName"><p>
        <input type="submit" value="Display">
</form>
```

- `doPut` for handling HTTP `PUT` requests.

```
protected void doPut(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a `PUT` request. The `PUT` operation allows a client to place a file on the server and is similar to sending a file by FTP.

When overriding this method, leave intact any content headers sent with the request (including `Content-Length`, `Content-Type`, `Content-Transfer-Encoding`, `Content-Encoding`, `Content-Base`, `Content-Language`, `Content-Location`, `Content-MD5`, and `Content-Range`). If your method cannot handle a content header, it must issue an error message and discard the request.

This method does not need to be either safe or idempotent. Operations that `doPut` performs can have side effects for which the user can be held accountable. When using this method, it may be useful to save a copy of the affected URL in temporary storage.

The fundamental difference between the `POST` and `PUT` requests is reflected in the different meaning of the `Request-URI`. The URI in a `POST` request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource.

```
<form enctype="multipart/form-data" action="some_url" method="PUT">
        <input type="file" name="fileToUpload" value="Select File">
        <input type="submit" value="Upload">
        <input type="reset" value="Reset">
</form>
```

- `doDelete` for handling HTTP `DELETE` requests.

```
protected void doDelete(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a `DELETE` request. The `DELETE` operation allows a client to remove a document or Web page from the server.

This method does not need to be either safe or idempotent. Operations requested through `DELETE` can have side effects for which users can be held accountable. When using this method, it may be useful to save a copy of the affected URL in temporary storage.

- `doHead` for handling HTTP `HEAD` requests.

```
protected void doHead(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Receives an HTTP `HEAD` request from the protected `service` method and handles the request. The client sends a `HEAD` request when it wants to see ONLY the HEADERS of a response, such as `Content-Type` or `Content-Length`. The HTTP `HEAD` method counts the output bytes in the response to set the `Content-Length` header accurately.

The `doHead` method in `HttpServlet` is a specialized form of the `doGet` method that returns only the headers produced by the `doGet` method.

- `doOptions` for handling HTTP `OPTIONS` requests.

```
protected void doOptions(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a `OPTIONS` request.

The `OPTIONS` request determines which HTTP methods the server supports and returns an appropriate header. For example, if a servlet overrides `doGet`, this method returns the following header:

```
Allow: GET, HEAD, TRACE, OPTIONS
```

- `doTrace` for handling HTTP `TRACE` requests.

```
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a `TRACE` request.

The `doTrace` method generates a response containing all instances of the headers sent in the `TRACE` request to the client, so that they can be used in debugging. There's no need to override this method.

Typically when developing HTTP-based servlets, a Servlet Developer will only concern himself with the `doGet` and `doPost` methods. The other methods are considered to be methods for use by programmers very familiar with HTTP programming.

Using the `HttpServletRequest` interface, write code to retrieve HTML form parameters from the request, retrieve HTTP request header information, or retrieve cookies from the request.

HTTP Protocol Parameters.

Request parameters for the servlet are the strings sent by the client to a servlet container as part of its request. When the request is an `HttpServletRequest` object, the container populates the parameters from the URI query string and POST-ed data.

The parameters are stored as a set of name-value pairs. Multiple parameter values CAN exist for any given parameter name. The following methods of the `ServletRequest` interface are available to access parameters:

- `getParameter`

  Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

  You should only use this method when you are sure the parameter has only ONE value. If the parameter might have MORE than one value, use `getParameterValues(String)`.

  If you use this method with a multivalued parameter, the value returned is equal to the FIRST value in the array returned by `getParameterValues`.

  If the parameter data was sent in the request body, such as occurs with an HTTP `POST` request, then reading the body directly via `getInputStream()` or `getReader()` can interfere with the execution of this method.

- `getParameterNames`

  Returns an `Enumeration` of `String` objects containing the names of the parameters contained in this request. If the request has no parameters, the method returns an EMPTY `Enumeration`.

- `getParameterValues`

  Returns an array of `String` objects containing all of the values the given request parameter has, or `null` if the parameter does not exist. If the parameter has a single value, the array has a length of 1.

- `getParameterMap`

  Returns an immutable `java.util.Map` containing parameter names as keys and parameter values as map values. The keys in the parameter map are of type `String`. The values in the parameter map are of type `String` array.

```
public interface ServletRequest {

        public java.lang.String getParameter(java.lang.String name);
        public java.util.Enumeration getParameterNames();
        public java.lang.String[] getParameterValues(java.lang.String name);
        public java.util.Map getParameterMap();

}
```

The `getParameterValues` method returns an array of `String` objects containing all the parameter values associated with a parameter name. The value returned from the `getParameter` method must be the FIRST value in the array of `String` objects returned by `getParameterValues`. The `getParameterMap` method returns a `java.util.Map` of the parameter of the request, which contains names as keys and parameter values as map values.

```
public void doPost(HttpServletRequest request, HttpServletResponse res)
                throws IOException, ServletException {
        Enumeration e = request.getParameterNames();
        PrintWriter out = res.getWriter ();
        while (e.hasMoreElements()) {
                String name = (String)e.nextElement();
                String value = request.getParameter(name);
                out.println(name + " = " + value);
        }
}
```

Data from the query string and the post body are aggregated into the request parameter set. Query string data is presented BEFORE post body data. For example, if a request is made with a query string of `a=hello` and a post body of `a=goodbye&a=world`, the resulting parameter set would be ordered `a=(hello, goodbye, world)`.

The following are the conditions that must be met before post FORM data will be populated to the parameter set:

1. The request is an HTTP or HTTPS request.

2. The HTTP method is `POST`.

3. The content type is `application/x-www-form-urlencoded`.

4. The servlet has made an initial call of any of the 'getParameter' family of methods on the request object.

If the conditions are not met and the post form data is not included in the parameter set, the post data must still be available to the servlet via the request object's input stream. If the conditions are met, post form data will no longer be available for reading directly from the request object's input stream.

Headers.

A servlet can access the headers of an HTTP request through the following methods of the `HttpServletRequest` interface:

- `getHeader`

  Returns the value of the specified request header as a `String`. If the request did not include a header of the specified name, this method returns `null`. If there are multiple headers with the same name, this method returns the first head in the request. The header name is case insensitive. You can use this method with any request header.

- `getHeaders`

Returns all the values of the specified request header as an `Enumeration` of `String` objects.

Some headers, such as `Accept-Language` can be sent by clients as several headers each with a different value rather than sending the header as a comma separated list. If the request did not include any headers of the specified name, this method returns an EMPTY `Enumeration`. The header name is case INSENSITIVE. You can use this method with any request header.

- `getHeaderNames`

  Returns an enumeration of all the header names this request contains. If the request has no headers, this method returns an empty enumeration.

The `getHeader` method returns a header given the name of the header. There can be multiple headers with the same name, e.g. `Cache-Control` headers, in an HTTP request. If there are multiple headers with the same name, the `getHeader` method returns the first header in the request. The `getHeaders` method allows access to all the header values associated with a particular header name, returning an `Enumeration` of `String` objects.

```
public interface HttpServletRequest {

        public java.lang.String getHeader(java.lang.String name);
        public java.util.Enumeration getHeaders(java.lang.String name);
        public java.util.Enumeration getHeaderNames();

}
```

Headers may contain `String` representations of `int` or `Date` data. The following convenience methods of the `HttpServletRequest` interface provide access to header data in a one of these formats:

- `getIntHeader`

  Returns the value of the specified request header as an int. If the request does not have a header of the specified name, this method returns -1. If the header cannot be converted to an integer, this method throws a `NumberFormatException`.

  The header name is case INSENSITIVE.

- `getDateHeader`

  Returns the value of the specified request header as a long value that represents a `Date` object. Use this method with headers that contain dates, such as `If-Modified-Since`.

  The date is returned as the number of milliseconds since January 1, 1970 GMT. The header name is case insensitive.

  If the request did not have a header of the specified name, this method returns -1. If the header can't be converted to a date, the method throws an `IllegalArgumentException`.

If the `getIntHeader` method cannot translate the header value to an `int`, a `NumberFormatException` is thrown. If the `getDateHeader` method cannot translate the header to a `Date` object, an `IllegalArgumentException` is thrown.

```
public interface HttpServletRequest {

        public int getIntHeader(java.lang.String name);
        public long getDateHeader(java.lang.String name);

}
```

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration e = request.getHeaderNames();
        while (e.hasMoreElements()) {
                String name = (String)e.nextElement();
                String value = request.getHeader(name);
                out.println(name + " = " + value);
        }
}
```

Cookies.

The `HttpServletRequest` interface provides the `getCookies` method to obtain an array of cookies that are present in the request. This method returns `null` if no cookies were sent.

The cookies are data sent from the client to the server on every request that the client makes. Typically, the only information that the client sends back as part of a cookie is the cookie name and the cookie value. Other cookie attributes that can be set when the cookie is sent to the browser, such as comments, are not typically returned. Several cookies might have the same name but different path attributes.

```
public interface HttpServletRequest {

        public Cookie[] getCookies();

}
```

```
package javax.servlet.http;

public class Cookie implements java.lang.Cloneable {
        ...
        public Cookie(java.lang.String name, java.lang.String value);
        public java.lang.String getName();
        public java.lang.String getPath();
        public java.lang.String getValue();
        ...
}
```

Using the `HttpServletResponse` interface, write code to set an HTTP response header, set the content type of the response, acquire a text stream for the response, acquire a binary stream for the response, redirect an HTTP request to another URL, or add cookies to the response.

Headers.

A servlet can set headers of an HTTP response via the following methods of the `HttpServletResponse` interface:

- `setHeader`

  Sets a response header with the given name and value. If the header had already been set, the new value OVERWRITES the previous one. The `containsHeader` method can be used to test for the presence of a header before setting its value.

- `addHeader`

Adds a response header with the given name and value. This method allows response headers to have multiple values.

```
public interface HttpServletResponse extends javax.servlet.ServletResponse {

        public void setHeader(java.lang.String name, java.lang.String value);
        public void addHeader(java.lang.String name, java.lang.String value);

}
```

The `setHeader` method sets a header with a given name and value. A previous header is replaced by the new header. Where a set of header values exist for the name, the values are cleared and replaced with the new value.

The `addHeader` method adds a header value to the set with a given name. If there are no headers already associated with the name, a new set is created.

Headers may contain data that represents an `int` or a `Date` object. The following convenience methods of the `HttpServletResponse` interface allow a servlet to set a header using the correct formatting for the appropriate data type:

- `setIntHeader`

  Sets a response header with the given name and integer value. If the header had already been set, the new value overwrites the previous one. The `containsHeader` method can be used to test for the presence of a header before setting its value.

- `setDateHeader`

  Sets a response header with the given name and date-value. The date is specified in terms of milliseconds since the epoch. If the header had already been set, the new value overwrites the previous one. The `containsHeader` method can be used to test for the presence of a header before setting its value.

- `addIntHeader`

  Adds a response header with the given name and integer value. This method allows response headers to have multiple values.

- `addDateHeader`

  Adds a response header with the given name and date-value. The date is specified in terms of milliseconds since the epoch. This method allows response headers to have multiple values.

```
public interface HttpServletResponse extends javax.servlet.ServletResponse {

        public void setIntHeader(java.lang.String name, int value);
        public void setDateHeader(java.lang.String name, long date);
        public void addIntHeader(java.lang.String name, int value);
        public void addDateHeader(java.lang.String name, long date);

}
```

To be successfully transmitted back to the client, headers must be set before the response is committed. Headers set after the response is committed will be IGNORED by the servlet container.

Content type.

The charset for the MIME body response can be specified explicitly using the `setContentType(String)` method. Explicit specifications take precedence over implicit specifications. If no charset is specified, ISO-8859-1 will be used. The `setContentType` method MUST be called BEFORE `getWriter` and BEFORE committing the response for the character encoding to be used.

There are 2 ways to define content type:

- `ServletResponse.setContentType(String);`
- `HttpServletResponse.setHeader("Content-Type", "text/html");`

Acquire a text stream.

To send CHARACTER data, use the `PrintWriter` object returned by `ServletResponse.getWriter()`

```
public interface ServletResponse {

        public java.io.PrintWriter getWriter() throws IOException;

}
```

Returns a `PrintWriter` object that can send character text to the client. The `PrintWriter` uses the character encoding returned by `getCharacterEncoding()`. Calling `flush()` on the `PrintWriter` commits the response.

Either this method or `getOutputStream()` may be called to write the body, NOT BOTH.

Acquire a binary stream.

`ServletResponse.getOutputStream()` provides an output stream for sending BINARY data to the client. A `ServletOutputStream` object is normally retrieved via this method.

```
public interface ServletResponse {

        public ServletOutputStream getOutputStream() throws IOException;

}
```

The servlet container does NOT encode the binary data.

Calling `flush()` on the `ServletOutputStream` commits the response. Either this method or `getWriter()` may be called to write the body, NOT BOTH.

Redirect an HTTP request to another URL.

The `HttpServletResponse.sendRedirect` method will set the appropriate headers and content body to redirect the client to a different URL. It is legal to call this method with a relative URL path, however the underlying container must translate the relative path to a fully qualified URL for transmission back to the client. If a partial URL is given and, for whatever reason, cannot be converted into a valid URL, then this method must throw an `IllegalArgumentException`.

```
public interface HttpServletResponse extends javax.servlet.ServletResponse {

        public void sendRedirect(java.lang.String location) throws IOException;

}
```

Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container must convert the relative URL to an absolute URL before sending the response to the client. If the location is relative without a leading '/' the container interprets it as relative to the current request URI. If the location is relative with a leading '/' the container interprets it as relative to the servlet container root.

If the response has already been committed, this method throws an `IllegalStateException`. After using this method, the response should be considered to be committed and should not be written to.

This method will have the side effect of committing the response, if it has not already been committed, and terminating it. No further output to the client should be made by the servlet after these methods are called. If data is written to the response after this method are called, the data is ignored.

If data has been written to the response buffer, but not returned to the client (i.e. the response is not committed), the data in the response buffer must be cleared and replaced with the data set by these methods. If the response is committed, this method must throw an `IllegalStateException`.

Add cookies to the response.

The servlet sends cookies to the browser by using the `HttpServletResponse.addCookie(Cookie)` method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.

```
public interface HttpServletResponse extends javax.servlet.ServletResponse {

        public void addCookie(Cookie cookie);

}
```

Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.

Describe the purpose and event sequence of the servlet life cycle: (1) servlet class loading, (2) servlet instantiation, (3) call the init method, (4) call the `service` method, and (5) call `destroy` method.

A servlet is managed through a well defined life cycle that defines how it is loaded and instantiated, is initialized, handles requests from clients, and is taken out of service. This life cycle is expressed in the API by the `init`, `service`, and `destroy` methods of the `javax.servlet.Servlet` interface that all servlets must implement directly or indirectly through the `GenericServlet` or `HttpServlet` abstract classes.

Servlet class loading and instantiation.

The servlet container is responsible for loading and instantiating servlets. The loading and instantiation can occur when the container is started, or delayed until the container determines the servlet is needed to service a request.

When the servlet engine is started, needed servlet classes must be located by the servlet container. The servlet container loads the servlet class using normal Java class loading facilities. The loading may be from a local file system, a remote file system, or other network services.

After loading the `Servlet` class, the container instantiates it for use.

Servlet class initialization.

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data, initialize costly resources (such as JDBC connections), and perform other one-time activities. The container initializes the servlet instance by calling the `init` method of the `Servlet` interface with a unique (per servlet declaration) object implementing the `ServletConfig` interface.

```
public interface Servlet {

        public void init(ServletConfig config) throws ServletException;

}
```

This configuration object allows the servlet to access name-value initialization parameters from theWeb application's configuration information. The configuration object also gives the servlet access to an object (implementing the `ServletContext` interface) that describes the servlet's runtime environment.

During initialization, the servlet instance can throw an `UnavailableException` or a `ServletException`. In this case, the servlet must not be placed into active service and must be released by the servlet container. The `destroy` method is not called as it is considered unsuccessful initialization.

A new instance may be instantiated and initialized by the container after a failed initialization. The exception to this rule is when an `UnavailableException` indicates a minimum time of unavailability, and the container must wait for the period to pass before creating and initializing a new servlet instance.

Request handling.

After a servlet is properly initialized, the servlet container may use it to handle client requests. Requests are represented by request objects of type `ServletRequest`. The servlet fills out response to requests by calling methods of a provided object of type `ServletResponse`. These objects are passed as parameters to the `service` method of the `Servlet` interface.

```
public interface Servlet {

        public void service(ServletRequest req, ServletResponse res)
                throws ServletException, IOException;

}
```

In the case of an HTTP request, the objects provided by the container are of types `HttpServletRequest` and `HttpServletResponse`.

```
public abstract class HttpServlet extends javax.servlet.GenericServlet
        implements java.io.Serializable {

        protected void service(HttpServletRequest req, HttpServletResponse res)
                throws ServletException, IOException;

}
```

Note that a servlet instance placed into service by a servlet container may handle NO requests during its lifetime.

End of service.

The servlet container is not required to keep a servlet loaded for any particular period of time. A servlet instance may be kept active in a servlet container for a period of milliseconds, for the lifetime of the servlet container (which could be a number of days, months, or years), or any amount of time in between.

When the servlet container determines that a servlet should be removed from service, it calls the `destroy` method of the `Servlet` interface to allow the servlet to release any resources it is using and save any persistent state. For example, the container may do this when it wants to conserve memory resources, or when it is being shut down.

```
public interface Servlet {

        public void destroy();

}
```

Before the servlet container calls the `destroy` method, it must allow any threads that are currently running in the `service` method of the servlet to complete execution, or exceed a server-defined time limit.

Once the `destroy` method is called on a servlet instance, the container may not route other requests to that instance of the servlet. If the container needs to enable the servlet again, it must do so with a new instance of the servlet's class.

After the `destroy` method completes, the servlet container must release the servlet instance so that it is eligible for garbage collection.

Chapter 2. The Structure and Deployment of Web Applications

Construct the file and directory structure of a Web Application that may contain (a) static content, (b) JSP pages, (c) servlet classes, (d) the deployment descriptor, (e) tag libraries, (f) JAR files, and (g) Java class files; and describe how to protect resource files from HTTP access.

A Web application exists as a structured hierarchy of directories. The root of this hierarchy serves as the document root for files that are part of the application. For example, for a Web application with the context path `/catalog` in a Web container, the `index.html` file at the base of the Web application hierarchy can be served to satisfy a request from `/catalog/index.html`.

A special directory exists within the application hierarchy named "WEB-INF". This directory contains all things related to the application that aren't in the document root of the application. The `WEB-INF` node is NOT part of the public document tree of the application. NO file contained in the `WEB-INF` directory may be served directly to a client by the container. However, the contents of the `WEB-INF` directory are visible to servlet code using the `getResource` and `getResourceAsStream` method calls on the `ServletContext`, and may be exposed using the `RequestDispatcher` calls. Hence, if the Application Developer needs access, from servlet code, to application specific configuration information that he does not wish to be exposed directly to the Web client, he may place it under this directory. Since requests are matched to resource mappings in a case-sensitive manner, client requests for `/WEB-INF/foo`, `/WEb-iNf/foo`, for example, should not result in contents of the Web application located under `/WEB-INF` being returned, nor any form of directory listing thereof.

The contents of the `WEB-INF` directory are:

- The `/WEB-INF/web.xml` deployment descriptor.
- The `/WEB-INF/classes/` directory for servlet and utility classes. The classes in this directory must be available to the application class loader.
- The `/WEB-INF/lib/*.jar` area for Java ARchive files. These files contain servlets, beans, and other utility classes useful to the Web application. The Web application class loader must be able to load classes from any of these archive files.

The Web application class loader must load classes from the `WEB-INF/classes` directory first, and then from library JARs in the `WEB-INF/lib` directory. Also, any requests from the client to access the resources in `WEB-INF/` directory MUST be returned with a `SC_NOT_FOUND` (404) response.

Web applications can be packaged and signed into a Web ARchive format (WAR) file using the standard Java archive tools. For example, an application for issue tracking might be distributed in an archive file called `issuetrack.war`.

When packaged into such a form, a `META-INF` directory will be present which contains information useful to Java archive tools. This directory MUST NOT be directly served as content by the container in response to a Web client's request, though its contents are visible to servlet code via the `getResource` and `getResourceAsStream` calls on the `ServletContext`. Also, any requests to access the resources in `META-INF` directory must be returned with a `SC_NOT_FOUND` (404) response.

Tag extensions written in JSP using tag files can be placed in one of two locations. The first possibility is in the `/META-INF/tags/` directory (or a subdirectory of `/META-INF/tags/`) in a JAR file installed in the `/WEB-INF/lib/` directory of the web application. Tags placed here are typically part of a reusable library of tags that can be easily dropped into any web application.

The second possibility is in the `/WEB-INF/tags/` directory (or a subdirectory of `/WEB-INF/tags/`) of the web application. Tags placed here are within easy reach and require little packaging. Only files with a `.tag` or `.tagx` extension are recognized by the container to be tag files.

Tag files that appear in any other location are not considered tag extensions and must be ignored by the JSP container. For example, a tag file that appears in the root of a web application would be treated as content to be served.

The following is a listing of all the files in a sample Web application:

```
/index.html
/howto.jsp
/feedback.jsp
/images/banner.gif
/images/jumping.gif
/WEB-INF/web.xml
/WEB-INF/lib/jspbean.jar
/WEB-INF/lib/jstl.jar
/WEB-INF/jsp/example-taglib.tld
/WEB-INF/jsp/debug-taglib.tld
/WEB-INF/jsp2/jsp2-example-taglib.tld
/WEB-INF/tags/displayProducts.tag
/WEB-INF/tags/helloWorld.tag
/WEB-INF/tags/panel.tag
/WEB-INF/tags/xhtmlbasic.tag
/WEB-INF/classes/com/mycorp/servlets/MyServlet.class
/WEB-INF/classes/com/mycorp/util/MyUtils.class
```

Describe the purpose and semantics for each of the following deployment descriptor elements: `error-page`, `init-param`, `mime-mapping`, `servlet`, `servlet-class`, `servlet-mapping`, `servlet-name`, and `welcome-file`.

Error pages.

To allow developers to customize the appearance of content returned to a Web client when a servlet generates an error, the deployment descriptor defines a list of error page descriptions. The syntax allows the configuration of resources to be returned by the container either when a servlet or filter calls `sendError` on the response for specific status codes, or if the servlet generates an exception or error that propagates to the container.

If the `sendError` method is called on the response, the container consults the list of error page declarations for the Web application that use the status-code syntax and attempts a match. If there is a match, the container returns the resource as indicated by the location entry.

A servlet or filter may throw the following exceptions during processing of a request:

- runtime exceptions or errors
- `ServletExceptions` or subclasses thereof
- `IOExceptions` or subclasses thereof

The Web application may have declared error pages using the `exception-type` element. In this case the container matches the exception type by comparing the exception thrown with the list of `error-page` definitions that use the `exception-type` element. A match results in the container returning the resource indicated in the location entry. The closest match in the class heirarchy wins.

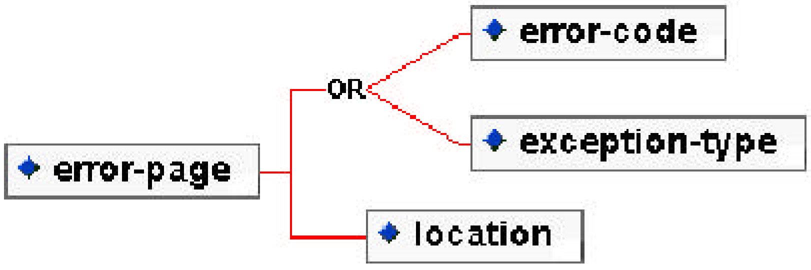
If no `error-page` declaration containing an `exception-type` fits using the class-heirarchy match, and the exception thrown is a `ServletException` or subclass thereof, the container extracts the wrapped exception, as defined by the `ServletException.getRootCause` method. A second pass is made over the error page declarations, again attempting the match against the error page declarations, but using the wrapped exception instead.

`error-page` declarations using the `exception-type` element in the deployment descriptor MUST be unique up to the class name of the `exception-type`. Similarly, `error-page` declarations using the `status-code` element MUST be unique in the deployment descriptor up to the status code.

If a servlet generates an error that is not handled by the error page mechanism as described above, the container must ensure to send a response with status 500.

The default servlet and container will use the `sendError` method to send 4xx and 5xx status responses, so that the error mechanism may be invoked. The default servlet and container will use the `setStatus` method for 2xx and 3xx responses and will not invoke the error page mechanism.

You can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any Web component and a Web resource. To set up the mapping, you add an `error-page` element to the Web application deployment descriptor.



```
<!--
The error-page element contains a mapping between an error code or
exception type to the path of a resource in the web application
-->

<!ELEMENT error-page ((error-code | exception-type), location)>
```

The `error-page` contains a mapping between an error code or an exception type to the path of a resource in the Web application. The sub-element `exception-type` contains a fully qualified class name of a Java exception type. The sub-element `location` element contains the location of the resource in the web application relative to the root of the web application. The value of the location MUST have a leading '/'.

```
<web-app>
        ...
        <error-page>
                <exception-type>exception.BookNotFoundException</exception-type>
                <location>/errorpage.html</location>
        </error-page>
</web-app>
```

```
<web-app>
        ...
        <error-page>
                <exception-type>exception.OrderException</exception-type>
                <location>/errorpage.jsp</location>
        </error-page>
</web-app>
```

```
<web-app>
        ...
        <error-page>
                <error-code>404</error-code>
                <location>/404.html</location>
        </error-page>
</web-app>
```

Init parameters.

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data, initialize costly resources (such as JDBC connections), and perform other one-time activities. The container initializes the servlet instance by calling the `init` method of the `Servlet` interface with a unique (per servlet declaration) object implementing the `ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters from the Web application's configuration information.

```
<!--
The init-param element contains a name/value pair as an
initialization param of the servlet
-->

<!ELEMENT init-param (param-name, param-value, description?)>
```

A servlet configuration object used by a servlet container to pass information to a servlet during initialization.

- `getInitParameter`

  Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist.

- `getInitParameterNames`

Returns the names of the servlet's initialization parameters as an `Enumeration` of `String` objects, or an EMPTY `Enumeration` if the servlet has no initialization parameters.

```
public interface ServletConfig {

        public java.lang.String getInitParameter(java.lang.String name);
        public java.util.Enumeration getInitParameterNames();

}
```

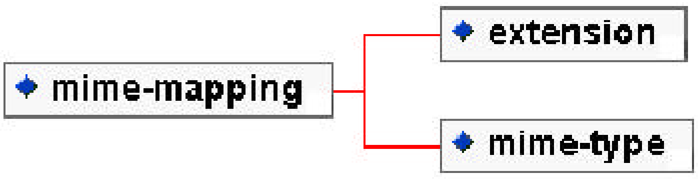
```
<web-app>
        ...
        <servlet>
                <servlet-name>catalog</servlet-name>
                <servlet-class>com.mycorp.CatalogServlet</servlet-class>
                <init-param>
                        <param-name>bgcolor</param-name>
                        <param-value>yellow</param-value>
                </init-param>
        </servlet>
        ...
</web-app>
```

```
...
private String bgcolor;

public void init(ServletConfig config) throws ServletException {
        super.init(config);
        try {
                bgcolor = config.getInitParameter("bgcolor");
                System.out.println("bgcolor: " + bgcolor);
        } catch (Exception e) {
                System.out.println("error: " + e.toString());
        }
}
```

MIME mapping.

The `mime-mapping` defines a mapping between an extension and a mime type (example: "text/plain"). The `extension` element contains a string describing an extension, such as "txt".



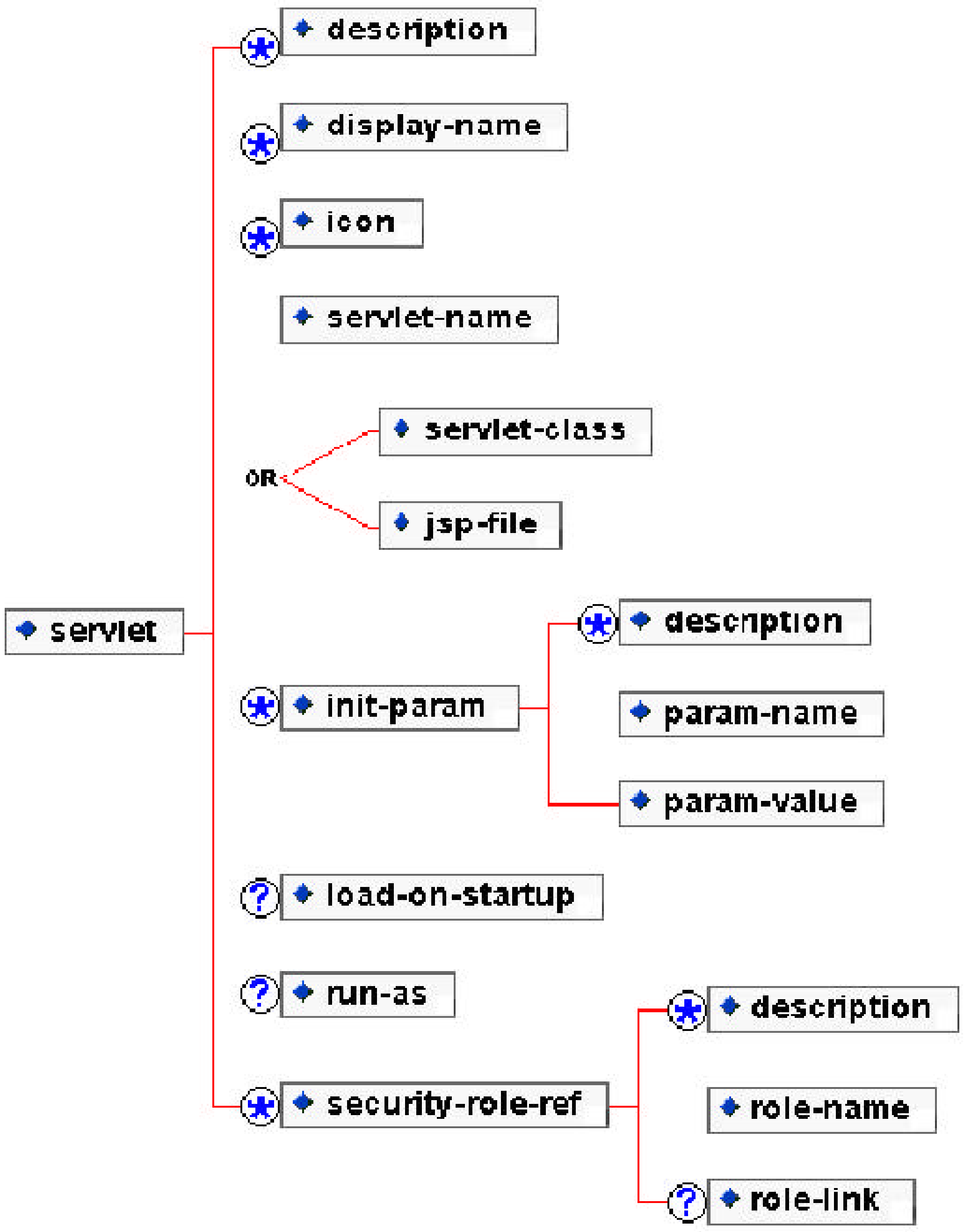
```
<!--
The mime-mapping element defines a mapping between an extension and
a mime type.
-->

<!ELEMENT mime-mapping (extension, mime-type)>
```

```
<web-app>
        ...
        <mime-mapping>
                <extension>pdf</extension>
                <mime-type>application/pdf</mime-type>
        </mime-mapping>
        ...
</web-app>
```

Servlet.

The `servlet` is used to declare a servlet. It contains the declarative data of a servlet. The `jsp-file` element contains the full path to a JSP file within the web application beginning with a "/". If a `jsp-file` is specified and the `load-on-startup` element is present, then the JSP should be precompiled and loaded. The `servlet-name` element contains the canonical name of the servlet. Each servlet name is UNIQUE within the web application. The element content of `servlet-name` MUST NOT be empty. The `servlet-class` contains the fully qualified class name of the servlet. The `run-as` element specifies the identity to be used for the execution of a component. It contains an optional description, and the name of a security role specified by the `role-name` element. The element `load-on-startup` indicates that this servlet should be loaded (instantiated and have its `init()` called) on the startup of the Web application. The element content of this element must be an INTEGER indicating the order in which the servlet should be loaded. If the value is a negative integer, or the element is not present, the container is free to load the servlet whenever it chooses. If the value is a positive integer or 0, the container must load and initialize the servlet as the application is deployed. The container must guarantee that servlets marked with lower integers are loaded before servlets marked with higher integers. The container may choose the order of loading of servlets with the same `load-on-startup` value. The `security-role-ref` element declares the security role reference in a component's or in a deployment component's code. It consists of an optional `description`, the security role name used in the code (`role-name`), and an optional link to a security role (`role-link`). If the security role is not specified, the deployer must choose an appropriate security role.



The `servlet-class` element contains the fully qualified class name of the servlet.

```
<!--
The servlet element contains the declarative data of a
servlet.
If a jsp-file is specified and the load-on-startup element is
present, then the JSP should be precompiled and loaded.
-->

<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
        (servlet-class|jsp-file), init-param*, load-on-startup?,
        security-role-ref*)>
```

Servlet mapping.

The `servlet-mapping` defines a mapping between a servlet and a URL pattern.

The path used for mapping to a servlet is the request URL from the request object minus the context path and the path parameters. The URL path mapping rules below are used in order. The first successful match is used with no further matches attempted:

1. The container will try to find an exact match of the path of the request to the path of the servlet. A successful match selects the servlet.

2. The container will recursively try to match the longest path-prefix. This is done by stepping down the path tree a directory at a time, using the '/' character as a path separator. The longest match determines the servlet selected.

3. If the last segment in the URL path contains an extension (e.g. .jsp), the servlet container will try to match a servlet that handles requests for the extension. An extension is defined as the part of the last segment after the last '.' character.

4. If neither of the previous three rules result in a servlet match, the container will attempt to serve content appropriate for the resource requested. If a "default" servlet is defined for the application, it will be used.

The container MUST use case-sensitive string comparisons for matching.

In the Web application deployment descriptor, the following syntax is used to define mappings:

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

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

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

- All other strings are used for exact matches only.

Request_URI = Context_Path [1] + Servlet_Path [2] + Path_Info [3] + Query_String [4]

http://server.com/my_app_context[1]/catalog[2]/product[3]?mode=view[4]

Context path.

Specifies the path prefix associated with a web application mapping. For a default application (rooted at the base of the web server's URL namespace), the context path is an empty string. For a non-default application, the context path starts with a forward slash ('/') but does not end with one. For example, `/my_app_context` maps requests that include `/my_app_context` to the `my_app_context` application. The `HttpServletRequest.getContextPath()` method returns a string representing the context path.

Servlet path.

Specifies the portion of the URL that matches the servlet mapping. This starts with a slash ('/'). The `HttpServletRequest.getServletPath()` method returns a string representing the servlet path.

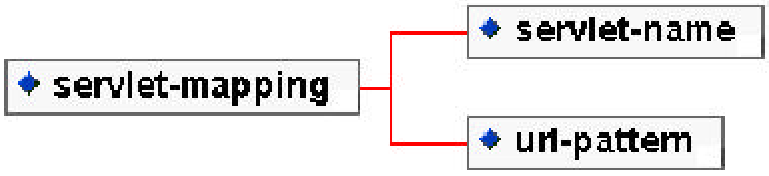
Path information.

Comprises the remaining portion of the request path prior to query string parameters. The `HttpServletRequest.getPathInfo()` method returns a string representing the remainder of the path.

Query string.

Is contained in the request URL after the path. The `HttpServletRequest.getQueryString()` method returns `null` if the URL does not have a query string. Same as the value of the CGI variable

QUERY_STRING.



```
<web-app>
        ...
        <servlet-mapping>
                <servlet-name>catalog</servlet-name>
                <url-pattern>/catalog/*</url-pattern>
        </servlet-mapping>
        ...
</web-app>
```

```
<!--
The servlet-mapping element defines a mapping between a servlet and
a url pattern
-->

<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
```

Welcome files.

Web Application developers can define an ordered list of partial URIs called welcome files in the Web application deployment descriptor.

The purpose of this mechanism is to allow the deployer to specify an ordered list of partial URIs for the container to use for appending to URIs when there is a request for a URI that corresponds to a directory entry in the WAR not mapped to a Web component. This kind of request is known as a valid partial request.

The use for this facility is made clear by the following common example: A welcome file of `'index.html'` can be defined so that a request to a URL like `host:port/webapp/directory/`, where `'directory'` is an entry in the WAR that is not mapped to a servlet or JSP page, is returned to the client as `'host:port/webapp/directory/index.html'`.

If a Web container receives a valid partial request, the Web container must examine the welcome file list defined in the deployment descriptor. The welcome file list is an ordered list of partial URLs with no trailing or leading '/'. The Web server must append each welcome file in the order specified in the deployment descriptor to the partial request and check whether a static resource or servlet in the WAR is mapped to that request URI. The Web container must send the request to the first resource in the WAR that matches.

If no matching welcome file is found in the manner described, the container may handle the request in a manner it finds suitable. For some configurations this may mean returning a directory listing or for others returning a 404 response.

Consider a Web application where:

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

```
/foo/index.html
/foo/default.jsp
/foo/orderform.html
/foo/home.gif
/catalog/default.jsp
/catalog/products/shop.jsp
/catalog/products/register.jsp
```

A request URI of `/foo` will be redirected to a URI of `/foo/`.

A request URI of `/foo/` will be returned as `/foo/index.html`.

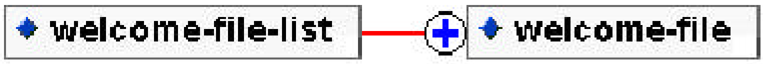
A request URI of `/catalog` will be redirected to a URI of `/catalog/`.

A request URI of `/catalog/` will be returned as `/catalog/default.jsp`.

A request URI of `/catalog/index.html` will cause a 404 (not found).

A request URI of `/catalog/products` will be redirected to a URI of `/catalog/products/`.

A request URI of `/catalog/products/` will be passed to the "default" servlet, if any. If no "default" servlet is mapped, the request may cause a 404 (not found), may cause a directory listing including `shop.jsp` and `register.jsp`, or may cause other behavior defined by the container.



```
<web-app>
        ...
        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
                <welcome-file>index.html</welcome-file>
                <welcome-file>index.htm</welcome-file>
        </welcome-file-list>
        ...
</web-app>
```

The Java 2 Platform, Enterprise Edition defines a naming environment that allows applications to easily access resources and external information without explicit knowledge of how the external information is named or organized. As servlets are an integral component type of J2EE technology, provision has been made in the Web application deployment descriptor for specifying information allowing a servlet to obtain references to resources and enterprise beans. The deployment elements that contain this information are: `env-entry`, `ejb-ref`, `ejb-local-ref`, `resource-ref`, `resource-env-ref`.

NOTE, you NEED TO KNOW for the SCWCD Exam how to code the deployment descriptor and what it means for all of the following tags:

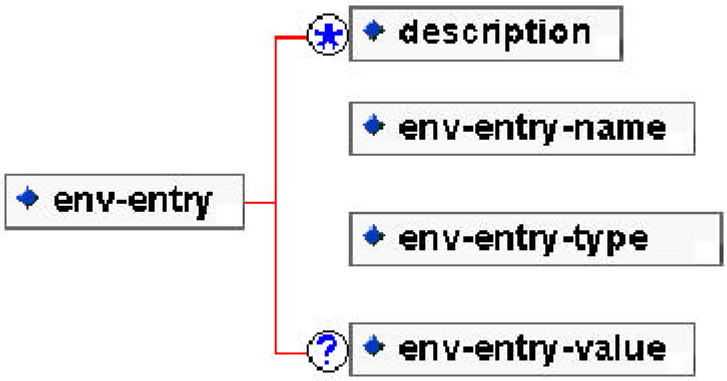
- `env-entry`
- `ejb-ref`

- `ejb-local-ref`
- `resource-ref`
- `resource-env-ref`

`env-entry` element

The `env-entry` declares an application's environment entry. The sub-element `env-entry-name` contains the name of a deployment component's environment entry. The name is a JNDI name relative to the `java:comp/env` context. The name MUST be unique within a deployment component. The `env-entry-type` contains the fully-qualified Java type of the environment entry value that is expected by the application's code. The sub-element `env-entry-value` designates the value of a deployment component's environment entry. The value MUST be a String that is valid for the constructor of the specified type that takes a single `String` as a parameter, or a single character for `java.lang.Character`.

The following are the legal values of `env-entry-type`: java.lang.Boolean, `java.lang.Byte`, `java.lang.Character`, `java.lang.String`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`.

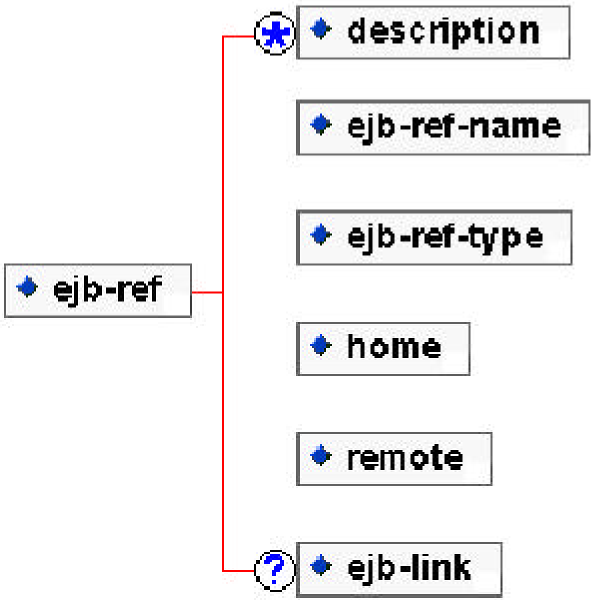


Example:

```
<env-entry>
        <env-entry-name>test/MyEnv1</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>10</env-entry-value>
</env-entry>
<env-entry>
        <env-entry-name>test/MyEnv2</env-entry-name>
        <env-entry-type>java.lang.Boolean</env-entry-type>
        <env-entry-value>true</env-entry-value>
</env-entry>
```

`ejb-ref` element

The `ejb-ref` declares the reference to an enterprise bean's home. The `ejb-ref-name` specifies the name used in the code of the deployment component that is referencing the enterprise bean. The `ejb-ref-type` is the expected type of the referenced enterprise bean, which is either Entity or Session. The `home` defines the fully qualified name of the the referenced enterprise bean's home interface. The `remote` defines the fully qualified name of the referenced enterprise bean's remote interface. The `ejb-link` specifies that an EJB reference is linked to the enterprise bean.

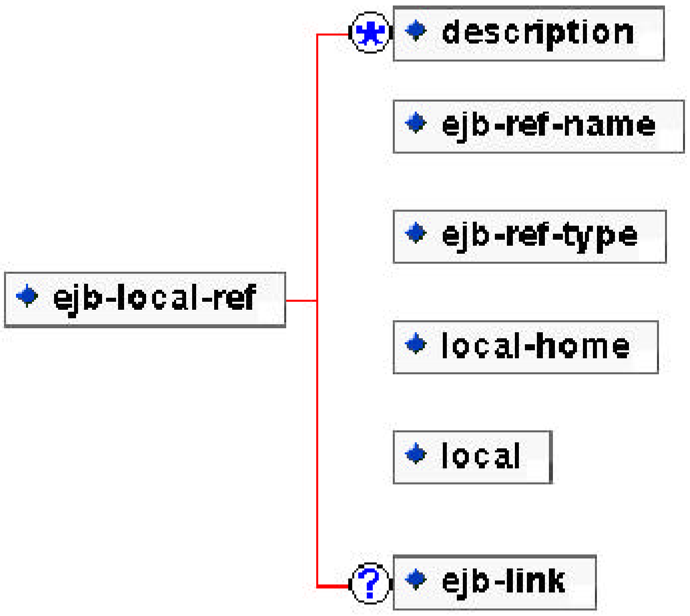


Example:

```
<ejb-ref>
        <ejb-ref-name>ejb/RemoteSession</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.example.RemoteSessionHome</home>
        <remote>com.example.RemoteSession</remote>
</ejb-ref>
```

`ejb-local-ref` element

The `ejb-local-ref` declares the reference to the enterprise bean's local home. The `local-home` defines the fully qualified name of the enterprise bean's local home interface. The `local` defines the fully qualified name of the enterprise bean's local interface.

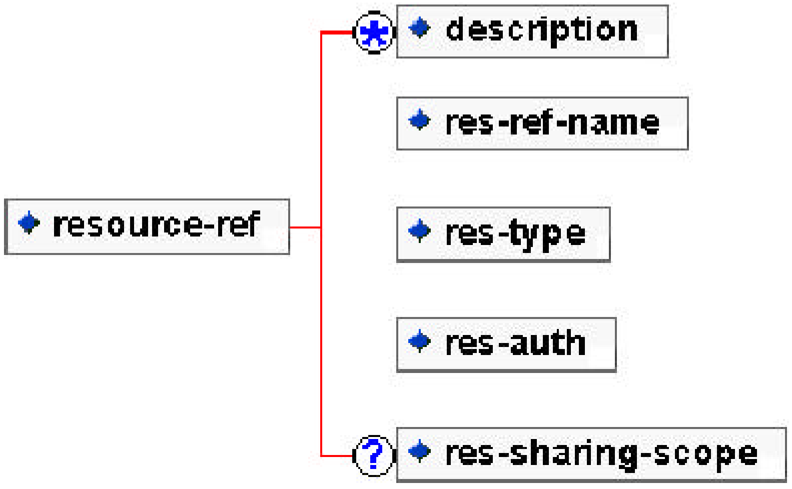


Example:

```
<ejb-local-ref>
        <ejb-ref-name>ejb/LocalSession</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local-home>com.example.LocalSessionHome</local-home>
        <local>com.example.LocalSession</local>
        <ejb-link>LocalSession</ejb-link>
</ejb-local-ref>
```

`resource-ref` element

The `resource-ref` contains the declaration of a deployment component's reference to the external resource. The `res-ref-name` specifies the name of a resource manager connection factory reference. The name is a JNDI name relative to the `java:comp/env` context. The name MUST be unique within a deployment file. The `res-type` element specifies the type of the data source. The type is the fully qualified Java language class or the interface expected to be implemented by the data source. The `res-auth` specifies whether the deployment component code signs on programmatically to the resource manager, or whether the container will sign on to the resource manager on behalf of the deployment component. In the latter case, the container uses the information supplied by the deployer. The `res-sharing-scope` specifies whether connections obtained through the given resource manager connection factory reference can be shared. The value, if specified, MUST be either `Shareable` or `Unshareable`.

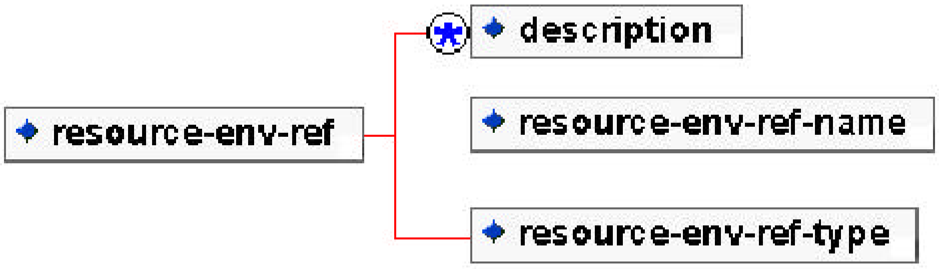


Example:

```
<resource-ref>
        <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

`resource-env-ref` element

The `resource-env-ref` contains the deployment component's reference to the administered object associated with a resource in the deployment component's environment. The `resource-env-ref-name` specifies the name of the resource environment reference. The value is the environment entry name used in the deployment component code and is a JNDI name relative to the `java:comp/env` context and MUST be unique within the deployment component. The `resource-env-ref-type` specifies the type of the resource environment reference. It is the fully qualified name of a Java language class or the interface.



Example:

```
<resource-env-ref>
        <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
        <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

Construct the correct structure for each of the following deployment descriptor elements: `error-page`, `init-param`, `mime-mapping`, `servlet`, `servlet-class`, `servlet-mapping`, `servlet-name`, and `welcome-file`.

Error pages.

```
<!--
The error-page element contains a mapping between an error code or
exception type to the path of a resource in the web application
-->

<!ELEMENT error-page ((error-code | exception-type), location)>
```

Init parameters.

```
<!--
The init-param element contains a name/value pair as an
initialization param of the servlet
-->

<!ELEMENT init-param (param-name, param-value, description?)>
```

MIME mapping.

```
<!--
The mime-mapping element defines a mapping between an extension and
a mime type.
-->

<!ELEMENT mime-mapping (extension, mime-type)>
```

```
<!--
The extension element contains a string describing an
extension. example: "txt"
-->

<!ELEMENT extension (#PCDATA)>
```

```
<!--
The mime-type element contains a defined mime type. example: "text/
plain"
-->

<!ELEMENT mime-type (#PCDATA)>
```

Servlet.

```
<!--
The servlet element contains the declarative data of a
servlet.
If a jsp-file is specified and the load-on-startup element is
present, then the JSP should be precompiled and loaded.
-->

<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
        (servlet-class|jsp-file), init-param*, load-on-startup?,
        security-role-ref*)>
```

```
<!--
The servlet-name element contains the canonical name of the
servlet.
-->

<!ELEMENT servlet-name (#PCDATA)>
```

```
<!--
The servlet-class element contains the fully qualified class name
of the servlet.
-->

<!ELEMENT servlet-class (#PCDATA)>
```

```
<!--
The jsp-file element contains the full path to a JSP file within
the web application.
-->

<!ELEMENT jsp-file (#PCDATA)>
```

Servlet mapping.

```
<!--
The servlet-mapping element defines a mapping between a servlet and
a url pattern

-->
<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
```

Welcome files.

```
<!--
The welcome-file-list contains an ordered list of welcome files
elements.
-->
```

```
<!ELEMENT welcome-file-list (welcome-file+)>
```

env-entry

```
<!--
The env-entry element contains the declaration of an application's
environment entry.
This element is required to be honored on in J2EE compliant servlet
containers.

The env-entry-type element contains the fully qualified Java type of
the environment entry value that is expected by the application
code.

The following are the legal values of env-entry-type:
java.lang.Boolean, java.lang.String, java.lang.Integer,
java.lang.Double, java.lang.Float.
-->

<!ELEMENT env-entry (description?, env-entry-name, env-entry-value?,
env-entry-type)>
```

```
<env-entry>
        <env-entry-name>tableName</env-entry-name>
        <env-entry-value>StockTable</env-entry-value>
        <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

ejb-ref

```
<!--
The ejb-ref element is used for the declaration of a reference to
an enterprise bean's home. The declaration consists of:
- an optional description
- the EJB reference name used in the code of
the web application that's referencing the enterprise bean
- the expected type of the referenced enterprise bean
- the expected home and remote interfaces of the referenced
enterprise bean
- optional ejb-link information, used to specify the referenced
enterprise bean

The ejb-ref element is used to declare a reference to an enterprise
bean.

The ejb-ref-name element contains the name of an EJB
reference. This is the JNDI name that the servlet code uses to get a
reference to the enterprise bean.

The ejb-ref-type element contains the expected type of the
referenced enterprise bean.

The ejb-ref-type element MUST be one of the following:

<ejb-ref-type>Entity</ejb-ref-type>
```

```
<ejb-ref-type>Session</ejb-ref-type>

The home element contains the fully qualified name of the EJB's
home interface

The remote element contains the fully qualified name of the EJB's
remote interface

The ejb-link element is used in the ejb-ref element to specify that
an EJB reference is linked to an EJB in an encompassing Java2
Enterprise Edition (J2EE) application package.
The value of the ejb-link element must be the ejb-name of and EJB in
the J2EE application package.
-->

<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
remote, ejb-link?)>
```

```
<!-- reference on a remote bean without ejb-link-->
<ejb-ref>
        <ejb-ref-name>ejb/MySessionBean</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>org.sample.beans.MySessionBeanHome</home>
        <remote>org.sample.beans.MySessionBean</remote>
</ejb-ref>
```

```
<!-- reference on a remote bean using ejb-link-->
<ejb-ref>
        <ejb-ref-name>ejb/EjbLinkMySessionBean</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>org.sample.beans.MySessionBeanHome</home>
        <remote>org.sample.beans.MySessionBean</remote>
        <ejb-link>MySB.jar#MySessionBean</ejb-link>
</ejb-ref>
```

ejb-local-ref

```
<!--
The ejb-local-ref element is used for the declaration of a reference
to an enterprise bean's local home. The declaration consists of:
- an optional description
- the EJB reference name used in the code of the web application
that's referencing the enterprise bean
- the expected type of the referenced enterprise bean
- the expected local home and local interfaces of the referenced
enterprise bean
- optional ejb-link information, used to specify the referenced
enterprise bean

The ejb-ref-type element contains the expected type of the
referenced enterprise bean.

The ejb-ref-type element MUST be one of the following:

<ejb-ref-type>Entity</ejb-ref-type>
<ejb-ref-type>Session</ejb-ref-type>
-->
```

```
<!ELEMENT ejb-local-ref (description?, ejb-ref-name, ejb-ref-type,
local-home, local, ejb-link?)>
```

```
<!-- reference on a local bean -->
<ejb-local-ref>
        <ejb-ref-name>ejb/MySBLocal</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local-home>org.sample.beans.MySessionBeanLocalHome</local-home>
        <local>org.sample.beans.MySessionBeanLocal</local>
        <ejb-link>secusb.jar#Op</ejb-link>
</ejb-local-ref>
```

resource-ref

```
<!--
The resource-ref element contains a declaration of a Web
Application's reference to an external resource.

The res-auth element indicates whether the application component
code performs resource signon programmatically or whether the
container signs onto the resource based on the principle mapping
information supplied by the deployer. Must be CONTAINER or SERVLET !
-->

<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth)>
```

```
<resource-ref>
        <res-ref-name>jdbc/BookDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
</resource-ref>
```

resource-env-ref

```
<!--
The resource-env-ref element contains a declaration of a web
application's reference to an administered object associated with a
resource in the web application's environment. It consists of an
optional description, the resource environment reference name, and
an indication of the resource environment reference type expected by
the web application code.
-->

<!ELEMENT resource-env-ref (description?, resource-env-ref-name,
resource-env-ref-type)>
```

```
<resource-env-ref>
        <resource-env-ref-name>jms/Orders</resource-env-ref-name>
        <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
```

```
</resource-env-ref>
```

Explain the purpose of a WAR file and describe the contents of a WAR file, how one may be constructed.

Web applications can be packaged and signed into a Web ARchive format (WAR) file using the standard Java archive tools. For example, an application for issue tracking might be distributed in an archive file called `issuetrack.war`. When packaged into such a form, a `META-INF` directory will be present which contains information useful to Java archive tools. This directory must not be directly served as content by the container in response to a Web client's request, though its contents are visible to servlet code via the `getResource` and `getResourceAsStream` calls on the `ServletContext`. Also, any requests to access the resources in `META-INF` directory must be returned with a `SC_NOT_FOUND`(404) response.

A WAR usually contains following resources:

- Servlets, JavaServer Pages (JSP), Custom Tag Libraries.
- Server-side utility classes (database beans, shopping carts, and so on).
- Static web resources (HTML, image, and sound files, and so on).
- Client-side classes (applets and utility classes).

The directory structure of a Web application consists of two parts. The first part is the public directory structure containing HTML/XML documents, JSP pages, images, applets, and so on. The container appropriately serves the directory's contents against incoming requests. The second part is a special `WEB-INF` directory that contains the following files and directories:

- `web.xml` - the web application deployment descriptor.
- Tag Library Descriptor files.
- `classes/` - a directory that contains server-side classes: servlet, utility classes, and JavaBeans components.
- `lib/` - a directory that contains JAR archives of libraries (tag libraries and any utility libraries called by server-side classes).
- `tags/` - a directory that contains Tag files (made easily accessible to JSPs without the need to explicitly write a Tag Library Descriptor files).

The structure of the WAR files looks like this:

```
WEB-INF/
WEB-INF/web.xml
WEB-INF/classes/
WEB-INF/lib/
WEB-INF/tags/
```

To prepare the web application for deployment, package it in a WAR file using the following `jar` utility command from the top-level directory of the application:

```
jar cvf web_app.war .
```

where `web_app` is the web application name.

Chapter 3. The Web Container Model

For the `ServletContext` initialization parameters: write servlet code to access initialization parameters; and create the deployment descriptor elements for declaring initialization parameters.

The following methods of the `ServletContext` interface allow the servlet access to context initialization parameters associated with a Web application as specified by the Application Developer in the deployment descriptor:

- `getInitParameter`

  Returns a `String` containing the value of the named context-wide initialization parameter, or `null` if the parameter does not exist. This method can make available configuration information useful to an entire "web application". For example, it can provide a webmaster’s email address or the name of a system that holds critical data.

- `getInitParameterNames`

  Returns the names of the context's initialization parameters as an `Enumeration` of `String` objects, or an EMPTY `Enumeration` if the context has NO initialization parameters.

Initialization parameters are used by an Application Developer to convey setup information. Typical examples are a Webmaster’s e-mail address, or the name of a system that holds critical data.

```
public interface ServletContext {

	public java.lang.String getInitParameter(java.lang.String name);
	public java.util.Enumeration getInitParameterNames();

}
```

Context initialization parameters that define shared `String` constants used within your application, which can be customized by the system administrator who is installing your application. The values actually assigned to these parameters can be retrieved in a servlet or JSP page by calling:

```
javax.servlet.ServletContext context = getServletContext();
String value = context.getInitParameter("webmaster");
```

where "webmaster" matches the `param-name` element of one of these initialization parameters.

You can define any number of context initialization parameters, including zero:

```
<web-app>
	...
	<context-param>
		<param-name>webmaster</param-name>
		<param-value>myaddress@mycompany.com</param-value>
		<description>
			The EMAIL address of the administrator to whom questions
			and comments about this application should be addressed.
		</description>
	</context-param>
	...
</web-app>
```

```
<!--
The context-param element contains the declaration of a web
application’s servlet context initialization parameters.
-->

<!ELEMENT context-param (param-name, param-value, description?)>
```

For the fundamental servlet attribute scopes (request, session, and context): write servlet code to add, retrieve, and remove attributes; given a usage scenario, identify the proper scope for an attribute; and identify multi-threading issues associated with each scope.

Request Attributes.

Attributes are objects associated with a request. Attributes may be set by the container to express information that otherwise could not be expressed via the API, or may be set by a servlet to communicate information to another servlet (via the `RequestDispatcher`). Attributes are accessed with the following methods of the `ServletRequest` interface:

- `getAttribute`

  Returns the value of the named attribute as an `Object`, or `null` if no attribute of the given name exists. Attributes can be set two ways. The servlet container may set attributes to make available custom information about a request. Attributes can also be set programatically using `setAttribute(String, Object)`. This allows information to be embedded into a request before a `RequestDispatcher` call. Attribute names should follow the same conventions as package names. This specification reserves names matching `java.*`, `javax.*`, and `sun.*`.

- `getAttributeNames`

  Returns an `Enumeration` containing the names of the attributes available to this request. This method returns an EMPTY `Enumeration` if the request has no attributes available to it.

- `setAttribute`

  Stores an attribute in this request. Attributes are reset between requests. This method is most often used in conjunction with `RequestDispatcher`. Attribute names should follow the same conventions as package names. Names beginning with `java.*`, `javax.*`, and `com.sun.*`, are reserved for use by Sun Microsystems. If the object passed in is `null`, the effect is the same as calling `removeAttribute(String)`.

- `removeAttribute`

  Removes an attribute from this request. This method is not generally needed as attributes only persist as long as the request is being handled.

Only ONE attribute value may be associated with an attribute name.

```
package javax.servlet;

public interface ServletRequest {

	public java.lang.Object getAttribute(java.lang.String name);
```

```
        public java.util.Enumeration getAttributeNames();
        public void setAttribute(java.lang.String name, java.lang.Object o);
        public void removeAttribute(java.lang.String name);

}
```

Attribute names beginning with the prefixes of "java." and "javax." are RESERVED for definition by this specification. Similarly, attribute names beginning with the prefixes of "sun.", and "com.sun." are reserved for definition by Sun Microsystems. It is suggested that all attributes placed in the attribute set be named in accordance with the reverse domain name convention suggested by the Java Programming Language Specification for package naming.

Session Attributes.

A servlet can bind an object attribute into an `HttpSession` implementation by name. Any object bound into a session is available to any other servlet that belongs to the same `ServletContext` and handles a request identified as being a part of the same session.

- `getAttribute`

  Returns the object bound with the specified name in this session, or `null` if no object is bound under the name.

- `getAttributeNames`

  Returns an `Enumeration` of `String` objects containing the names of all the objects bound to this session.

- `setAttribute`

  Binds an object to this session, using the name specified. If an object of the same name is already bound to the session, the object is replaced. After this method executes, and if the new object implements `HttpSessionBindingListener`, the container calls `HttpSessionBindingListener.valueBound`. The container then notifies any `HttpSessionAttributeListeners` in the web application. If an object was already bound to this session of this name that implements `HttpSessionBindingListener`, its `HttpSessionBindingListener.valueUnbound` method is called. If the value passed in is `null`, this has the same effect as calling `removeAttribute()`.

- `removeAttribute`

  Removes the object bound with the specified name from this session. If the session does not have an object bound with the specified name, this method does nothing. After this method executes, and if the object implements `HttpSessionBindingListener`, the container calls `HttpSessionBindingListener.valueUnbound`. The container then notifies any `HttpSessionAttributeListeners` in the web application.

```
package javax.servlet.http;

public interface HttpSession {

        public java.lang.Object getAttribute(java.lang.String name);
        public java.util.Enumeration getAttributeNames();
        public void setAttribute(java.lang.String name, java.lang.Object value);
        public void removeAttribute(java.lang.String name);

}
```

Some objects may require notification when they are placed into, or removed from, a session. This information can be obtained by having the object implement the `HttpSessionBindingListener` interface. This interface defines the following methods that will signal an object being bound into, or being unbound from, a session

- `valueBound`
- `valueUnbound`

The `valueBound` method must be called BEFORE the object is made available via the `getAttribute` method of the `HttpSession` interface. The `valueUnbound` method must be called AFTER the object is no longer available via the `getAttribute` method of the `HttpSession` interface.

Multiple servlets executing request threads may have active access to a single session object at the same time. The Developer has the responsibility for synchronizing access to session resources as appropriate.

Within an application marked as distributable, all requests that are part of a session must be handled by one Java Virtual Machine (JVM) at a time. The container must be able to handle all objects placed into instances of the `HttpSession` class using the `setAttribute` or `putValue` methods appropriately. The following restrictions are imposed to meet these conditions:

- The container must accept objects that implement the `Serializable` interface.
- The container may choose to support storage of other designated objects in the `HttpSession`, such as references to Enterprise JavaBeans components and transactions.
- Migration of sessions will be handled by container-specific facilities.

The distributed servlet container must throw an `IllegalArgumentException` for objects where the container cannot support the mechanism necessary for migration of the session storing them.

Containers must notify any session attributes implementing the `HttpSessionActivationListener` during migration of a session. They must notify listeners of passivation prior to serialization of a session, and of activation after deserialization of a session.

Application Developers writing distributed applications should be aware that since the container may run in more than one Java virtual machine, the developer cannot depend on static variables for storing an application state. They should store such states using an enterprise bean or a database.

Context Attributes.

A servlet can bind an object attribute into the context by name. Any attribute bound into a context is available to any other servlet that is part of the same Web application. The following methods of `ServletContext` interface allow access to this functionality:

- `setAttribute`

  Binds an object to a given attribute name in this servlet context. If the name specified is already used for an attribute, this method will REPLACE the attribute with the new to the new attribute. If listeners are configured on the `ServletContext` the container notifies them accordingly. If a `null` value is passed, the effect is the same as calling `removeAttribute()`. Attribute names should follow the same convention as package names. The Java Servlet API specification reserves names matching `java.*`, `javax.*`, and `sun.*`.

- `getAttribute`

  Returns the servlet container attribute with the given name, or `null` if there is no attribute by that name. An attribute allows a servlet container to give the servlet additional information not already provided by this interface. See your server documentation for information about its attributes. A list of supported attributes can be retrieved using `getAttributeNames`. The attribute is returned as a `java.lang.Object` or some subclass. Attribute names should follow the same convention as package names. The Java Servlet API specification reserves names matching `java.*`, `javax.*`, and `sun.*`.

- `getAttributeNames`

  Returns an `Enumeration` containing the attribute names available within this servlet context. Use the `getAttribute(String)` method with an attribute name to get the value of an attribute.

- `removeAttribute`

  Removes the attribute with the given name from the servlet context. After removal, subsequent calls to `getAttribute(String)` to retrieve the attribute's value will return `null`. If listeners are configured on the `ServletContext` the container notifies them accordingly.

```
package javax.servlet;

public interface ServletContext {

        public void setAttribute(java.lang.String name, java.lang.Object object)
        public java.lang.Object getAttribute(java.lang.String name);
        public java.util.Enumeration getAttributeNames();
        public void removeAttribute(java.lang.String name);

}
```

Context attributes are LOCAL to the JVM in which they were created. This prevents `ServletContext` attributes from being a shared memory store in a distributed container. When information needs to be shared between servlets running in a distributed environment, the information should be placed into a session, stored in a database, or set in an Enterprise JavaBeans component.

Describe the Web container request processing model; write and configure a filter; create a request or response wrapper; and given a design problem, describe how to apply a filter or a wrapper.

Request processing model.

The container, when receiving an incoming request, processes the request as follows:

- Identifies the target Web resource according to the rules of mappings.
- If there are filters matched by servlet name and the Web resource has a `<servlet-name>`, the container builds the chain of filters matching in the order declared in the deployment descriptor. The last filter in this chain corresponds to the last `<servlet-name>` matching filter and is the filter that invokes the target Web resource.
- If there are filters using `<url-pattern>` matching and the `<url-pattern>` matches the request URI according to the rules of mappings, the container builds the chain of `<url-pattern>` matched filters in the same order as declared in the deployment descriptor. The last filter in this chain is the last `<url-pattern>` matching filter in the deployment descriptor for this request URI. The last filter in this chain is the filter that invokes the first filter in the `<servlet-name>` matching chain, or invokes the target Web resource if there are none.

The order the container uses in building the chain of filters to be applied for a particular request URI is as follows:

1. First, the `<url-pattern>` matching filter mappings in the same order that these elements appear in the deployment descriptor.
2. Next, the `<servlet-name>` matching filter mappings in the same order that these elements appear in the deployment descriptor.

Writing and configuring a filter.

Filters are Java components that allow on the fly transformations of payload and header information in both the request into a resource and the response from a resource. Filters differ from Web components in that they usually do not themselves create a response. Instead, a filter provides functionality that can be "attached" to any kind of Web resource. As a consequence, a filter should not have any dependencies on a Web resource for which it is acting as a filter, so that it can be composable with more than one type of Web resource.

A filter is a reusable piece of code that can transform the content of HTTP requests, responses, and header information. Filters do not generally create a response or respond to a request as servlets do, rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource. Filters can act on dynamic or static content.

The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

You can configure a Web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the Web application containing the component is deployed and is instantiated when a Web container loads the component.

The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `javax.servlet` package.

The application developer creates a filter by implementing the `javax.servlet.Filter` interface and providing a public constructor taking NO arguments. The class is packaged in the Web Archive along with the static content and servlets that make up the Web application. A filter is declared using the `<filter>` element in the deployment descriptor. A filter or collection of filters can be configured for invocation by defining `<filter-mapping>` elements in the deployment descriptor. This is done by mapping filters to a particular servlet by the servlet's logical name, or mapping to a group of servlets and static content resources by mapping a filter to a URL pattern.

```
package javax.servlet;

public interface Filter {
        public void init(FilterConfig filterConfig) throws ServletException;
        public void doFilter(ServletRequest request, ServletResponse response,
                FilterChain chain) throws java.io.IOException, ServletException;
        public void destroy();

}
```

The most important method in this interface is the doFilter method, which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if it wishes to modify request headers or data.
- Customize the response object if it wishes to modify response headers or data.

- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target Web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. It invokes the next entity by calling the `doFilter` method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.

- Examine response headers after it has invoked the next filter in the chain.

- Throw an exception to indicate an error in processing.

In addition to `doFilter`, you must implement the `init` and `destroy` methods. The `init` method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the `FilterConfig` object passed to `init`.

After deployment of the Web application, and before a request causes the container to access a Web resource, the container must locate the list of filters that must be applied to the Web resource as described below. The container must ensure that it has instantiated a filter of the appropriate class for each filter in the list, and called its `init(FilterConfig config)` method. The filter may throw an exception to indicate that it cannot function properly. If the exception is of type `UnavailableException`, the container may examine the `isPermanent` attribute of the exception and may choose to retry the filter at some later time.

Only ONE instance per `<filter>` declaration in the deployment descriptor is instantiated per Java Virtual Machine (JVM) of the container. The container provides the filter config as declared in the filter's deployment descriptor, the reference to the `ServletContext` for the Web application, and the set of initialization parameters.

When the container receives an incoming request, it takes the first filter instance in the list and calls its `doFilter` method, passing in the `ServletRequest` and `ServletResponse`, and a reference to the `FilterChain` object it will use.

The `doFilter` method of a filter will typically be implemented following this or some subset of the following pattern:

1. The method examines the request's headers.

2. The method may wrap the request object with a customized implementation of `ServletRequest` or `HttpServletRequest` in order to modify request headers or data.

3. The method may wrap the response object passed in to its `doFilter` method with a customized implementation of `ServletResponse` or `HttpServletResponse` to modify response headers or data.

4. The filter may invoke the next entity in the filter chain. The next entity may be another filter, or if the filter making the invocation is the last filter configured in the deployment descriptor for this chain, the next entity is the target Web resource. The invocation of the next entity is effected by calling the `doFilter` method on the `FilterChain` object, and passing in the request and response with which it was called or passing in wrapped versions it may have created.

   The filter chain's implementation of the `doFilter` method, provided by the container, must locate the next entity in the filter chain and invoke its `doFilter` method, passing in the appropriate request and response objects.

   Alternatively, the filter chain can block the request by not making the call to invoke the next entity, leaving the filter responsible for filling out the response object.

5. After invocation of the next filter in the chain, the filter may examine response headers.

6. Alternatively, the filter may have thrown an exception to indicate an error in processing. If the filter throws an `UnavailableException` during its `doFilter` processing, the container must not

attempt continued processing down the filter chain. It may choose to retry the whole chain at a later time if the exception is not marked permanent.

7. When the last filter in the chain has been invoked, the next entity accessed is the target servlet or resource at the end of the chain.

8. Before a filter instance can be removed from service by the container, the container must first call the `destroy` method on the filter to enable the filter to release any resources and perform other cleanup operations.

```
public final class ExampleFilter implements Filter {
        private String attribute = null;
        private FilterConfig filterConfig = null;

        public void init(FilterConfig filterConfig) throws ServletException {
                this.filterConfig = filterConfig;
                this.attribute = filterConfig.getInitParameter("attribute");
        }

        public void doFilter(ServletRequest request, ServletResponse response,
                FilterChain chain) throws IOException, ServletException {

                // Store ourselves as a request attribute (if requested)
                if (attribute != null) {
                        request.setAttribute(attribute, this);
                }

                // Time and log the subsequent processing
                long startTime = System.currentTimeMillis();
                chain.doFilter(request, response);
                long stopTime = System.currentTimeMillis();
                filterConfig.getServletContext().log
                        (this.toString() + ": " + (stopTime - startTime) +
                        " milliseconds");
        }

        public void destroy() {
                this.attribute = null;
                this.filterConfig = null;
        }

}
```

```
<web-app>
        ...
        <!-- Define servlet-mapped and path-mapped example filters -->
        <filter>
                <filter-name>Servlet Mapped Filter</filter-name>
                <filter-class>filters.ExampleFilter</filter-class>
                <init-param>
                        <param-name>attribute</param-name>
                        <param-value>filters.ExampleFilter.SERVLET_MAPPED</param
                </init-param>
        </filter>

        <filter>
                <filter-name>Path Mapped Filter</filter-name>
                <filter-class>filters.ExampleFilter</filter-class>
                <init-param>
                        <param-name>attribute</param-name>
                        <param-value>filters.ExampleFilter.PATH_MAPPED</param-va
                </init-param>
        </filter>
        <!-- Define filter mappings for the defined filters -->
        <filter-mapping>
                <filter-name>Servlet Mapped Filter</filter-name>
                <servlet-name>invoker</servlet-name>
        </filter-mapping>

        <filter-mapping>
                <filter-name>Path Mapped Filter</filter-name>
                <url-pattern>/servlet/*</url-pattern>
        </filter-mapping>
        ...
</web-app>
```

Here is another example of a filter declaration:

```
<filter>
        <filter-name>Image Filter</filter-name>
        <filter-class>com.acme.ImageServlet</filter-class>
</filter>
```

Once a filter has been declared in the deployment descriptor, the assembler uses the `filter-mapping` element to define servlets and static resources in the Web application to which the filter is to be applied. Filters can be associated with a servlet using the `servlet-name` element. For example, the following code example maps the `Image Filter` filter to the `ImageServlet` servlet:

```
<filter-mapping>
        <filter-name>Image Filter</filter-name>
        <servlet-name>ImageServlet</servlet-name>
</filter-mapping>
```

Filters can be associated with groups of servlets and static content using the `url-pattern` style of filter mapping:

```
<filter-mapping>
        <filter-name>Logging Filter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

Here the `Logging Filter` is applied to all the servlets and static content pages in the Web application, because every request URI matches the `'/*'` URL pattern.

The `url-pattern` matching takes precedence (is applied first) over the `servlet-name` matching (is applied next).

```
<!--
Declares a filter in the web application. The filter is mapped to
either a servlet or a URL pattern in the filter-mapping element,
using the filter-name value to reference. Filters can access the
initialization parameters declared in the deployment descriptor at
runtime via the FilterConfig interface.
```

```
Used in: web-app
-->

<!ELEMENT filter (icon?, filter-name, display-name?, description?,
        filter-class, init-param*)>
```

```
<!--
Declaration of the filter mappings in this web application. The
container uses the filter-mapping declarations to decide which
filters to apply to a request, and in what order. The container
matches the request URI to a Servlet in the normal way. To determine
which filters to apply it matches filter-mapping declarations either
on servlet-name, or on url-pattern for each filter-mapping element,
depending on which style is used. The order in which filters are
invoked is the order in which filter-mapping declarations that match
a request URI for a servlet appear in the list of filter-mapping
elements.The filter-name value must be the value of the <filter-name>
sub-elements of one of the <filter> declarations in the deployment
descriptor.

Used in: web-app
-->

<!ELEMENT filter-mapping (filter-name, (url-pattern | servlet-name), dispatcher*
```

The `dispatcher` has four legal values: `FORWARD`, `REQUEST`, `INCLUDE`, and `ERROR`. A value of `FORWARD` means the `Filter` will be applied under `RequestDispatcher.forward()` calls. A value of `REQUEST` means the `Filter` will be applied under ordinary client calls to the PATH or SERVLET. A value of `INCLUDE` means the `Filter` will be applied under `RequestDispatcher.include()` calls. A value of `ERROR` means the `Filter` will be applied under the error page mechanism. The absence of any dispatcher elements in a `filter-mapping` indicates a default of applying filters only under ordinary client calls to the PATH or SERVLET (`REQUEST`).

```
<filter-mapping>
        <filter-name>Logging Filter</filter-name>
        <url-pattern>/products/*</url-pattern>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

This example would result in the `Logging Filter` being invoked by client requests starting `/products/...` and underneath a request dispatcher `forward()` call where the request dispatcher has path commencing `/products/...`.

Wrapping request and response objects.

As well as performing basic pre and post processing operations a filter can also wrap up the request or response objects in a custom wrapper. Such custom wrappers can then modify the information provided to the servlet via a request object or process information generated by the servlet via the response object. There are four classes that make up the Wrapping API. These are the `javax.servlet.ServletRequestWrapper`, `javax.servlet.ServletResponseWrapper`, `javax.servlet.http.HttpServletRequestWrapper` and `javax.servlet.http.HttpServletResponseWrapper`. These classes implement the respective interfaces (e.g. `ServletRequest`, `ServletResponse`, `HttpServletRequest` and `HttpServletResponse`) and can thus be used anywhere that these interfaces are specified. Most notably they can therefore be used inside a `Filter` to wrap the actual request or response object up so that the filter can control either the data accessed by the `Servlet` (or JSP) or generated by the `Servlet` or JSP. A particular use of these wrappers is to perform some pre or post processing of the data being used or generated by the `Servlet` so that the `Servlet` does not need to know about this processing.

A filter that modifies a response must usually capture the response before it is returned to the client. The way to do this is to pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the `getWriter` or `getOutputStream` method to return this stand-in stream. The wrapper is passed to the `doFilter` method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object. This approach follows the well-known Wrapper or Decorator pattern described in 'Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995)'.

To override request methods, you wrap the request in an object that extends `ServletRequestWrapper` or `HttpServletRequestWrapper`. To override response methods, you wrap the response in an object that extends `ServletResponseWrapper` or `HttpServletResponseWrapper`.

Example of filter with wrapper (post processing of servlet's output):

```
public final class HitCounterFilter implements Filter {
        private FilterConfig filterConfig = null;

        public void init(FilterConfig filterConfig)
                        throws ServletException {
                this.filterConfig = filterConfig;
        }

        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException
                StringWriter sw = new StringWriter();
                PrintWriter writer = new PrintWriter(sw);
                Counter counter = (Counter)filterConfig.getServletContext().
                        getAttribute("hitCounter");
                writer.println();
                writer.println("===============");
                writer.println("The number of hits is: " + counter.incCounter())
                writer.println("===============");
                // Log the resulting string
                writer.flush();
                filterConfig.getServletContext().log(sw.getBuffer().toString());

                PrintWriter out = response.getWriter();
                CharResponseWrapper wrapper =
                        new CharResponseWrapper((HttpServletResponse)response);
                chain.doFilter(request, wrapper);
                CharArrayWriter caw = new CharArrayWriter();
                caw.write(wrapper.toString().
                        substring(0, wrapper.toString().indexOf("</body>")-1));
                caw.write("<p>\n<center><center>" +
                        messages.getString("Visitor") +
                        "<font color='red'>" + counter.getCounter() +
                        "</font><center>");
                caw.write("\n</body></html>");
                response.setContentLength(caw.toString().length());
                out.write(caw.toString());
                out.close();
        }

        public void destroy() {
                this.filterConfig = null;
        }
}
```

```
public class CharResponseWrapper extends HttpServletResponseWrapper {
    private CharArrayWriter output;

    public String toString() {
        return output.toString();
    }

    public CharResponseWrapper(HttpServletResponse response) {
        super(response);
        output = new CharArrayWriter();
    }

    public PrintWriter getWriter() {
        return new PrintWriter(output);
    }
}
```

`HitCounterFilter` wraps the response in a `CharResponseWrapper`. The wrapped response is passed to the next object in the filter chain. Next servlet writes (buffers) its response into the stream created by `CharResponseWrapper`. When `chain.doFilter` returns, `HitCounterFilter` retrieves the servlet's response from `PrintWriter` and writes it to a buffer (`CharArrayWriter`). The filter inserts the value of the counter into the buffer, resets the content length header of the response, and finally writes the contents of the buffer to the response stream.

Describe the Web container life cycle event model for requests, sessions, and web applications; create and configure listener classes for each scope life cycle; create and configure scope attribute listener classes; and given a scenario, identify the proper attribute listener to use.

The application events facility gives the Web Application Developer greater control over the lifecycle of the `ServletContext` and `HttpSession` and `ServletRequest`, allows for better code factorization, and increases efficiency in managing the resources that the Web application uses.

Application event listeners are classes that implement one or more of the servlet event listener interfaces. They are instantiated and registered in the Web container at the time of the deployment of the Web application. They are provided by the Developer in the WAR.

Servlet event listeners support event notifications for state changes in the `ServletContext`, `HttpSession` and `ServletRequest` objects. Servlet context listeners are used to manage resources or state held at a JVM level for the application. HTTP session listeners are used to manage state or resources associated with a series of requests made into a Web application from the same client or user. Servlet request listeners are used to manage state across the lifecycle of servlet requests.

There may be multiple listener classes listening to each event type, and the Developer may specify the order in which the container invokes the listener beans for each event type.

Servlet Context Events and Listeners.

Implementations of the following interface receive notifications about changes to the servlet context of the web application they are part of. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application. All `ServletContextListener`s are notified of context initialization before any filter or servlet in the web application is initialized. All servlets and filters have been destroyed before any `ServletContextListener`s are notified of context destruction.

```
package javax.servlet;
```

```
public interface ServletContextListener extends java.util.EventListener {

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

}
```

This is the event class for notifications about changes to the servlet context of a web application:

```
package javax.servlet;

public class ServletContextEvent extends java.util.EventObject {

        public ServletContext getServletContext();

}
```

Implementations of the following interface receive notifications of changes to the attribute list on the servlet context of a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application.

- `attributeAdded` notifies that a new attribute was added to the servlet context. Called after the attribute is added.
- `attributeRemoved` notifies that an existing attribute has been removed from the servlet context. Called after the attribute is removed.
- `attributeReplaced(` notifies that an attribute on the servlet context has been replaced. Called after the attribute is replaced.

```
package javax.servlet;

public interface ServletContextAttributeListener extends java.util.EventListener

        public void attributeAdded(ServletContextAttributeEvent scae);
        public void attributeRemoved(ServletContextAttributeEvent scae);
        public void attributeReplaced(ServletContextAttributeEvent scae);

}
```

This is the event class for notifications about changes to the attributes of the servlet context of a web application:

```
package javax.servlet;

public class ServletContextAttributeEvent extends javax.servlet.ServletContextEv

        public java.lang.String getName();
        public java.lang.Object getValue();

}
```

HTTP Session Events and Listeners.

Implementations of the following interface are notified of changes to the list of active sessions in a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application.

```
package javax.servlet.http;

public interface HttpSessionListener extends java.util.EventListener {

        public void sessionCreated(HttpSessionEvent hse);
        public void sessionDestroyed(HttpSessionEvent hse);

}
```

This is the class representing event notifications for changes to sessions within a web application:

```
package javax.servlet.http;

public class HttpSessionEvent extends java.util.EventObject {

        public HttpSession getSession();

}
```

Following listener interface can be implemented in order to get notifications of changes to the attribute lists of sessions within this web application:

- `attributeAdded` notifies that a new attribute was added to the session. Called after the attribute is added.

- `attributeRemoved` notifies that an existing attribute has been removed from the session. Called after the attribute is removed.

- `attributeReplaced` notifies that an attribute has been replaced in the session. Called after the attribute is replaced.

```
package javax.servlet.http;

public interface HttpSessionAttributeListener extends java.util.EventListener {

        public void attributeAdded(HttpSessionBindingEvent hsbe);
        public void attributeRemoved(HttpSessionBindingEvent hsbe);
        public void attributeReplaced(HttpSessionBindingEvent hsbe);

}
```

Events of the following type are either sent to an object that implements `HttpSessionBindingListener` when it is bound or unbound from a session, or to a `HttpSessionAttributeListener` that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session. The session binds the object by a call to `HttpSession.setAttribute` and unbinds the object by a call to `HttpSession.removeAttribute`. The `getValue` function returns the value of the attribute that has been added, removed or replaced. If the attribute was added (or bound), this is the value of the attribute. If the attribute was removed (or unbound), this is the value of the removed attribute. If the attribute was replaced, this is the OLD value of the attribute.

```
package javax.servlet.http;

public class HttpSessionBindingEvent extends javax.servlet.http.HttpSessionEvent
```

```
        public java.lang.String getName();
        public HttpSession getSession();
        public java.lang.Object getValue(); // returns :
                // new value for added (bounded) attributes,
                // old value for replaced and removed (unbounded) attributes

}
```

When container migrates a session between VMs in a distributed container setting, all session attributes implementing the `HttpSessionActivationListener` interface are notified.

Objects that are bound to a session may listen to container events notifying them that sessions will be passivated and that session will be activated. A container that migrates session between VMs or persists sessions is required to notify all attributes bound to sessions implementing `HttpSessionActivationListener`.

```
package javax.servlet.http;

public interface HttpSessionActivationListener extends java.util.EventListener {

        public void sessionDidActivate(HttpSessionEvent hse);
        public void sessionWillPassivate(HttpSessionEvent hse);

}
```

The following interface causes an OBJECT to be notified when it is bound to or unbound from a session. The object is notified by an `HttpSessionBindingEvent` object. This may be as a result of a servlet programmer explicitly unbinding an attribute from a session, due to a session being invalidated, or due to a session timing out.

```
package javax.servlet.http;

public interface HttpSessionBindingListener extends java.util.EventListener {

        public void valueBound(HttpSessionBindingEvent hsbe);
        public void valueUnbound(HttpSessionBindingEvent hsbe);

}
```

Servlet Request Events and Listeners.

A `ServletRequestListener` can be implemented by the developer interested in being notified of requests coming in and out of scope in a web component. A request is defined as coming into scope when it is about to enter the first servlet or filter in each web application, as going out of scope when it exits the last servlet or the first filter in the chain.

```
package javax.servlet;

public interface ServletRequestListener {

        requestInitialized(ServletRequestEvent sre);
        requestDestroyed(ServletRequestEvent sre);

}
```

Events of this kind indicate lifecycle events for a `ServletRequest`. The source of the event is the `ServletContext` of this web application.

```
package javax.servlet;

public class ServletRequestEvent extends java.util.EventObject {

        public ServletRequest getServletRequest();
        public ServletContext getServletContext();

}
```

A `ServletRequestAttributeListener` can be implemented by the developer interested in being notified of request attribute changes. Notifications will be generated while the request is within the scope of the web application in which the listener is registered. A request is defined as coming into scope when it is about to enter the first servlet or filter in each web application, as going out of scope when it exits the last servlet or the first filter in the chain.

```
package javax.servlet;

public interface ServletRequestAttributeListener {
        public void attributeAdded(ServletRequestAttributeEvent srae);
        public void attributeRemoved(ServletRequestAttributeEvent srae);
        public void attributeReplaced(ServletRequestAttributeEvent srae);

}
```

The following is the event class for notifications of changes to the attributes of `ServletRequest` in an application. The `getValue()` function returns the value of the attribute that has been added, removed or replaced. If the attribute was added, this is the value of the attribute. If the attribute was removed, this is the value of the REMOVED attribute. If the attribute was replaced, this is the OLD value of the attribute.

```
package javax.servlet;

public class ServletRequestAttributeEvent extends ServletRequestEvent {

        public java.lang.String getName();
        public java.lang.Object getValue();

}
```

Table 3.1. Events and Listener Interfaces

Scope	Event	Listener Interface and Event Class
Servlet Context	Initialization and destruction	`ServletContextListener`, `ServletContextEvent`
	Attribute added, removed, or replaced	`ServletContextAttributeListener`, `ServletContextAttributeEvent`
HTTP Session	Created and destroyed	`HttpSessionListener`, `HttpSessionEvent`
	Activated and passivated (migrated)	`HttpSessionActivationListener`, `HttpSessionEvent`
	Attribute added, removed, or replaced	`HttpSessionAttributeListener`, `HttpSessionBindingEvent` (note the class name !)

	Object bound or unbound	`HttpSessionBindingListener` (note, interface must be implemented by attribute class !), `HttpSessionBindingEvent`
Servlet Request	Initialized and destroyed	`ServletRequestListener`, `ServletRequestEvent`
	Attribute added, removed, or replaced	`ServletRequestAttributeListener`, `ServletRequestAttributeEvent`

The Developer of the Web application provides listener classes implementing one or more of the listener classes in the servlet API. Each listener class must have a public constructor taking NO arguments. The listener classes are packaged into the WAR, either under the `WEB-INF/classes` archive entry, or inside a JAR in the `WEB-INF/lib` directory.

Listener classes are declared in the Web application deployment descriptor using the `listener` element. They are listed by class name in the order in which they are to be invoked. During Web application execution, listeners are invoked in the order of their registration.

On application shutdown, listeners are notified in REVERSE order to their declarations with notifications to session listeners preceeding notifications to context listeners. Session listeners must be notified of session invalidations prior to context listeners being notified of application shutdown.

```
<web-app>
        ...
        <listener>
                <listener-class>listeners.ContextListener</listener-class>
        </listener>
        ...
</web-app>
```

In distributed Web containers, `HttpSession` instances are scoped to the particular JVM servicing session requests, and the `ServletContext` object is scoped to the Web container's JVM. Distributed containers are not required to propagate either servlet context events or `HttpSession` events to other JVMs. Listener class instances are scoped to one per deployment descriptor declaration per Java Virtual Machine.

Describe the `RequestDispatcher` mechanism; write servlet code to create a request dispatcher; write servlet code to forward or include the target resource; and identify and describe the additional request-scoped attributes provided by the container to the target resource.

`RequestDispatcher` description.

`RequestDispatcher` defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the `RequestDispatcher` object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

An object implementing the `RequestDispatcher` interface may be obtained via the following methods:

- `ServletContext.getRequestDispatcher(String path)`
- `ServletContext.getNamedDispatcher(String name)`
- `ServletRequest.getRequestDispatcher(String path)`

The `ServletContext.getRequestDispatcher` method takes a `String` argument describing a path within the scope of the `ServletContext`. This path must be relative to the root of the `ServletContext` and begin with a `'/'`. The method uses the path to look up a servlet, using the servlet path matching rules, wraps it with a `RequestDispatcher` object, and returns the resulting object. If no servlet can be resolved based on the given path, a `RequestDispatcher` is provided that returns the content for that path.

The `ServletContext.getNamedDispatcher` method takes a `String` argument indicating the NAME of a servlet known to the `ServletContext`. If a servlet is found, it is wrapped with a `RequestDispatcher` object and the object is returned. If no servlet is associated with the given name, the method must return `null`.

To allow `RequestDispatcher` objects to be obtained using relative paths that are relative to the path of the current request (not relative to the root of the `ServletContext`), the `ServletRequest.getRequestDispatcher` method is provided in the `ServletRequest` interface. The behavior of this method is similar to the method of the same name in the `ServletContext`. The servlet container uses information in the request object to transform the given relative path against the current servlet to a complete path. For example, in a context rooted at `'/'` and a request to `/garden/tools.html`, a request dispatcher obtained via `ServletRequest.getRequestDispatcher ("header.html")` will behave exactly like a call to `ServletContext.getRequestDispatcher ("/garden/header.html")`.

`RequestDispatcher` creation and using.

```
public class Dispatcher extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse res) {
            RequestDispatcher dispatcher =
                        request.getRequestDispatcher("/template.jsp");
            if (dispatcher != null) dispatcher.forward(request, response);
        }
}
```

`forward` should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an `IllegalStateException`. Uncommitted output in the response buffer is automatically cleared before the forward.

```
public class Dispatcher extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse res) {
                RequestDispatcher dispatcher =
                        getServletContext().getRequestDispatcher("/banner");
                if (dispatcher != null) dispatcher.include(request, response);
        }
}
```

Includes the content of a resource (servlet, JSP page, HTML file) in the response. In essence, this method enables programmatic server-side includes. The `ServletResponse` object has its path elements and parameters remain unchanged from the caller's. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.

```
package javax.servlet;

public interface RequestDispatcher {

        public void forward(ServletRequest request, ServletResponse response)
                throws ServletException, java.io.IOException;
        public void include(ServletRequest request, ServletResponse response)
                throws ServletException, java.io.IOException;

}
```

The `include` method of the `RequestDispatcher` interface may be called at ANY time. The target servlet of the include method has access to all aspects of the request object, but its use of the response object is more limited. It can only write information to the `ServletOutputStream` or `Writer` of the response object and commit a response by writing content past the end of the response buffer, or by explicitly calling the `flushBuffer` method of the `ServletResponse` interface. It CANNOT set headers or call any method that affects the headers of the response. Any attempt to do so must be ignored.

The `forward` method of the `RequestDispatcher` interface may be called by the calling servlet ONLY when NO output has been committed to the client. If output data exists in the response buffer that has not been committed, the content must be cleared before the target servlet's service method is called. If the response has been committed, an `IllegalStateException` must be thrown.

The path elements of the request object exposed to the target servlet must reflect the path used to obtain the `RequestDispatcher`. The only exception to this is if the `RequestDispatcher` was obtained via the `getNamedDispatcher` method. In this case, the path elements of the request object must reflect those of the original request. Before the `forward` method of the `RequestDispatcher` interface returns, the response content MUST be sent and committed, and closed by the servlet container.

The `ServletContext` and `ServletRequest` methods that create `RequestDispatcher` objects using path information allow the optional attachment of query string information to the path. For example, a Developer may obtain a `RequestDispatcher` by using the following code:

```
String path = "/raisins.jsp?orderno=5";
RequestDispatcher rd = context.getRequestDispatcher(path);
rd.include(request, response);
```

Parameters specified in the query string used to create the `RequestDispatcher` take precedence over other parameters of the same name passed to the included servlet. The parameters associated with a `RequestDispatcher` are scoped to apply only for the duration of the `include` or `forward` call.

Additional request-scoped attributes.

Except for servlets obtained by using the `getNamedDispatcher` method, a servlet that has been invoked by another servlet using the `include` method of `RequestDispatcher` has access to the path by which it was invoked.

The following request attributes must be set:

- `javax.servlet.include.request_uri`
- `javax.servlet.include.context_path`
- `javax.servlet.include.servlet_path`
- `javax.servlet.include.path_info`
- `javax.servlet.include.query_string`

These attributes are accessible from the included servlet via the `getAttribute` method on the request object and their values must be equal to the request URI, context path, servlet path, path info, and query string of the INCLUDED servlet, respectively. If the request is subsequently included, these attributes are replaced for that include.

If the included servlet was obtained by using the `getNamedDispatcher` method, these attributes MUST NOT be set.

Except for servlets obtained by using the `getNamedDispatcher` method, a servlet that has been invoked by another servlet using the `forward` method of `RequestDispatcher` has access to the path of the ORIGINAL request.

The following request attributes must be set:

- `javax.servlet.forward.request_uri`
- `javax.servlet.forward.context_path`
- `javax.servlet.forward.servlet_path`
- `javax.servlet.forward.path_info`
- `javax.servlet.forward.query_string`

The values of these attributes must be equal to the return values of the `HttpServletRequest` methods `getRequestURI`, `getContextPath`, `getServletPath`, `getPathInfo`, `getQueryString` respectively, invoked on the request object passed to the first servlet object in the call chain that received the request from the client.

These attributes are accessible from the forwarded servlet via the `getAttribute` method on the request object. Note that these attributes must always reflect the information in the original request even under the situation that multiple forwards and subsequent includes are called.

If the forwarded servlet was obtained by using the `getNamedDispatcher` method, these attributes must not be set.

Chapter 4. Session Management

Write servlet code to store objects into a session object and retrieve objects from a session object.

Sessions are represented by an `HttpSession` object. You access a session by calling the `HttpServletRequest.getSession()` or `HttpServletRequest.getSession(boolean)` method of a request object. This method returns the current session associated with this request, or, if the request does not have a session, it creates one (unless boolean argument is `false`).

You can associate object -valued attributes with a session by name. Such attributes are accessible by any Web component that belongs to the same Web context and is handling a request that is part of the same session.

For example, you can save shopping cart as a session attribute. This allows the shopping cart to be saved between requests and also allows cooperating servlets to access the cart. Some servlet adds items to the cart; another servlet displays, deletes items from, and clears the cart; and next servlet retrieves the total cost of the items in the cart.

```
public class CashierServlet extends HttpServlet {
        public void doGet (HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

                // Get the user's session and shopping cart
                HttpSession session = request.getSession();
                ShoppingCart cart = (ShoppingCart) session.getAttribute("cart");
                ...
                // Determine the total price of the user's books
                double total = cart.getTotal();
                ...
        }
}
```

```
package javax.servlet.http;

public interface HttpSession {

        public java.lang.Object getAttribute(java.lang.String name);
        public java.util.Enumeration getAttributeNames();
        public void removeAttribute(java.lang.String name);
        public void setAttribute(java.lang.String name, java.lang.Object value);

}
```

Given a scenario describe the APIs used to access the session object, explain when the session object was created, and describe the mechanisms used to destroy the session object, and when it was destroyed.

A session is considered 'new' when it is only a prospective session and has not been established. Because HTTP is a request-response based protocol, an HTTP session is considered to be new until a client 'joins' it. A client joins a session when session tracking information has been returned to the server indicating that a session has been established. Until the client joins a session, it cannot be assumed that the next request from the client will be recognized as part of a session.

The session is considered to be 'new' if either of the following is true:

- The client does not yet know about the session.
- The client chooses not to join a session.

These conditions define the situation where the servlet container has no mechanism by which to associate a request with a previous request.

To obtain a session, use the `getSession()` or `getSession(boolean)` method of the `javax.servlet.http.HttpServletRequest` object. When you first obtain the `HttpSession` object, the one of three ways used to establish tracking of the session: cookies, URL rewriting, or Secure Sockets Layer (SSL) information. Assume the servlet container uses cookies. In such a case the servlet container creates a unique session ID and typically sends it back to the browser as a cookie. Each subsequent request from this user (at the same browser) passes the cookie containing the session ID, and the servlet container uses this ID to find the user's existing `HttpSession` object.

If argument in `getSession(boolean)` method is set to `true`, the `HttpSession` object is created if it does not already exist (the same as call of `getSession()` method).

If argument in `getSession(boolean)` method is set to `false`, the `HttpSession` object is NOT created if it does not already exist and method returns `null`.

You can end a session:

- Automatically by servlet container if a session is inactive for a specified time. The administrators provide a way to specify the amount of time after which to invalidate a session.
- By coding the servlet to call the `HttpSession.invalidate()` method on the session object.

In the HTTP protocol, there is no explicit termination signal when a client is no longer active. This means that the only mechanism that can be used to indicate when a client is no longer active is a timeout period.

The default timeout period for sessions is defined by the servlet container and can be obtained via the `int HttpSession.getMaxInactiveInterval()` (sec.) method of the `HttpSession` interface. This timeout can be changed by the Developer using the `HttpSession.setMaxInactiveInterval(int)` (sec.) method of the `HttpSession` interface. The timeout periods used by these methods are defined in

SECONDS. By definition, if the timeout period for a session is set to -1 (or ANY NEGATIVE), the session will never expire. The session invalidation will not take effect until all servlets using that session have exited the service method. Once the session invalidation is initiated, a new request must not be able to see that session.

```
package javax.servlet.http;

public interface HttpSession {

        public int getMaxInactiveInterval();
        public void setMaxInactiveInterval(int interval);
        public void invalidate();
        public boolean isNew();

}
```

Another way to configure session timeout (for all sessions within one web-application) is to use deployment descriptor (`web.xml`). The `session-timeout` element defines the default session timeout interval for all sessions created in this web application. The specified timeout must be expressed in a whole number of MINUTES. If the timeout is 0 or less, the container ensures the default behaviour of sessions is NEVER to time out. If this element is not specified, the container must set its default timeout period.

```
<web-app>
        ...
        <session-config>
                <session-timeout>30</session-timeout>  <!-- 30 minutes -->
        </session-config>
</web-app>
```

Using session listeners, write code to respond to an event when an object is added to a session, and write code to respond to an event when a session object migrates from one VM to another.

Object is added to session. Listener notification.

```
package listeners;

import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;

public final class SessionListener implements HttpSessionAttributeListener {

        public void attributeAdded(HttpSessionBindingEvent event) {
                System.out.println("attributeAdded");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
        }

        public void attributeRemoved(HttpSessionBindingEvent event) {
                System.out.println("attributeRemoved");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
    }

    public void attributeReplaced(HttpSessionBindingEvent event) {
                System.out.println("attributeReplaced");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
    }
}
```

```
<web-app>
        ...
        <listener>
                <listener-class>listeners.SessionListener</listener-class>
        </listener>
</web-app>
```

Object is added to session. Object notification.

```
package beans;

import javax.servlet.http.HttpSessionBindingListener;
import javax.servlet.http.HttpSessionBindingEvent;

public class CounterBean implements HttpSessionBindingListener {

        private int count = 1;

        public int getValue() {
                return count;
        }

        public void increment() {
                count++;
        }

        public void valueBound(HttpSessionBindingEvent event) {
                System.out.println("attributeBounded");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
        }

        public void valueUnbound(HttpSessionBindingEvent event) {
                System.out.println("attributeUnbounded");
                System.out.println(event.getSession().getId());
                System.out.println(event.getName() + "', '" + event.getValue());
        }
}
```

Note, you don't need to (and must not) configure `HttpSessionBindingListener` in the deployment descriptor, just make a class implementing this interface.

Object migrates from one VM to another. Object notification.

```
package beans;

import javax.servlet.http.HttpSessionActivationListener;
import javax.servlet.http.HttpSessionEvent;

public class CounterBean implements HttpSessionActivationListener {

        private int count = 1;

        public int getValue() {
                return count;
        }

        public void increment() {
                count++;
        }

        public void sessionWillPassivate(HttpSessionEvent se) {
                System.out.println("session is about to be passivated");
                // save counter's value to persistent storage
                ....
        }

        public void sessionDidActivate(HttpSessionEvent se) {
                System.out.println("session has just been activated");
                // retrieve counter's value from persistent storage
                ....
        }
}
```

Note, you don't need to (and must not) configure `HttpSessionActivationListener` in the deployment descriptor, just make a class implementing this interface.

Given a scenario, describe which session management mechanism the Web container could employ, how cookies might be used to manage sessions, how URL rewriting might be used to manage sessions, and write servlet code to perform URL rewriting.

The following approaches are using for tracking a user's sessions:

- Cookies

  Session tracking through HTTP cookies is the most used session tracking mechanism and is required to be supported by all servlet containers. The container sends a cookie to the client. The client will then return the cookie on each subsequent request to the server, unambiguously associating the request with a session. The name of the session tracking cookie must be '`JSESSIONID`' (uppercase !).

  ```
  Set-Cookie: JSESSIONID=49EBBB19A1B2F8D10EE075F6F14CB8C9; Path=/
  ```

- SSL Sessions

  Secure Sockets Layer, the encryption technology used in the HTTPS protocol, has a built-in mechanism allowing multiple requests from a client to be unambiguously identified as being part of a session. A servlet container can easily use this data to define a session.

- URL Rewriting

  URL rewriting is the lowest common denominator of session tracking. When a client will not accept a cookie, URL rewriting may be used by the server as the basis for session tracking. URL rewriting involves adding data, a session ID, to the URL path that is interpreted by the container to associate the request with a session.

  The session ID must be encoded as a path parameter in the URL string. The name of the parameter must be '`jsessionid`' (lowercase !). Here is an example of a URL containing encoded path information:

  ```
  http://www.myserver.com/catalog/index.html;jsessionid=1234
  ```

```
package javax.servlet.http;

public interface HttpServletRequest extends javax.servlet.ServletRequest {
        ...
        public boolean isRequestedSessionIdFromCookie();
        public boolean isRequestedSessionIdFromURL();
        public boolean isRequestedSessionIdValid();
}
```

There are 2 methods in the `HttpServletResponse` for URL rewriting:

- `encodeURL(String)`

  Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. The implementation of this method includes the logic to determine whether the session ID needs to be encoded in the URL. For example, if the browser supports cookies, or session tracking is turned off, URL encoding is unnecessary. For robust session tracking, all URLs emitted by a servlet should be run through this method. Otherwise, URL rewriting cannot be used with browsers which do not support cookies.

- `encodeRedirectURL(String)`

  Encodes the specified URL for use in the `sendRedirect` method or, if encoding is not needed, returns the URL unchanged. The implementation of this method includes the logic to determine whether the session ID needs to be encoded in the URL. Because the rules for making this determination can differ from those used to decide whether to encode a normal link, this method is separated from the `encodeURL` method. All URLs sent to the `HttpServletResponse.sendRedirect` method should be run through this method. Otherwise, URL rewriting cannot be used with browsers which do not support cookies.

```
package javax.servlet.http;

public interface HttpServletResponse extends javax.servlet.ServletResponse {

        public java.lang.String encodeURL(java.lang.String url);
        public java.lang.String encodeRedirectURL(java.lang.String url)

}
```

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
                throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        ...
        out.print("<form action='");
        out.print(response.encodeURL("SessionExample"));
        out.print("' ");
        out.println("method='post'>");
}
```

Chapter 5. Web Application Security

Based on the servlet specification, compare and contrast the following security mechanisms: (a) authentication, (b) authorization, (c) data integrity, and (d) confidentiality.

Authentication.

Authentication means by which communicating entities prove to one another that they are acting on behalf of specific identities that are authorized for access.

Authentication is any process by which you verify that someone is who they claim they are. This usually involves a username and a password, but can include any other method of demonstrating identity, such as a smart card, retina scan, voice recognition, or fingerprints. Authentication is equivalent to showing your drivers license at the ticket counter at the airport.

Authorization (access control for resources).

Authorization means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

Authorization is finding out if the person, once identified, is permitted to have the resource. This is usually determined by finding out if that person is a part of a particular group, if that person has paid admission, or has a particular level of security clearance. Authorization is equivalent to checking the guest list at an exclusive party, or checking for your ticket when you go to the opera.

Data Integrity.

Data integrity means used to prove that information has not been modified by a third party while in transit.

Confidentiality (Data Privacy).

Confidentiality means used to ensure that information is made available only to users who are authorized to access it.

In the deployment descriptor, declare a security constraint, a Web resource, the transport guarantee, the login configuration, and a security role.

Specifying Security Constraints.

Security constraints are a declarative way of defining the protection of web content. A security constraint associates authorization and or user data constraints with HTTP operations on web resources. A security constraint, which is represented by `security-constraint` in deployment descriptor, consists of the following elements:

- web resource collection (`web-resource-collection` in deployment descriptor)
- authorization constraint (`auth-constraint` in deployment descriptor)
- user data constraint (`user-data-constraint` in deployment descriptor)

A security constraint that does not contain an authorization constraint shall combine with authorization constraints that name or imply roles to allow unauthenticated access. The special case of an authorization constraint that names NO roles shall combine with any other constraints to OVERRIDE their affects and cause access to be PRECLUDED.

The HTTP operations and web resources to which a security constraint applies (i.e. the constrained requests) are identified by one or more web resource collections. A web resource collection consists of the following elements:

- URL patterns (`url-pattern` in deployment descriptor)
- HTTP methods (`http-method` in deployment descriptor)

An authorization constraint establishes a requirement for authentication and names the authorization roles permitted to perform the constrained requests. A user must be a member of at least one of the named roles to be permitted to perform the constrained requests. The special role name '*' is a shorthand for all role names defined in the deployment descriptor. An authorization constraint that names NO roles indicates that access to the constrained requests MUST NOT be permitted under any circumstances. An authorization constraint consists of the following element:

- role name (`role-name` in deployment descriptor)

A user data constraint establishes a requirement that the constrained requests be received over a protected transport layer connection. The strength of the required protection is defined by the value of the transport guarantee. A transport guarantee of `INTEGRAL` is used to establish a requirement for content integrity and a transport guarantee of `CONFIDENTIAL` is used to establish a requirement for confidentiality. The transport guarantee of `NONE` indicates that the container must accept the constrained requests when received on any connection including an unprotected one. A user data constraint consists of the following element:

- transport guarantee (`transport-guarantee` in deployment descriptor)

If no authorization constraint applies to a request, the container must accept the request without requiring user authentication. If no user data constraint applies to a request, the container must accept the request when received over any connection including an unprotected one.

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>restricted methods</web-resource-name>
                <url-pattern>/*</url-pattern>
                <url-pattern>/acme/wholesale/*</url-pattern>
                <url-pattern>/acme/retail/*</url-pattern>
                <http-method>DELETE</http-method>
                <http-method>PUT</http-method>
        </web-resource-collection>
        <auth-constraint/>
</security-constraint>
```

'/*' DELETE access precluded

'/*' PUT access precluded

'/acme/wholesale/*' DELETE access precluded

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>wholesale</web-resource-name>
                <url-pattern>/acme/wholesale/*</url-pattern>
                <http-method>GET</http-method>
                <http-method>PUT</http-method>
        </web-resource-collection>
        <auth-constraint>
                <role-name>SALESCLERK</role-name>
        </auth-constraint>
</security-constraint>
```

'/acme/wholesale/*' GET SALESCLERK

```
<security-constraint>
        <web-resource-collection>
```

```
                <web-resource-name>wholesale</web-resource-name>
                <url-pattern>/acme/wholesale/*</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
                <role-name>CONTRACTOR</role-name>
        </auth-constraint>
        <user-data-constraint>
                <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
</security-constraint>
```

'/acme/wholesale/*' POST CONTRACTOR CONFIDENTIAL

```
<!--
The security-constraint element is used to associate security
constraints with one or more web resource collections

Used in: web-app
-->

<!ELEMENT security-constraint (display-name?, web-resource-collection+,
auth-constraint?, user-data-constraint?)>
```

Web resource.

```
<!--
The web-resource-collection element is used to identify a subset
of the resources and HTTP methods on those resources within a web
application to which a security constraint applies. If no HTTP methods
are specified, then the security constraint applies to all HTTP
methods.

Used in: security-constraint
-->

<!ELEMENT web-resource-collection (web-resource-name, description?,
url-pattern*, http-method*)>
```

Transport guarantee.

```
<!--
The user-data-constraint element is used to indicate how data
communicated between the client and container should be protected.

Used in: security-constraint
-->

<!ELEMENT user-data-constraint (description?, transport-guarantee)>
```

Login configuration.

```
<!--
The login-config element is used to configure the authentication
method that should be used, the realm name that should be used for
this application, and the attributes that are needed by the form login
mechanism.

Used in: web-app
-->

<!ELEMENT login-config (auth-method?, realm-name?, form-login-config?)>
```

```
<!-- login configuration uses form-based authentication -->
<login-config>
        <auth-method>FORM</auth-method>
        <realm-name>Form-Based Authentication Area</realm-name>
        <form-login-config>
                <form-login-page>/protected/login.jsp</form-login-page>
                <form-error-page>/protected/error.jsp</form-error-page>
        </form-login-config>
</login-config>
```

Security role.

```
<!--
The security-role element contains the definition of a security
role. The definition consists of an optional description of the
security role, and the security role name.

Used in: web-app

Example:

    <security-role>
        <description>
            This role includes all employees who are authorized
            to access the employee service application.
        </description>
        <role-name>employee</role-name>
    </security-role>
-->

<!ELEMENT security-role (description?, role-name)>
```

```
<!-- Security roles referenced by web application -->
<security-role>
        <role-name>user</role-name>
</security-role>
<security-role>
        <role-name>admin</role-name>
</security-role>
```

Compare and contrast the authentication types (BASIC, DIGEST, FORM, and CLIENT-CERT); describe how the type works; and given a scenario, select an appropriate type.

A web client can authenticate a user to a web server using one of the following mechanisms:

- HTTP Basic Authentication
- HTTP Digest Authentication
- HTTPS Client Authentication
- Form Based Authentication

HTTP Basic Authentication.

HTTP Basic Authentication, which is based on a username and password, is the authentication mechanism defined in the HTTP/1.0 specification. A web server requests a web client to authenticate the user. As part of the request, the web server passes the realm (a string) in which the user is to be authenticated. The realm string of Basic Authentication does not have to reflect any particular security policy domain (confusingly also referred to as a realm). The web client obtains the username and the password from the user and transmits them to the web server. The web server then authenticates the user in the specified realm.

Basic Authentication is not a secure authentication protocol. User passwords are sent in simple base64 ENCODING (not ENCRYPTED !), and the target server is not authenticated. Additional protection can alleviate some of these concerns: a secure transport mechanism (HTTPS), or security at the network level (such as the IPSEC protocol or VPN strategies) is applied in some deployment scenarios.

```
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>User Auth</web-resource-name>
                        <url-pattern>/auth/*</url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>manager</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                <auth-method>BASIC</auth-method>
                <realm-name>User Auth</realm-name>
        </login-config>

        <security-role>
                <role-name>admin</role-name>
        </security-role>
        <security-role>
                <role-name>manager</role-name>
        </security-role>
</web-app>
```

HTTP Digest Authentication.

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However the authentication is performed by transmitting the password in an ENCRYPTED form which is much MORE SECURE than the simple base64 encoding used by Basic Authentication, e.g. HTTPS Client Authentication. As Digest Authentication is not currently in widespread use, servlet containers are encouraged but NOT REQUIRED to support it.

The advantage of this method is that the cleartext password is protected in transmission, it cannot be determined from the digest that is submitted by the client to the server.

Digested password authentication supports the concept of digesting user passwords. This causes the stored version of the passwords to be encoded in a form that is not easily reversible, but that the Web server can still utilize for authentication. From a user perspective, digest authentication acts almost identically to basic authentication in that it triggers a login dialog. The difference between basic and digest authentication is that on the network connection between the browser and the server, the password is encrypted, even on a non-SSL connection. In the server, the password can be stored in clear text or encrypted text, which is true for all login methods and is independent of the choice that the application deployer makes.

```
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>User Auth</web-resource-name>
                        <url-pattern>/auth/*</url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>manager</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                <auth-method>DIGEST</auth-method>
                <realm-name>User Auth</realm-name>
        </login-config>

        <security-role>
                <role-name>admin</role-name>
        </security-role>
        <security-role>
                <role-name>manager</role-name>
        </security-role>
</web-app>
```

HTTPS Client Authentication.

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to possess a Public Key Certificate (PKC). Currently, PKCs are useful in e-commerce applications and also for a single-sign-on from within the browser. Servlet containers that are not J2EE technology compliant are not required to support the HTTPS protocol.

Client-certificate authentication is a more secure method of authentication than either BASIC or FORM authentication. It uses HTTP over SSL, in which the server and, optionally, the client authenticate one another with Public Key Certificates. Secure Sockets Layer (SSL) provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. It is issued by a trusted organization, which is called a certificate authority (CA), and provides identification for the bearer. If you specify client-certificate authentication, the Web server will authenticate the client using the client's X.509 certificate, a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure (PKI). Prior to running an application that uses SSL, you must configure SSL support on the server and set up the public key certificate.

```
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>User Auth</web-resource-name>
                        <url-pattern>/auth/*</url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>manager</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                <auth-method>CLIENT-CERT</auth-method>
                <realm-name>User Auth</realm-name>
        </login-config>

        <security-role>
                <role-name>admin</role-name>
        </security-role>
        <security-role>
                <role-name>manager</role-name>
        </security-role>
</web-app>
```

Form Based Authentication.

The look and feel of the 'login screen' cannot be varied using the web browser's built-in authentication mechanisms. This specification introduces a required form based authentication mechanism which allows a Developer to CONTROL the LOOK and FEEL of the login screens.

The web application deployment descriptor contains entries for a login form and error page. The login form must contain fields for entering a username and a password. These fields must be named `j_username` and `j_password`, respectively.

When a user attempts to access a protected web resource, the container checks the user's authentication. If the user is authenticated and possesses authority to access the resource, the requested web resource is activated and a reference to it is returned. If the user is not authenticated, all of the following steps occur:

1. The login form associated with the security constraint is sent to the client and the URL path triggering the authentication is stored by the container.

2. The user is asked to fill out the form, including the username and password fields.

3. The client posts the form back to the server.

4. The container attempts to authenticate the user using the information from the form.

5. If authentication fails, the error page is returned using either a forward or a redirect, and the status code of the response is set to 200.

6. If authentication succeeds, the authenticated user's principal is checked to see if it is in an authorized role for accessing the resource.

7. If the user is authorized, the client is redirected to the resource using the stored URL path.

The error page sent to a user that is not authenticated contains information about the failure.

Form Based Authentication has the same lack of security as Basic Authentication since the user password is transmitted as plain text and the target server is not authenticated. Again additional protection can alleviate some of these concerns: a secure transport mechanism (HTTPS), or security at the network level (such as the IPSEC protocol or VPN strategies) is applied in some deployment scenarios.

Form based login and URL based session tracking can be problematic to implement. Form based login should be used only when sessions are being maintained by cookies or by SSL session information.

In order for the authentication to proceed appropriately, the action of the login form must always be `j_security_check`. This restriction is made so that the login form will work no matter which resource it is for, and to avoid requiring the server to specify the action field of the outbound form.

Here is an example showing how the form should be coded into the HTML page:

```
<form method='post' action='j_security_check'>
        <input type='text' name='j_username'>
        <input type='password' name='j_password'>
</form>
```

```
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>User Auth</web-resource-name>
                        <url-pattern>/auth/*</url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>manager</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                <auth-method>FORM</auth-method>
                <realm-name>User Auth</realm-name>
                <form-login-config>
                        <form-login-page>login.jsp</form-login-page>
                        <form-error-page>error.jsp</form-error-page>
                </form-login-config>
        </login-config>

        <security-role>
                <role-name>admin</role-name>
        </security-role>
        <security-role>
                <role-name>manager</role-name>
        </security-role>
</web-app>
```

Chapter 6. The JavaServer Pages (JSP) Technology Model

Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c) standard and custom actions, and (d) expression language elements.

Template text.

The semantics of template (or uninterpreted) text is very simple: the template text is passed through to the current `out JspWriter` implicit object, after applying the substitutions of Quoting and Escape Conventions.

XML syntax:

```
<jsp:text>
        hi you all
</jsp:text>
```

Quoting in Template Text:

- A literal `<%` is quoted by `<\%`
- Only when the EL is enabled for a page, a literal `$` can be quoted by `\$`. This is not required but is useful for quoting EL expressions.

Scripting elements.

Scripting elements are commonly used to manipulate objects and to perform computation that affects the content generated.

Disabling scripting elements can be done by setting the `scripting-invalid` element to `true` in the JSP configuration. For example, the following `web.xml` fragment defines a group that disables scripting elements for all JSP pages delivered using the `.jsp` extension:

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

There are three classes of scripting elements: *declarations*, *scriptlets* and *expressions*.

Declarations are used to declare scripting language constructs that are available to all other scripting elements. Scriptlets are used to describe actions to be performed in response to some request. Scriptlets that are program fragments can also be used to do things like iterations and conditional execution of other elements in the JSP page. Expressions are complete expressions in the scripting language that get evaluated at response time; commonly, the result is converted into a string and inserted into the output stream.

Each scripting element has a <%-based syntax as follows:

```
<%! this is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

White space is optional after <%!, <%, and <%=, and before %>.

XML syntax:

```
<jsp:declaration> declaration goes here </jsp:declaration>
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
<jsp:expression> expression goes here </jsp:expression>
```

Declarations.

Declarations are used to declare VARIABLES and METHODS in the scripting language used in a JSP page. A declaration must be a complete declarative statement, or sequence thereof, according to the syntax of the scripting language specified. Declarations DO NOT produce any output into the current out stream. Declarations are initialized when the JSP page is initialized and are made available to other declarations, scriptlets, and expressions.

For example, the following declaration below declares an integer, global to the page:

```
<%! int i; %>
```

The following declaration does the same and initializes it to zero. This type of initialization should be done with care in the presence of multiple requests on the page:

```
<%! int i = 0; %>
```

The next declaration declares a method GLOBAL to the page:

```
<%!
        public String someMethod(int i) {
                if (i<3) return("...");
                ...
        }
%>
```

Scriptlets.

Scriptlets can contain any code fragments that are valid for the scripting language specified in the language attribute of the page directive.

Scriptlets are executed at request-processing time. Whether or not they produce any output into the out stream depends on the code in the scriptlet. Scriptlets CAN have side-effects, modifying the objects visible to them.

When all scriptlet fragments in a given translation unit are combined in the order they appear in the JSP page, they must yield a valid statement, or sequence of statements, in the specified scripting language.

Here is a simple example where the page changed dynamically depending on the time of day.

```
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) {%>
        Good Morning
<% } else { %>
        Good Afternoon
<% } %>
```

A scriptlet can also have a LOCAL variable declaration, for example the following scriptlet just declares and initializes an integer, and later increments it.

```
<% int i; i= 0; %>
About to increment i...
<% i++; %>
```

Expressions.

An expression element in a JSP page is a scripting language expression that is evaluated and the result is coerced to a `String`. The result is subsequently emitted into the current `out JspWriter` object.

If the result of the expression cannot be coerced to a `String` the following must happen: If the problem is detected at translation time, a translation time error shall occur. If the coercion cannot be detected during translation, a `ClassCastException` shall be raised at request time.

A scripting language may support side-effects in expressions when the expression is evaluated. Expressions are evaluated left-to-right in the JSP page. If an expression appears in more than one run-time attribute, they are evaluated left-to-right in the tag. An expression might change the value of the out object, although this is not something to be done lightly.

The expression must be a complete expression in the scripting language in which it is written, or a translation error must occur.

Expressions are evaluated at request processing time. The value of an expression is converted to a `String` and inserted at the proper position in the `.jsp` file.

This example inserts the current date:

```
<%= (new java.util.Date()).toLocaleString() %>
```

Comments.

There are two types of comments in a JSP page: comments to the JSP page itself, documenting what the page is doing; and comments that are intended to appear in the generated document sent to the client.

In order to generate comments that appear in the response output stream to the requesting client, the HTML and XML comment syntax is used, as follows:

```
<!-- HTML comments ... -->
```

These comments are treated as uninterpreted template text by the JSP container. Dynamic content that appears within HTML/XML comments, such as actions, scriptlets and expressions, is still processed by the container. If the generated comment is to have dynamic data, this can be obtained through an expression syntax, as in:

```
<!-- comments <%= expression %> more comments ... -->
```

A JSP comment is of the form:

```
<%-- page code comments --%>
```

The body of the content is ignored completely. Comments are useful for documentation but also are used to 'comment out' some portions of a JSP page. Note that JSP comments do not nest.

An alternative way to place a comment in JSP is to use the comment mechanism of the scripting language. For example:

```
<% /* this is a code comment */ %>
```

Directives.

Directives are messages to the JSP container. Directives have this syntax:

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

There may be optional white space after the `<%@` and before `%>`.

Directives DO NOT produce any output into the current out stream.

There are three directives: the `page`, the `taglib` and the `include`.

The `page` Directive.

The page directive defines a number of page dependent properties and communicates these to the JSP container.

This `<jsp:directive.page>` element describes the same information following the XML syntax.

A translation unit (JSP source file and any files included via the `include` directive) can contain more than one instance of the `page` directive, all the attributes will apply to the complete translation unit (i.e. `page` directives are position independent). An exception to this position independence is the use of the `pageEncoding` and `contentType` attributes in the determination of the page character encoding; for this purpose, they should appear at the beginning of the page. There shall be ONLY ONE occurrence of any attribute/value pair defined by this directive in a given translation unit, unless the values for the duplicate attributes are IDENTICAL for all occurrences. The `import` and `pageEncoding` attributes are exempt from this rule and can appear multiple times. Multiple uses of the `import` attribute are cumulative (with ordered set union semantics). The `pageEncoding` attribute can occur at most once per file (or a translation error will result), and applies only to the file in which it appears. Other such multiple attribute/value (re)definitions result in a fatal translation error if the values do not match.

The following directive provides some user-visible information on this JSP page:

```
<%@ page info="my latest JSP Example" %>
```

The following directive requests no buffering, and provides an error page.

```
<%@ page buffer="none" errorPage="/oops.jsp" %>
```

The following directive indicates that the scripting language is based on Java, that the types declared in the package `com.myco` are directly available to the scripting code, and that a buffering of 16KB should be used.

```
<%@ page language="java" import="com.myco.*" buffer="16kb" %>
```

```
<%@ page page_directive_attr_list %>

page_directive_attr_list ::=
        { language="scriptingLanguage" }
        { extends="className" }
        { import="importList" }
        { session="true|false" }
        { buffer="none|sizekb" }
        { autoFlush="true|false" }
        { isThreadSafe="true|false" }
        { info="info_text" }
        { errorPage="error_url" }
        { isErrorPage="true|false" }
        { contentType="ctinfo" }
        { pageEncoding="peinfo" }
        { isELIgnored="true|false" }
```

The `taglib` Directive.

The set of significant tags a JSP container interprets can be extended through a tag library.

The `taglib` directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.

If a JSP container implementation cannot locate a tag library description, a fatal translation error shall result.

It is a fatal translation error for the `taglib` directive to appear after actions or functions using the prefix.

In the following example, a tag library is introduced and made available to this page using the `super` prefix; no other tag libraries should be introduced in this page using this prefix. In this particular case, we assume the tag library includes a `doMagic` element type, which is used within the page.

```
<%@ taglib uri=”http://www.mycorp/supertags” prefix=”super” %>
...
<super:doMagic>
...
</super:doMagic>
```

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

`tagdir` indicates this prefix is to be used to identify tag extensions installed in the `/WEB-INF/tags/` directory or a subdirectory. An implicit tag library descriptor is used. A translation error must occur if the value does not start with `/WEB-INF/tags/`. A translation error must occur if the value does not point to a directory that exists. A translation error must occur if used in conjunction with the `uri` attribute.

The `include` Directive.

The `include` directive is used to substitute text and/or code at JSP page translation-time. The `<%@ include file="relativeURLspec" %>` directive inserts the text of the specified resource into the page or tag file. The included file is subject to the access control available to the JSP container.

The `<jsp:directive.include>` element describes the same information following the XML syntax.

The following example requests the inclusion, at translation time, of a copyright file. The file may have elements which will be processed too:

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

Syntax:

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

Summary of Include Mechanisms in JSP 2.0:

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

- file relative
- static
- Content IS parsed by JSP container

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

- page relative
- static OR dynamic
- Content is NOT parsed - it is included in place

Write JSP code that uses the directives: (a) 'page' (with attributes 'import', 'session', 'contentType', and 'isELIgnored'), (b) 'include', and (c) 'taglib'.

`page` Directive with `import` attribute.

Lets you specify the packages that should be imported by the servlet into which the JSP page gets translated. Generates `import` statements at top of servlet definition.

Default imports: `java.lang.*`; `javax.servlet.*`; `javax.servlet.jsp.*`; `javax.servlet.http.*`.

```
<%@ page import="package.class" %>
```

or multiple classes/packages (separated by comma, NOT semicolon).

```
<%@ page import="package.classA, package.classB, other.package.*" %>
```

page Directive with session attribute.

Controls whether or not page participates in HTTP sessions. Indicates that `session` (of type `HttpSession`) should be bound to existing session.

`false` value means NO sessions will be used automatically. Attempts to access session variable by servlet will cause RUN-TIME failure.

By default, it IS part of a session. All related pages have to do this for it to be useful.

```
<%@ page session="true" %> <%-- default !!! --%>
```

or

```
<%@ page session="false" %>
```

page Directive with contentType attribute.

Specify the MIME type of the page generated by the servlet that results from the JSP page. Attribute value cannot be computed at request time.

```
<%@ page contentType="MIME-Type" %>
```

or

```
<%@ page contentType="MIME-Type; charset=Character-Set" %>
```

it is the same as (scriptlet):

```
<% response.setContentType("MIME-Type; charset=Character-Set"); %>
```

Note, you CANNOT use the `contentType` attribute for conditional run-time task. The following ALWAYS results in the Excel MIME type (`page` directive is evaluated only once during translation phase, and not during execution phase):

```
<% boolean usingExcel = checkUserRequest(request); %>
<% if (usingExcel) { %>
        <%@ page contentType="application/vnd.ms-excel" %>
<% } %>
```

the following approach should be used instead :

```
<% boolean usingExcel = checkUserRequest(request); %>
<% if (usingExcel) { %>
        <% response.setContentType("application/vnd.ms-excel"); %>
<% } %>
```

`page` Directive with `isELIgnored` attribute.

The attribute is used to control whether the JSP 2.0 Expression Language (EL) is ignored (`true`) or evaluated normally (`false`).

EL expressions will be ignored by default in JSP 1.2 applications. When upgrading a web application to JSP 2.0, EL expressions WILL BE INTERPRETED by default. The escape sequence '\$' can be used to escape EL expressions that should not be interpreted by the container. Alternatively, the `isELIgnored` `page` directive attribute, or the `<el-ignored>` configuration element can be used to deactivate EL for entire translation units.

```
<%@ page isELIgnored="false" %> <!-- default for JSP 2.0 -->
```

or

```
<%@ page isELIgnored="true" %> <!-- default for JSP 1.2 -->
```

`include` Directive.

Lets you insert a file into servlet class at time the JSP file is translated into servlet. Should be placed in document at point where you want file inserted.

The `include` directive lets you reuse navigation bars, tables, and other elements in multiple pages. The `include` directive includes a file in a JSP document at DOCUMENT TRANSLATION TIME. Included file can contain JSP code. Inclusion is recursive: included files may include files.

```
<html>
<head>
        <title>Including Files at Translation Time (JSP)</title>
</head>
<body>
        <%@ include file="somePage.jsp" %>
</body>
</html>
```

`taglib` Directive.

Can be used to define custom tags.

```
<%@ taglib prefix="example" uri="http://www.server.com/example-taglib" %>
```

and `web.xml`:

```
<taglib>
  <taglib-uri>http://www.server.com/example-taglib</taglib-uri>
  <taglib-location>/WEB-INF/example-taglib.tld</taglib-location>
</taglib>
```

Notice the `taglib-location` specifies the location of the TLD. The `taglib-uri` is, for the most part, an arbitrary name given to the tag library. The name you give it can't conflict with other tag libraries in your deployment descriptor. In fact, adding the `taglib` element to the deployment descriptor is actually optional. You could instead reference the TLD directly in the taglib directive:

```
<%@ taglib prefix="example" uri="/WEB-INF/example-taglib.tld" %>
```

This isn't recommended because it reduces flexibility if you ever choose to rename or move the TLD. The uri would have to be changed in every JSP that used it.

Write a JSP Document (XML-based document) that uses the correct syntax.

JSP syntax comment:

```
<%-- comment --%>
```

has no XML syntax analog.

JSP syntax `page` directive:

```
<%@ page ... %>
```

XML syntax `page` directive:

```
<jsp:directive.page ... />
```

JSP syntax `taglib` directive:

```
<%@ taglib ... %>
```

has no XML syntax analog.

`jsp:root` element is annotated with namespace information:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
        xmlns:eg="http://java.apache.org/tomcat/examples-taglib"
        xmlns:test="urn:jsptld:/tomcat/taglib"
        xmlns:temp="urn:jsptld:/WEB-INF/tlds/my.tld"
        version="2.0">

        ...

</jsp:root>
```

A `taglib` directive of the form

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

is translated into an `xmlns:prefix` attribute on the root of the JSP document, with a value that depends on `uriValue`.

- If `uriValue` is a RELATIVE path, then the value used is `urn:jsptld:uriValue`.
- If `uriValue` is a ABSOLUTE path, then `uriValue` is used directly.

A `taglib` directive of the form:

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

is translated into an `xmlns:prefix` attribute on the root of the JSP document, with a value of the form `urn:jsptagdir:tagDirValue`.

JSP syntax `include` directive:

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

XML syntax `include` directive:

```
<jsp:directive.include file="relativeURL" />
```

A file on the local system to be included when the JSP page is translated into a servlet. The URL must be a relative one.

JSP syntax declaration:

```
<%! ... %>
```

XML syntax declaration:

```
<jsp:declaration> ... </jsp:declaration>
```

Declarations are translated into a `jsp:declaration` element. For example:

```
<%!
        public String someFunc(int i) {
                if (i<3) return("...");
        }
%>
```

is translated into the following:

```
<jsp:declaration>
        <![CDATA[
                public String someFunc(int i) {
                        if (i<3) return("...");
                }
        ]]>
</jsp:declaration>
```

Alternatively, we could use an `&lt;` and instead say:

```
<jsp:declaration>
        public String someFunc(int i) {
                if (i&lt;3) return("...");
        }
</jsp:declaration>
```

JSP syntax scriptlet:

```
<% ... %>
```

XML syntax scriptlet:

```
<jsp:scriptlet> ... </jsp:scriptlet>
```

JSP syntax expression:

```
<%= ... %>
```

XML syntax expression:

```
<jsp:expression> ... </jsp:expression>
```

JSP syntax template text:

```
some template text
```

XML syntax template text:

```
<jsp:text>
        some template text
</jsp:text>
```

A JSP document syntax:

```
<html>
        <title>positiveTagLib</title>
        <body>
                <%@ taglib uri="http://java.apache.org/tomcat/examples-taglib"
                        prefix="eg" %>
                <%@ taglib uri="/tomcat/taglib" prefix="test" %>
                <%@ taglib uri="WEB-INF/tlds/my.tld" prefix="temp" %>
                <eg:test toBrowser="true" att1="Working">
                Positive Test taglib directive </eg:test>
        </body>
</html>
```

A XML document syntax:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
        xmlns:eg="http://java.apache.org/tomcat/examples-taglib"
        xmlns:test="urn:jsptld:/tomcat/taglib"
        xmlns:temp="urn:jsptld:/WEB-INF/tlds/my.tld"
        version="2.0">

        <jsp:text><![CDATA[
<html>
        <title>positiveTagLib</title>
        <body>
        ]]></jsp:text>
        <eg:test toBrowser="true" att1="Working">
                <jsp:text>Positive test taglib directive</jsp:text>
        </eg:test>

        <jsp:text><![CDATA[
        </body>
</html>
        ]]></jsp:text>

</jsp:root>
```

Describe the purpose and event sequence of the JSP page life cycle: (1) JSP page translation, (2) JSP page compilation, (3) load class, (4) create instance, (5) call the `jspInit` method, (6) call the `_jspService` method, and (7) call the `jspDestroy` method.

JSP page translation.

During the translation phase the container locates or creates the JSP page implementation class that corresponds to a given JSP page. This process is determined by the semantics of the JSP page. The container interprets the standard directives and actions, and the custom actions referencing tag libraries used in the page. A tag library may optionally provide a validation method acting on the XML View of a JSP page, to validate that a JSP page is correctly using the library.

The translation of a JSP source page into its implementation class can occur at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page.

The JSP container creates a JSP page implementation class for each JSP page. The name of the JSP page implementation class is implementation dependent. The JSP Page implementation object belongs to an implementation-dependent named package. The package used may vary between one JSP and another, so minimal assumptions should be made. As of JSP 2.0, it is illegal to refer to any classes from the unnamed (a.k.a. default) package. This may result in a translation error on some containers,

The contract on the JSP page implementation class:

- Implements `HttpJspPage` if the protocol is HTTP, or `JspPage` otherwise.
- All of the methods in the `Servlet` interface are declared `final`.

Additionally, it is the responsibility of the JSP page author that the provided superclass satisfies:

- The `service` method of the servlet API invokes the `_jspService` method.
- The `init(ServletConfig)` method stores the configuration, makes it available via `getServletConfig`, then invokes `jspInit`.
- The `destroy` method invokes `jspDestroy`.

A JSP container may give a fatal translation error if it detects that the provided superclass does not satisfy these requirements, but most JSP containers will not check them.

JSP page compilation.

A JSP page may be compiled into its implementation class plus deployment information during development (a JSP page can also be compiled at deployment time). In this way JSP page authoring tools and JSP tag libraries may be used for authoring servlets. The benefits of this approach include:

- Removal of the start-up lag that occurs when a container must translate a JSP page upon receipt of the first request.

- Reduction of the footprint needed to run a JSP container, as the Java compiler is not needed.

Compilation of a JSP page in the context of a web application provides resolution of relative URL specifications in include directives and elsewhere, tag library references, and translation -time actions used in custom actions.

A JSP page can also be compiled at deployment time.

JSP class loading.

JSP servlet is loaded.

Creating instance of JSP class.

JSP servlet instance is created.

The enforcement of the contract between the JSP container and the JSP page author is aided by the requirement that the Servlet class corresponding to the JSP page must implement the `javax.servlet.jsp.HttpJspPage` interface (or the `javax.servlet.jsp.JspPage` interface if the protocol is not HTTP).

```
package javax.servlet.jsp;

public interface JspPage extends javax.servlet.Servlet {

        public void jspInit();
        public void jspDestroy();
        public void _jspService(ServletRequestSubtype request,
                ServletResponseSubtype response)
                throws ServletException, IOException;
                // _jspService - depends on the specific protocol used and
                // cannot be expressed in a generic way in Java.
}
```

```
package javax.servlet.jsp;

public interface HttpJspPage extends JspPage {

        public void _jspService(javax.servlet.http.HttpServletRequest req,
                javax.servlet.http.HttpServletResponse res)
                throws javax.servlet.ServletException, java.io.IOException;

}
```

The `jspInit` method.

The `jspInit` method, if present, will be called to prepare the page BEFORE the first request is delivered.

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

Method is optionally defined in JSP page. Method is invoked when the JSP page is initialized. When method is called all the methods in servlet, including `getServletConfig` are available.

The `_jspService` method.

The `_jspService(...)` method corresponds to the body of the JSP page. This method is defined automatically by the JSP container and should NEVER be defined by the JSP page author.

The formal types of the request/response parameters:

```
void _jspService(javax.servlet.ServletRequest req,
                 javax.servlet.ServletResponse res)
                 throws IOException, ServletException {
         ...
}
```

The request and response interfaces together describe a protocol-dependent contract between the JSP container and the class that implements the JSP page. The HTTP contract is defined by the `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` interfaces:

```
void _jspService(javax.servlet.http.HttpServletRequest req,
                 javax.servlet.http.HttpServletResponse res)
                 throws IOException, ServletException {
         ...
}
```

Method MAY NOT be defined in JSP page. The JSP container automatically generates this method, based on the contents of the JSP page. Method invoked at each client request.

The `jspDestroy` method.

The JSP container can reclaim resources used by a JSP page when a request is not being serviced by the JSP page by invoking its `jspDestroy` method, if present.

```
<%!
        public void jspDestroy() {
                ...
        }
%>
```

Method is optionally defined in JSP page. Method is invoked BEFORE destroying the page.

Given a design goal, write JSP code using the appropriate implicit objects: (a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext, and (i) exception.

The `request` object.

Protocol dependent subtype of `javax.servlet.ServletRequest`, e.g: `javax.servlet.http.HttpServletRequest`.

This is the `HttpServletRequest` associated with the request, and lets you look at the request parameters (via `getParameter`), the request type (GET, POST, HEAD, etc.), and the incoming HTTP headers (cookies, referer, etc.).

The request triggering the service invocation. Has a request scope.

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

The `response` object.

Protocol dependent subtype of `javax.servlet.ServletResponse`, e.g: `javax.servlet.http.HttpServletResponse`.

This is the `HttpServletResponse` associated with the response to the client. Note that, since the output stream (see `out` below) is buffered, it is legal to set HTTP status codes and response headers, even though this is not permitted in regular servlets once any output has been sent to the client.

The response to the request. Has a page scope.

The `out` object.

An object of type `javax.servlet.jsp.JspWriter`.

This is the `PrintWriter` used to send output to the client. However, in order to make the `response` object (see the previous section) useful, this is a buffered version of `PrintWriter` called `JspWriter`. Note that you can adjust the buffer size, or even turn buffering off, through use of the `buffer` attribute of the `page` directive. Also note that `out` is used almost exclusively in scriptlets, since JSP expressions automatically get placed in the output stream, and thus rarely need to refer to `out` explicitly.

An object that writes into the output stream. Has a page scope.

```
<html>
        <body>

                <% out.println("Hello !"); %>

        </body>
</html>
```

NOTE, JSP page authors are prohibited from writing directly to either the `PrintWriter` or `OutputStream` associated with the `ServletResponse`. The following example is INVALID:

```
<html>
        <body>

                <% response.getWriter().println("Hello !"); %>

        </body>
</html>
```

The `session` object.

An object of type `javax.servlet.http.HttpSession`.

This is the `HttpSession` object associated with the request. Sessions are created automatically, so this variable is bound even if there was no incoming session reference. The one exception is if you use the `session` attribute of the `page` directive to turn sessions off, in which case attempts to reference the `session` variable cause errors at the time the JSP page is translated into a servlet.

The session object created for the requesting client (if any). This variable is only valid for HTTP protocols. Has a session scope.

The `config` object.

An object of type `javax.servlet.ServletConfig`.

The `ServletConfig` for this JSP page. Has a page scope.

The `application` object.

An object of type `javax.servlet.ServletContext`.

The servlet context obtained from the servlet configuration object (as in the call `this.getServletConfig().getContext()`). Has an application scope.

```
<%

        javax.servlet.RequestDispatcher rd;
        rd = application.getRequestDispatcher("/NextPage.jsp");
        rd.forward(request, response);

%>
```

The `page` object.

An object of type `java.lang.Object`.

This is simply a synonym for `this`, and is not very useful in Java. It was created as a placeholder for the time when the scripting language could be something other than Java.

The instance of this page's implementation class processing the current request. Has a page scope.

The `pageContext` object.

A `PageContext` is an object that provides a context to store references to objects used by the page, encapsulates implementation -dependent features, and provides convenience methods. A JSP page implementation class can use a `PageContext` to run unmodified in any compliant JSP container while taking advantage of implementation-specific improvements like high performance `JspWriters`.

```
public abstract class PageContext extends JspContext {

        public abstract java.lang.Exception getException();
        public abstract java.lang.Object getPage(); // instance of Servlet
        public abstract javax.servlet.ServletRequest getRequest();
        public abstract javax.servlet.ServletResponse getResponse();
        public abstract javax.servlet.ServletConfig getServletConfig();
        public abstract javax.servlet.ServletContext getServletContext();
        public abstract javax.servlet.http.HttpSession getSession();
        public abstract void handlePageException(java.lang.Exception e);
        public abstract void include(java.lang.String relativeUrlPath);
        public abstract void forward(java.lang.String relativeUrlPath);
}
```

`PageContext` extends `JspContext` to provide useful context information for when JSP technology is used in a `Servlet` environment. A `PageContext` instance provides access to all the namespaces associated with a JSP page, provides access to several page attributes, as well as a layer above the implementation details. Implicit objects are added to the `pageContext` automatically. The `PageContext` class is an `abstract` class, designed to be extended to provide implementation dependent implementations thereof, by conformant JSP engine runtime environments. The following methods provide convenient access to implicit objects: `getException()`, `getPage()`, `getRequest()`, `getResponse()`, `getSession()`, `getServletConfig()` and `getServletContext()`. The following methods provide support for forwarding, inclusion and error handling: `forward()`, `include()`, and `handlePageException()`.

The `exception` object.

An object of type `java.lang.Throwable`.

The uncaught `Throwable` that resulted in the error page being invoked. Has a page scope.

A JSP is considered an Error Page if it sets the `page` directive's `isErrorPage` attribute to `true`. If a `page` has `isErrorPage` set to `true`, then the `exception` implicit scripting language variable of that page is initialized.

`error.jsp`:

```
<%@ page isErrorPage="true" %>
...
<%= exception.getMessage() %> <br>

With the following stack trace: <br>

<%
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        exception.printStackTrace(new PrintStream(baos));
        out.print(baos);
%>
```

Configure the deployment descriptor to declare one or more tag libraries, deactivate the evaluation language, and deactivate the scripting language.

A tag library is a collection of actions that encapsulate some functionality to be used from within a JSP page. A tag library is made available to a JSP page through a `taglib` directive that identifies the tag library via a URI (Universal Resource Identifier).

The URI identifying a tag library may be any valid URI as long as it can be used to uniquely identify the semantics of the tag library.

The URI identifying the tag library is associated with a Tag Library Description (TLD) file and with tag handler classes. Both Classic and Simple Tag Handlers (implemented either in Java or as tag files) can be packaged together.

The explicit `web.xml` map provides a explicit description of the tag libraries that are being used in a web application.

The implicit map from TLDs means that a JAR file implementing a tag library can be dropped in and used immediatedly through its stable URIs.

The use of relative URI specifications in the taglib map enables very short names in the taglib directive. For example, if the map is:

```
<taglib>
        <taglib-uri>/myPRlibrary</taglib-uri>
        <taglib-location>/WEB-INF/tlds/PRlibrary_1_4.tld</taglib-location>
</taglib>
```

then it can be used as:

```
<%@ taglib uri="/myPRlibrary" prefix="x" %>
```

Finally, the fallback rule allows a `taglib` directive to refer directly to the TLD. This arrangement is very convenient for quick development at the expense of less flexibility and accountability. For example, in the case above, it enables:

```
<%@ taglib uri="/WEB-INF/tlds/PRlibrary_1_4.tld" prefix="x" %>
```

EL expressions will be ignored by default in JSP 1.2 applications. When upgrading a web application to JSP 2.0, EL expressions will be interpreted by default. The escape sequence '\$' can be used to escape EL expressions that should not be interpreted by the container. Alternatively, the `isELIgnored` page directive attribute, or the `<el-ignored>` configuration element can be used to deactivate EL for entire translation units.

The default mode for JSP pages in a Web Application delivered using a `web.xml` using the Servlet 2.4 format is to evaluate EL expressions; this automatically provides the default that most applications want.

The default mode can be explicitly changed by setting the value of the `el-ignored` element. The `el-ignored` element is a subelement of `jsp-property-group`. It has no subelements. Its valid values are `true` and `false`.

For example, the following `web.xml` fragment defines a group that deactivates EL evaluation for all JSP pages delivered using the `.jsp` extension:

```
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>true</el-ignored>
</jsp-property-group>
```

Page authors can override the default mode through the `isELIgnored` attribute of the page directive.

With the addition of the EL, some JSP page authors, or page authoring groups, may want to follow a methodology where scripting elements are not allowed.

The `scripting-invalid` element is a subelement of `jsp-property-group`. It has no subelements. Its valid values are `true` and `false`. Scripting is ENABLED by default. Disabling scripting elements can be done by setting the `scripting-invalid` element to `true` in the JSP configuration.

For example, the following `web.xml` fragment defines a group that disables scripting elements for all JSP pages delivered using the `.jsp` extension:

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

Given a specific design goal for including a JSP segment in another page, write the JSP code that uses the most appropriate inclusion mechanism (the `include` directive or the `jsp:include` standard action).

The `include` directive.

Includes a STATIC file in a JSP page, parsing the file's JSP elements. The `include` directive is processed when the JSP page is TRANSLATED into a servlet class.

JSP Syntax:

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

XML Syntax:

```
<jsp:directive.include file="relativeFileName" />
```

Example:

`include.jsp`:

```
<html>
        <head><title>An Include Test</title></head>
        <body bgcolor="white">

                The current date and time are :

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

        </body>
</html>
```

`date.jsp`:

```
<%= (new java.util.Date() ).toLocaleString() %>
```

The `jsp:include` action.

Includes a static file OR the result from another web component. The difference with the `include` directive is that not the source of the JSP, but it's output is included. The include JSP page is being executed within the servlet engine and it's output is returned to the calling page.

The main benefit of using the `jsp:include` action is that the URL to include can be constructed during execution of the page. However, every included page results in a new request to the servlet engine, which means there is a bit of a performance impact when compared to the `include` directive.

JSP Syntax:

```
<jsp:include page="{relativeURL | <%= expression %>}"
        flush="true | false" />
```

or

```
<jsp:include page="{relativeURL | <%= expression %>}"
        flush="true | false" >
        <jsp:param name="parameterName"
                value="{parameterValue | <%= expression %>}" />+
</jsp:include>
```

XML Syntax:

```
<jsp:include page="{relativeURL | %= expression %}"
        [ flush="true | false" ] />
```

or

```
<jsp:include page="{relativeURL | %= expression %}"
[ flush="true | false" ] >
        [ <jsp:param name="parameterName"
                value="{parameterValue | %= expression %}" /> ]+
</jsp:include>
```

Examples:

```
<jsp:include page="scripts/login.jsp" flush="true" />
```

The following example shows that there are two ways of passing named variables: either pass them on directly in the URL, or add them using `jsp:param` tags:

```
<jsp:include page="includes/page.jsp?param1=value" flush="true">
        <jsp:param name="param2" value="value2" />
</jsp:include>
```

It is also possible to dynamically choose the file to include. This example determines the file to include from a request parameter:

```
<jsp:include page='<%= request.getParameter("incFile") %>' />
```

Defining Implicit Includes.

The `include-prelude` element is an optional subelement of `jsp-property-group`. It has no subelements. Its value is a context-relative path that must correspond to an element in the Web Application. When the element is present, the given path will be automatically included (as in an `include` directive) at the beginning of the JSP page in the `jsp-property-group`. When there are more than one `include-prelude` element in a group, they are to be included in the order they appear. When more than one `jsp-property-group` applies to a JSP page, the corresponding includeprelude elements will be processed in the same order as they appear in the JSP configuration section of `web.xml`.

The `include-coda` element is an optional subelement of `jsp-property-group`. It has no subelements. Its value is a context-relative path that must correspond to an element in the Web Application. When the element is present, the given path will be automatically included (as in an `include` directive) at the end of the JSP page in the `jsp-property-group`. When there are more than one `include-coda` element in a group, they are to be included in the order they appear. When more than one `jsp-property-group` applies to a JSP page, the corresponding `include-coda` elements will be processed in the same order as they appear in the JSP configuration section of `web.xml`. Note that these semantics are in contrast to the way `url-patterns` are matched for other configuration elements.

Preludes and codas follow the same rules as statically included JSP segments. In particular, start tags and end tags must appear in the same file.

For example, the following `web.xml` fragment defines two groups. Together they indicate that everything in directory `/two/` have `/WEB-INF/jspf/prelude1.jspf` and `/WEB-INF/jspf/prelude2.jspf` at the beginning and `/WEB-INF/jspf/coda1.jspf` and `/WEB-INF/jspf/coda2.jspf` at the end, in that order, while other `.jsp` files only have `/WEB-INF/jspf/prelude1.jspf` at the beginning and `/WEB-INF/jspf/coda1.jspf` at the end:

```
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <include-prelude>/WEB-INF/jspf/prelude1.jspf</include-prelude>
        <include-coda>/WEB-INF/jspf/coda1.jspf</include-coda>
</jsp-property-group>
<jsp-property-group>
        <url-pattern>/two/*</url-pattern>
        <include-prelude>/WEB-INF/jspf/prelude2.jspf</include-prelude>
        <include-coda>/WEB-INF/jspf/coda2.jspf</include-coda>
</jsp-property-group>
```

Chapter 7. Building JSP Pages Using the Expression Language (EL)

Given a scenario, write EL code that accesses the following implicit variables including: `pageScope`, `requestScope`, `sessionScope`, and `applicationScope`, `param` and `paramValues`, `header` and `headerValues`, `cookie`, `initParam` and `pageContext`.

There are several implicit objects that are available to EL expressions used in JSP pages. These objects are always available under these names:

- `pageContext` - the `PageContext` object. Provides an API to access various objects including:
  - `context` - the context for the JSP page's servlet and any Web components contained in the same application.
  - `session` - the session object for the client.

    - `request` - the request triggering the execution of the JSP page.

- `pageScope` - a `java.util.Map` that maps page-scoped attribute names to their values.
- `requestScope` - a `java.util.Map` that maps request-scoped attribute names to their values.
- `sessionScope` - a `java.util.Map` that maps session-scoped attribute names to their values.
- `applicationScope` - a `java.util.Map` that maps application-scoped attribute names to their values.
- `param` - a `java.util.Map` that maps parameter names to a single `String` parameter value (obtained by calling `ServletRequest.getParameter(String name)`).
- `paramValues` - a `java.util.Map` that maps parameter names to a `String[]` of all values for that parameter (obtained by calling `ServletRequest.getParameterValues(String name)`).
- `header` - a `java.util.Map` that maps header names to a single String header value (obtained by calling `HttpServletRequest.getHeader(String name)`).
- `headerValues` - a `java.util.Map` that maps header names to a `String[]` of all values for that header.
- `cookie` - a `java.util.Map` that maps cookie names to a single `Cookie` object. Cookies are retrieved according to the semantics of `HttpServletRequest.getCookies()`. If the same name is shared by multiple cookies, an implementation must use the FIRST one encountered in the array of `Cookie` objects returned by the `getCookies()` method. However, users of the `cookie` implicit object must be aware that the ordering of cookies is currently unspecified in the servlet specification.
- `initParam` - a `java.util.Map` that maps context initialization parameter names to their `String` parameter value (obtained by calling `ServletContext.getInitParameter(String name)`).

Examples:

The request's URI (obtained from `HttpServletRequest`):

```
${pageContext.request.requestURI}
```

The value of the `numberOfItems` property of the session-scoped attribute named `cart`:

```
${sessionScope.cart.numberOfItems}
```

The context path:

```
${pageContext.request.contextPath}
```

The session-scoped attribute named '`profile`' (`null` if not found):

```
${sessionScope.profile}
```

The `String` value of the `productId` parameter, or `null` if not found:

```
${param.productId}
```

The value of the `productId` request parameter:

```
${param["productId"]}
```

The `String[]` containing all values of the `productId` parameter, or `null` if not found:

```
${paramValues.productId}
```

A collection's members can be accessed using square brackets as shown by retrieval of the `userName` parameter from the `param` object. Members of an array or `List` can be accessed if the value in square brackets can be coerced to an `int`.

```
<html>
	<head><title>Customer Profile for ${param["userName"]}</title></head>
	<body>
		...
	</body>
</html>
```

Maps can be accessed using the dot operator OR square brackets. For example, `${param.userName}` is EQUIVALENT to `${param["userName"]}`.

The `host` HTTP attribute:

```
${header["host"]}
```

Here is an example of accessing a page-scoped object that is called `pageColor`:

```
<body bgcolor="${pageScope.pageColor}">
```

it is equivalent to:

```
<body bgcolor="${pageScope['pageColor']}">
```

Given a scenario, write EL code that uses the following operators: property access (the '.' operator), collection access (the '[]' operator).

The EL borrows the JavaScript syntax for accessing structured data as either a property of an object (with the '.' operator) or as a named array element (with the ["name"] operator). JavaBeans component properties and `java.util.Map` entries, using the key as the property name, can be accessed this way. Here are some examples:

```
${myObj.myProperty}
${myObj["myProperty"]}
${myObj['myProperty']}
${myObj[varWithThePropertyName]}
```

As shown here, an EL expression must always be enclosed within '${' and '}' characters. The first three expressions access a property named `myProperty` in an object represented by a variable named `myObj`. The fourth expression access a property with a name that's held by a variable. Instead of a single variable, this syntax can be used with any expression that evaluates to the property name.

The ARRAY ACCESS operator is also used for data represented as a collection of indexed elements, such as a Java array or a `java.util.List`:

```
${myList[2]}
${myList[aVar + 1]}
```

Expressions with syntax '`${identifier[subexpression]}`' are evaluated as follows:

1. Evaluate the `identifier` and the `subexpression`; if either resolves to `null`, the expression is `null`.

2. If the `identifier` is a BEAN:

   The `subexpression` is coerced to a `String` value and that string is regarded as a name of one of the bean's properties. The `expression` resolves to the value of that property; for example, the expression `${name.["lastName"]}` translates into the value returned by `name.getLastName()`.

3. If the `identifier` is an ARRAY:

   The `subexpression` is coerced to an `int` value and the expression resolves to `identifier [subexpression]`. For example, for an array named `colors`, `colors[3]` represents the fourth object in the array. Because the `subexpression` is coerced to an `int`, you can also access that color like this: `colors["3"]`; in that case, JSTL coerces "3" into 3.

4. If the `identifier` is a LIST:

   The `subexpression` is also coerced to an `int` and the expression resolves to the value returned from `identifier.get(subexpression)`, for example: `colorList[3]` and `colorList["3"]` both resolve to the fourth element in the list.

5. If the identifier is a MAP:

   The `subexpression` is regarded as one of the map's keys. That expression is not coerced to a value because map keys can be any type of object. The expression evaluates to `identifier.get (subexpression)`, for example, `colorMap[Red]` and `colorMap["Red"]`. The former expression is valid only if a scoped variable named `Red` exists in one of the four JSP scopes and was specified as a key for the map named `colorMap`.

Table 7.1. Summary of [] collection access operator

Identifier type	Example use	Method invoked

JavaBean	`${colorBean.red}`
	
	`${colorBean["red"]}`
	
	`${colorBean['red']}`
		`colorBean.getRed()`

Array	${colorArray[2]} ${colorArray["2"]}	Array.get(colorArray, 2)
List	${colorList[2]} ${colorList["2"]}	colorList.get(2)
Map	${colorMap[red]}	colorMap.get(pageContext.findAttribute("red"))
	${colorMap["red"]}	colorMap.get("red")

You access a map's values through its keys, which you can specify with the [] operator, for example, in table above, `${colorMap[red]}` and `${colorMap["red"]}`. The former specifies an IDENTIFIER for the key, whereas the latter specifies a STRING. For the identifier, the `PageContext.findAttribute` method searches all FOUR JSP scopes (searching the page, request, session, and application scopes) for a scoped variable with the name that you specify, in this case, `red`. On the other hand, if you specify a string, it's passed directly to the map's `get` method.

Given a scenario, write EL code that uses the following operators: aritmetic operators, relational operators, and logical operators.

There are the arithmetic operators here: '+', '-', '*', '/', '%'. You can also use the following for the '/' (division) and '%' (remainder or modulo) operators: `div` and `mod`. You can see examples of these being used below:

```
6 + 7 = ${6+7}<br>
8 x 9 = ${8*9}<br>
```

The relational operators are shown below:

Table 7.2. The relational operators

Symbol version	Text Version
==	eq
!=	ne
<	lt
>	gt
>=	ge
<=	le

Here are some basic comparisons:

```
Is 1 less than 2? ${1<2} <br>
Does 5 equal 5? ${5==5} <br>
Is 6 greater than 7? ${6 gt 7}<br>
```

The logical operators are the same as the Java Programming Language, but they also have their textual equivalents within the EL.

Table 7.3. The logical operators

Symbol version	Text Version
&&	and
||	or
!	not

The `empty` operator allows you to test the following:

- Object references to see if they are `null`.
- Strings to see if they are empty.
- Arrays to see if they are empty.
- Lists to see if they are empty.
- Maps to see if they are empty.

You use the operator in the following way:

```
empty variableName
```

If any of the above conditions are met, then the operator returns `true`.

Given a scenario, write EL code that uses a function; write code for an EL function; and configure the EL function in a tag library descriptor.

The EL has qualified functions, reusing the notion of qualification from XML namespaces (and attributes), XSL functions, and JSP custom actions. Functions are mapped to `public static` methods in Java classes. In JSP 2.0 the map is specified in the TLD. The full syntax:

```
ns:func(a1, a2, ..., an)
```

As with the rest of EL, this element can appear in attributes and directly in template text.

The prefix `ns` must match the prefix of a tag library that contains a function whose name and signature matches the function being invoked (`func`), or a translation error must occur. If the prefix is omitted, the tag library associated with the default namespace is used (this is only possible in JSP documents).

In the following standard syntax example, `func1` is associated with `some-taglib`:

```
<%@ taglib prefix="some" uri="http://acme.com/some-taglib" %>

${some:func1(true)}
```

In the following JSP document example, both `func2` and `func3` are associated with `default-taglib`:

```
<some:tag xmlns="http://acme.com/default-taglib"
        xmlns:some="http://acme.com/some-taglib"
        xmlns:jsp="http://java.sun.com/JSP/Page">
```

```
                <some:other value="${func2(true)}">
                        ${func3(true)}
                </some:other>

</some:tag>
```

The Tag Library Descriptor (TLD) associated with a tag library lists the functions.

Each such function is given a name (as seen in the EL), and a `static` method in a specific class that will implement the function. The class specified in the TLD must be a `public` class, and must be specified using a fully-qualified class name (INCLUDING PACKAGES). The specified method must be a `public static` method in the specified class, and must be specified using a fully-qualified return type followed by the method name, followed by the fully-qualified argument types in parenthesis, separated by COMMAS. Failure to satisfy these requirements shall result in a translation-time error.

A tag library can have only one function element in the same tag library with the same value for their name element. If two functions have the same name, a translation-time error shall be generated.

The expression language allows you to define functions that can be invoked in an expression. Functions must be programmed as a `public static` method in a `public` class. Once the function is developed, its signature is mapped in a Tag Library Descriptor (TLD). Write class with STATIC function:

```
package com.example;

public class MyELFunctions {
        public static String concat(String str1, String str2) {
                return str1 + str2;
        }
}
```

In order to use `concat` method we have to add a `function` element to our tag library descriptor (TLD). You'll have to create a TLD file if it doesn't already exist. A tag library descriptor defines and configures tags in a tag library. Here is `/WEB-INF/example-taglib.tld`:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>1.0</tlib-version>
        <function>
                <description>Concatenates two strings</description>
                <name>concat</name>
                <function-class>com.example.MyELFunctions</function-class>
                <function-signature>
                        java.lang.String concat(java.lang.String, java.lang.Stri
                </function-signature>
        </function>
</taglib>
```

Add a `taglib` element to the deployment descriptor (`/WEB-INF/web.xml`):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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-app_2_4.xsd"
        version="2.4">

        <taglib>
                <taglib-uri>http://www.server.com/example-taglib</taglib-uri>
                <taglib-location>/WEB-INF/example-taglib.tld</taglib-location>
        </taglib>

</web-app>
```

Notice the `taglib-location` specifies the location of the TLD. The `taglib-uri` is, for the most part, an arbitrary name given to the tag library. The name you give it can't conflict with other tag libraries in your deployment descriptor. Adding the `taglib` element to the deployment descriptor is actually OPTIONAL. You could instead reference the TLD directly in the `taglib` directive on JSP:

```
<%@ taglib prefix="my" uri="/WEB-INF/example-taglib.tld" %>
```

This is NOT recommended because it reduces flexibility if you ever choose to rename or move the TLD. The uri would have to be changed in every JSP that used it.

The new JSP looks like the following:

```
<%@ taglib prefix="my" uri="http://www.server.com/example-taglib" %>
<html>
        <head><title>EL Function example</title></head>
        <body>
                str1 is : ${param["str1"]} <br>
                str2 is : ${param["str2"]} <br>
                concatenated : ${my:concat(param["str1"], param["str2"])}
        </body>
</html>
```

The prefix '`my`' given in the `taglib` directive is whatever you choose to distinguish it from tags and functions in other tag libraries used in the JSP.

Chapter 8. Building JSP Pages Using Standard Actions

Given a design goal, create a code snippet using the following standard actions: jsp:useBean (with attributes: 'id', 'scope', 'type', and 'class'), `jsp:getProperty`, and `jsp:setProperty` (with all attribute combinations).

Action `jsp:useBean`.

A `jsp:useBean` action associates an instance of a Java programming language object defined within a given `scope` and available with a given id with a newly declared scripting variable of the same `id`.

When a `jsp:useBean` action is used in an scriptless page, or in an scriptless context (as in the body of an action so indicated), there are no Java scripting variables created but instead an EL variable is created.

The `jsp:useBean` action is quite flexible; its exact semantics depends on the attributes given. The basic semantic tries to find an existing object using `id` and `scope`. If the object is not found it will attempt to create the object using the other attributes.

It is also possible to use this action to give a local name to an object defined elsewhere, as in another JSP page or in a servlet. This can be done by using the `type` attribute and not providing `class` or `beanName` attributes.

At least ONE of `type` and `class` MUST be present, and it is NOT VALID to provide both `class` and `beanName`. If `type` and `class` are present, `class` must be assignable to `type` (in the Java platform sense). For it not to be assignable is a translation time error.

The attribute `beanName` specifies the name of a Bean, as specified in the JavaBeans specification. It is used as an argument to the `instantiate` method in the `java.beans.Beans` class. It must be of the form `a.b.c`, which may be either a class, or the name of a resource of the form `a/b/c.ser` that will be resolved in the current `ClassLoader`. If this is not true, a request-time exception, as indicated in the semantics of the instantiate method will be raised. The value of this attribute can be a request-time attribute expression.

In the following example, a Bean with name `connection` of type `com.myco.myapp.Connection` is available after actions on this element, either because it was already created and found, or because it is newly created.

```
<jsp:useBean id="connection" class="com.myco.myapp.Connection" />
```

In the next example, the timeout property is set to 33 if the Bean was instantiated:

```
<jsp:useBean id="connection" class="com.myco.myapp.Connection">
        <jsp:setProperty name="connection" property="timeout" value="33">
</jsp:useBean>
```

In the following example, the object should have been present in the session. If so, it is given the local name `wombat` with `WombatType`. A `ClassCastException` may be raised if the object is of the wrong class, and an `InstantiationException` may be raised if the object is not defined:

```
<jsp:useBean id="wombat" type="my.WombatType" scope="session" />
```

Syntax:

```
<jsp:useBean id="name" scope="page|request|session|application" typeSpec />

typeSpec ::=
        class="className" |
        class="className" type="typeName" |
        type="typeName" class="className" |
        beanName="beanName" type="typeName" |
        type="typeName" beanName="beanName" |
        type="typeName"
```

or

```
<jsp:useBean id="name" scope="page|request|session|application" typeSpec >
        [body]
</jsp:useBean>
```

In this case, the [body] will be invoked if the Bean denoted by the action is CREATED. Typically, the body will contain either scriptlets or `jsp:setProperty` tags that will be used to modify the newly created object, but the contents of the body are not restricted.

Table 8.1. `jsp:useBean` attributes

Attribute	Description
id	The name used to identify the object instance in the specified scope's namespace, and also the scripting variable name declared and initialized with that object reference. The name specified is case sensitive and shall conform to the current scripting language variable-naming conventions.
scope	The scope within which the reference is available. The DEFAULT value is `page`. See the description of the `scope` attribute defined earlier herein. A translation error must occur if scope is not one of `page`, `request`, `session` or `application`.
class	The fully qualified name of the class that defines the implementation of the object. The class name is case sensitive. If the `class` and `beanName` attributes are not specified the object must be present in the given scope.
beanName	The name of a bean, as expected by the instantiate method of the `java.beans.Beans` class. This attribute can accept a request-time attribute expression as a value.
type	If specified, it defines the type of the scripting variable defined. This allows the type of the scripting variable to be distinct from, but related to, the type of the implementation class specified. The type is required to be either the class itself, a superclass of the class, or an interface implemented by the class specified. The object referenced is required to be of this type, otherwise a `java.lang.ClassCastException` shall occur at request time when the assignment of the object referenced to the scripting variable is attempted. If unspecified, the value is the same as the value of the class attribute.

Action `jsp:getProperty`.

The `jsp:getProperty` action places the value of a bean instance property, converted to a `String`, into the implicit `out` object, from which the value can be displayed as output. The bean instance must be defined as indicated in the `name` attribute before this point in the page (usually via a `jsp:useBean` action).

The conversion to `String` is done as in the `println` methods, i.e. the `toString()` method of the object is used for `Object` instances, and the primitive types are converted directly.

If the object is not found, a request-time exception is raised.

The value of the `name` attribute in `jsp:setProperty` and `jsp:getProperty` will refer to an object that is obtained from the `pageContext` object through its `findAttribute` method.

The object named by the name must have been "introduced" to the JSP processor using either the `jsp:useBean` action or a custom action with an associated `VariableInfo` entry for this name. If the object was not introduced in this manner, the container implementation is recommended (but not required) to raise a translation error, since the page implementation is in violation of the specification.

NOTE, a consequence of the previous paragraph is that objects that are stored in, say, the session by a front component are not automatically visible to `jsp:setProperty` and `jsp:getProperty` actions in that page unless a `jsp:useBean` action, or some other action, makes them visible.

If the JSP processor can ascertain that there is an alternate way guaranteed to access the same object, it can use that information. For example it may use a scripting variable, but it must guarantee that no intervening code has invalidated the copy held by the scripting variable. The truth is always the value held by the `pageContext` object.

Examples:

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

Syntax:

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

Table 8.2. `jsp:getProperty` attributes

Attribute	Description
name	The name of the object instance from which the property is obtained.
property	Names the property to get.

Action `jsp:setProperty`.

The `jsp:setProperty` action sets the values of properties in a bean. The `name` attribute that denotes the bean must be defined before this action appears.

There are two variants of the `jsp:setProperty` action. Both variants set the values of one or more properties in the bean based on the type of the properties. The usual bean introspection is done to discover what properties are present, and, for each, its name, whether it is simple or indexed, its type, and the setter and getter methods.

Properties in a Bean can be set from one or more parameters in the request object, from a `String` constant, or from a computed request-time expression. Simple and indexed properties can be set using `jsp:setProperty`.

When assigning values to indexed properties the value must be an array.

The following two actions set a value from the request parameter values:

```
<jsp:setProperty name="request" property="*" />
<jsp:setProperty name="user" property="user" param="username" />
```

The following two elemenst set a property from a value:

```
<jsp:setProperty name="results" property="col" value="${i mod 4}" />
<jsp:setProperty name="results" property="row" value="<%= i/4 %>" />
```

Syntax:

```

<jsp:setProperty name="beanName" prop_expr />

prop_expr ::=
        property="*" |
        property="propertyName" |
        property="propertyName" param="parameterName" |
        property="propertyName" value="propertyValue"

        propertyValue ::= string
```

Table 8.3. `jsp:setProperty` attributes

Attribute	Description
name	The name of a bean instance defined by a `jsp:useBean` action or some other action. The bean instance must contain the property to be set. The defining action must appear before the `jsp:setProperty` action in the same file.
property	The name of the property whose value will be set. If `propertyName` is set to '*' then the tag will iterate over the current `ServletRequest` parameters, matching parameter names and value type(s) to property names and setter method type(s), setting each matched property to the value of the matching parameter. If a parameter has a value of "", the corresponding property is not modified.
param	The name of the request parameter whose value is given to a bean property. The name of the request parameter usually comes from a web form. If param is omitted, the request parameter name is assumed to be the same as the bean property name. If the param is not set in the `Request` object, or if it has the value of "", the `jsp:setProperty` action has no effect. An action MAY NOT have both `param` and `value` attributes.
value	The value to assign to the given property. This attribute can accept a request-time attribute expression as a value. An action MAY NOT have both `param` and `value` attributes.

Given a design goal, create a code snippet using the following standard actions: jsp:include, jsp:forward, and jsp:param.

Action `jsp:include`.

A `jsp:include` action provides for the inclusion of static and dynamic resources in the same context as the current page.

Inclusion is into the current value of `out`. The resource is specified using a `relativeURLspec` that is interpreted in the context of the web application (i.e. it is mapped).

The page attribute of both the `jsp:include` and the `jsp:forward` actions are interpreted relative to the current JSP PAGE, while the `file` attribute in an `include` directive is interpreted relative to the current JSP FILE.

An included page cannot change the response status code or set headers. This precludes invoking methods like `setCookie`. Attempts to invoke these methods will be IGNORED. The constraint is equivalent to the one imposed on the `include` method of the `RequestDispatcher` class.

A `jsp:include` action may have `jsp:param` subelements that can provide values for some parameters in the request to be used for the inclusion.

Request processing resumes in the calling JSP page, once the inclusion is completed.

The `flush` attribute controls flushing. If `true`, then, if the page output is buffered and the `flush` attribute is given a `true` value, then the buffer is flushed prior to the inclusion, otherwise the buffer is not flushed. The default value for the flush attribute is `false`.

Examples:

```
<jsp:include page="/templates/copyright.html" />
```

The above example is a simple inclusion of an object. The path is interpreted in the context of the Web Application. It is likely a static object, but it could be mapped into, for instance, a servlet via `web.xml`.

Syntax:

```
<jsp:include page="urlSpec" flush="true|false" />
```

and

```
<jsp:include page="urlSpec" flush="true|false">
        { <jsp:param .... /> }*
</jsp:include>
```

The first syntax just does a request-time inclusion. In the second case, the values in the param subelements are used to augment the request for the purposes of the inclusion.

Table 8.4. `jsp:include` attributes

Attribute	Description
page	The URL is a relative `urlSpec`. Relative paths are interpreted relative to the current JSP page. Accepts a request-time attribute value (which must evaluate to a `String` that is a relative URL specification).
flush	Optional boolean attribute. If the value is `true`, the buffer is flushed now. The default value is `false`.

Action `jsp:forward`.

A `jsp:forward` action allows the runtime dispatch of the current request to a static resource, a JSP page or a Java servlet class in the same context as the current page. A `jsp:forward` effectively terminates the execution of the current page.

The request object will be adjusted according to the value of the `page` attribute.

A `jsp:forward` action may have `jsp:param` subelements that can provide values for some parameters in the request to be used for the forwarding.

If the page output is buffered, the buffer is CLEARED prior to forwarding.

If the page output is buffered and the buffer was flushed, an attempt to forward the request will result in an `IllegalStateException`.

If the page output was unbuffered and anything has been written to it, an attempt to forward the request will result in an `IllegalStateException`.

The following action might be used to forward to a static page based on some dynamic condition:

```
<% String whereTo = "/templates/" + someValue; %>
<jsp:forward page="<%= whereTo %>" />
```

Syntax:

```
<jsp:forward page="relativeURLspec" />
```

and

```
<jsp:forward page="urlSpec">
        { <jsp:param .... /> }*
</jsp:forward>
```

Table 8.5. `jsp:forward` attributes

Attribute	Description
page	The URL is a relative `urlSpec`. Relative paths are interpreted relative to the current JSP page. Accepts a request-time attribute value (which must evaluate to a `String` that is a relative URL specification).

Action `jsp:param`.

The `jsp:param` element is used to provide key/value information. This element is used in the `jsp:include`, `jsp:forward`, and `jsp:params` elements. A translation error shall occur if the element is used elsewhere.

When doing `jsp:include` or `jsp:forward`, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters, with NEW VALUES TAKING PRECEDENCE over existing values when applicable. The scope of the new parameters is the `jsp:include` or `jsp:forward` call; i.e. in the case of an `jsp:include` the new parameters (and values) will not apply after the include. This is the same behavior as in the `ServletRequest include` and `forward` methods.

For example, if the request has a parameter `A=foo` and a parameter `A=bar` is specified for forward, the forwarded request shall have `A=bar,foo`. Note that the NEW PARAM HAS PRECEDENCE.

Syntax:

```
<jsp:param name="name" value="value" />
```

This action has two mandatory attributes: `name` and `value`. `name` indicates the name of the parameter, and `value`, which may be a request-time expression, indicates its value.

Chapter 9. Building JSP Pages Using Tag Libraries

For a custom tag library or a library of Tag Files, create the '`taglib`' directive for a JSP page.

The set of significant tags a JSP container interprets can be extended through a tag library.

The `taglib` directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.

If a JSP container implementation cannot locate a tag library description, a fatal translation error shall result.

It is a fatal translation error for the `taglib` directive to appear after actions or functions using the prefix.

In the following example, a tag library is introduced and made available to this page using the `super` prefix; no other tag libraries should be introduced in this page using this prefix. In this particular case, we assume the tag library includes a `doMagic` element type, which is used within the page.

```
<%@ taglib uri="http://www.mycorp/supertags" prefix="super" %>

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

Syntax:

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

Table 9.1. `taglib` Directive attributes

Attribute	Description
uri	Either an absolute URI or a relative URI specification that uniquely identifies the tag library descriptor associated with this prefix. The URI is used to locate a description of the tag library.
tagdir	Indicates this prefix is to be used to identify tag extensions installed in the `/WEB-INF/tags/` directory or a subdirectory. An implicit tag library descriptor is used. A translation error must occur if the value does not start with `/WEB-INF/tags/`. A translation error must occur if the value does not point to a directory that exists. A translation error must occur if used in conjunction with the `uri` attribute.
prefix	Defines the prefix string in `<prefix:tagname>` that is used to distinguish a custom action, e.g `<myPrefix:myTag>`. Prefixes starting with `jsp:`, `jspx:`, `java:`, `javax:`, `servlet:`, `sun:`, and `sunw:` ARE RESERVED. A prefix must follow the naming convention specified in the XML namespaces specification. Empty prefixes are illegal in this version of the specification, and must result in a translation error.

A fatal translation-time error will result if the JSP page translator encounters a tag with name `prefix:Name` using a `prefix` that is introduced using the `taglib` directive, and `Name` is not recognized by the corresponding tag library.

Given a design goal, create the custom tag structure in a JSP page to support that goal.

```
<%@ taglib uri="mytags" prefix="codecamp"  %>

<ol>
        <!-- repeats N times. A null reps value means repeat once. -->
        <codecamp:repeat reps='<%=request.getParameter("repeats") %>'>
                <li><codecamp:prime length="40" />
        </codecamp:repeat>
```

```
</ol>
```

Given a design goal, use an appropriate JSP Standard Tag Library (JSTL v1.1) tag from the "core" tag library.

The center of JSTL is the `core` taglib. This can be split into five areas:

1. General purpose
2. Variables support
3. Conditional
4. Iterator
5. URL Related

To use the `core` library, use the following directive:

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

The `prefix` attribute specifies the prefix used in the tag name for a particular library. For example, the `core` library includes a tag named `out`. When combined with a prefix of `c`, the full tag would be `<c:out>`. You are free to use any prefix you like, but you must use different prefixes for each of the four standard tag libraries.

You must also put the corresponding `.tld` file for each tag library in your `/WEB-INF` directory and use the `taglib` element in your `web.xml` file to include the tag library:

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/core.tld</taglib-location>
</taglib>
```

General purpose tags.

The general-purpose tags let you display variable values, and enclose a group of tags within a try-catch block.

The `<c:out>` action provides a capability similar to JSP expressions such as `<%= scripting-language-expression %>` or `${el-expression}`. For example:

```
You have <c:out value="${sessionScope.user.itemCount}"/> items.
```

By default, `<c:out>` converts the characters <, >, ', ", & to their corresponding character entity codes (e.g. < is converted to &lt;). If these characters are not converted, the page may not be rendered properly by the browser, and it could also open the door for cross-site scripting attacks. The conversion may be bypassed by specifying `false` to the `escapeXml` attribute. The `<c:out>` action also supports the notion of DEFAULT values for cases where the value of an EL expression is `null`. In the example below, the value "unknown" will be displayed if the property city is not accessible.

```
<c:out value="${customer.address.city}" default="unknown"/>
```

The second option is to specify the default value as the content of the `<c:out>` tag:

```
<c:out value="${customer.address.street}">
    No address available
</c:out>
```

Syntax:

Without a body:

```
<c:out value="value" [escapeXml="{true|false}"]
        [default="defaultValue"] />
```

With a body (contains default value):

```
<c:out value="value" [escapeXml="{true|false}"]>
        default value
</c:out>
```

The `<c:catch>` action allows page authors to handle errors from any action in a uniform fashion, and allows for error handling for multiple actions at once. `<c:catch>` provides page authors with granular error handling: Actions that are of central importance to a page should not be encapsulated in a `<c:catch>`, so their exceptions will propagate to an error page, whereas actions with secondary importance to the page should be wrapped in a `<c:catch>`, so they never cause the error page mechanism to be invoked. The exception thrown is stored in the scoped variable identified by `var`, which always has `page` scope. If no exception occurred, the scoped variable identified by `var` is removed if it existed. If `var` is missing, the exception is simply caught and not saved.

Syntax:

```
<c:catch [var="varName"]>
        nested actions
</c:catch>
```

Variables support tags.

The action `<c:set>` is used to set the value of a JSP scoped attribute as follows:

```
<c:set var="foo" value="value"/>
```

In the following example, the `<c:set>` action sets the value of the `att1` scoped variable to the output of the `acme:foo` action. `<c:set>` – like all JSTL actions that create scoped attributes – creates scoped attributes in '`page`' scope by DEFAULT:

```
<c:set var="att1">
        <acme:foo>mumbojumbo</acme:foo>
</c:set>

<acme:atag att1="${att1}"/>
```

`<c:set>` may also be used to set the property of a JavaBeans object, or add or set a specific element in a `java.util.Map` object. For example:

```
<!-- set property in JavaBeans object -->
<c:set target="${cust.address}" property="city" value="${city}"/>

<!-- set/add element in Map object -->
<c:set target="${preferences}" property="color" value="${param.color}"/>
```

Syntax.

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

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

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

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

Syntax 3: Set a property of a target object (JavaBean object with setter property property, or a `java.util.Map` object) using attribute value:

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

Syntax 4: Set a property of a target object (JavaBean object with setter property property, or a `java.util.Map` object) using body content:

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

The `<c:remove>` action removes a scoped variable. If attribute scope is not specified, the scoped variable is removed according to the semantics of `PageContext.removeAttribute(varName)`. If attribute scope is specified, the scoped variable is removed according to the semantics of `PageContext.removeAttribute(varName, scope)`.

Syntax:

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

Conditional tags.

A simple conditional execution action evaluates its body content only if the test condition associated with it is true. In the following example, a special greeting is displayed only if this is a user's first visit to the site:

```
<c:if test="${user.visitCount == 1}">
        This is your first visit. Welcome to the site!
</c:if>
```

If the test condition evaluates to `true`, the JSP container processes the body content (JSP) and then writes it to the current `JspWriter`.

Syntax:

Syntax 1: Without body content:

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

Syntax 2: With body content:

```
<c:if test="testCondition"
        [var="varName"] [scope="{page|request|session|application}"]>
        body content (JSP)
</c:if>
```

The name of the exported scoped variable `var` for the resulting value of the test condition. The type of the scoped variable is `Boolean`.

The `<c:choose>` tag works like a Java `switch` statement in that it lets you choose between a number of alternatives. Where the `switch` statement has case statements, the `<c:choose>` tag has `<c:when>` tags. In a `switch` statement, you can specify a `default` clause to specify a default action in case none of the cases match. The `<c:choose>` equivalent of `default` is `<c:otherwise>` (optional), but note, it MUST be the LAST action nested within `<c:choose>`.

Syntax:

```
<c:choose>
        body content (<when> and <otherwise> subtags)
</c:choose>
```

```
<c:when test="testCondition">
        body content
</c:when>
```

```
<c:otherwise>
        conditional block
</c:otherwise>
```

Iterator tags.

The `<c:forEach>` action repeats its nested body content over the collection of objects specified by the items attribute. For example, the JSP code below creates an HTML table with one column that shows the default display value of each item in the collection:

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

A large number of collection types are supported by `<c:forEach>`, including all implementations of `java.util.Collection` (includes `List`, `LinkedList`, `ArrayList`, `Vector`, `Stack`, `Set`), and `java.util.Map` (includes `HashMap`, `Hashtable`, `Properties`, `Provider`, `Attributes`).

Arrays of objects as well as arrays of primitive types (e.g. `int`) are also supported. For arrays of primitive types, the current item for the iteration is automatically wrapped with its standard wrapper class (e.g. `Integer` for `int`, `Float` for `float`, etc.).

If the items attribute is of type `java.util.Map`, then the current item will be of type `java.util.Map.Entry`, which has the following two properties: `key` - the key under which this item is stored in the underlying `Map`; `value` - the value that corresponds to this key.

Syntax.

Syntax 1: Iterate over a collection of objects:

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

        body content (JSP)

</c:forEach>
```

Syntax 2: Iterate a fixed number of times:

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

        body content (JSP)

</c:forEach>
```

If specified, `begin` must be >= 0. If `end` is specified and it is less than `begin`, the loop is simply not executed. If specified, `step` must be >= 1. If `items` is null, it is treated as an `empty` collection, i.e., no iteration is performed. If `begin` is greater than OR EQUAL to the size of items, NO iteration is performed.

Examples:

```
<c:forEach var="i" start="1" end="10">
   Item <c:out value="${i}/><p>
</c:forEach>
```

```
<c:forEach var="emp" items="employees">
   Employee: <c:out value="${emp.name}"/>
</c:forEach>
```

`<c:forTokens>` iterates over tokens, separated by the supplied delimiters. The `items` attribute specifies the string to tokenize and the delimiters attribute specifies a list of delimiters (similar to the way `java.util.StringTokenizer` works).

Syntax:

```
<c:forTokens items="stringOfTokens" delims="delimiters"
        [var="varName"]
        [varStatus="varStatusName"]
        [begin="begin"] [end="end"] [step="step"]>

        body content

</c:forTokens>
```

For example:

```
<c:forTokens items="moe,larry,curly" delimiters="," var="stooge">
   <c:out value="${stooge}/><p>
</c:forTokens>
```

URL Related tags.

JSTL provides several tags for handling URLs and accessing Web resources. URLs can be difficult to work with when you must worry about URL rewriting (to insert the session ID when the browser doesn't support cookies), URL encoding of parameters, and referencing resources from a separate servlet context within the same servlet container.

The `<c:url>` tag formats a URL into a string and stores it into a variable. The `<c:url>` tag automatically performs URL rewriting when necessary. The `var` attribute specifies the variable that will contain the formatted URL. The optional `scope` attribute specifies the scope of the variable (`page` is the default). The `value` attribute specifies the URL to be formatted.

Syntax.

Syntax 1: Without body content:

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

Syntax 2: With body content to specify query string parameters:

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

        <c:param> subtags

</c:url>
```

As a security precaution, the URL is only rewritten for relative URLs.

```
<c:param name="name" value="value" />
```

```
<c:param name="name">
        parameter value
</c:param>
```

Examples:

```
<c:url var="trackURL" value="/tracking.html"/>
```

```
<c:url var="trackURL" value="/track.jsp" context="/tracking"/>
```

```
<c:url value="/track.jsp" var="trackingURL">
        <c:param name="trackingId" value="1234"/>
        <c:param name="reportType" value="summary"/>
</c:url>
```

The `<c:import>` tag is similar to the `<jsp:import>` tag, but it is much more powerful. For example, the `<jsp:import>` tag usually just imports resources from within the same servlet container. The `<jsp:import>` tag can import data from other servers as well as from within the same container. Also, the `<jsp:import>` tag automatically inserts the imported content directly into the JSP. Although the `<c:import>` tag can automatically insert content, it can also return the content as either a STRING or a READER. The only required attribute in the `<c:import>` tag is `url`, which specifies the URL to be imported. As with the `<c:url>` tag, the URL may be a relative URL, a context-relative URL, or an absolute URL.

Syntax.

Syntax 1: Resource content inlined or exported as a `String` object:

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

        optional body content for <c:param> subtags

</c:import>
```

Syntax 2: Resource content exported as a `Reader` object:

```
<c:import url="url" [context="context"]
        varReader="varReaderName"
        [charEncoding="charEncoding"]>

        body content where varReader is consumed by another action

</c:import>
```

Examples:

```
<c:import var="data" url="/data.xml"/>
<c:out value="${data}/>
```

is equivalent to:

```
<c:import url="/data.xml"/>
```

Using of reader:

```
<c:import url="/data.xml" varReader="dataReader" scope="session"/>
```

`<c:redirect>` sends an HTTP redirect to the client.

Syntax.

Syntax 1: Without body content:

```
<c:redirect url="value" [context="context"] />
```

Syntax 2: With body content to specify query string parameters:

```
<c:redirect url="value" [context="context"]>
        <c:param> subtags
</c:redirect>
```

Chapter 10. Building a Custom Tag Library

Describe the semantics of the "Classic" custom tag event model when each event method (doStartTag, doAfterBody, and doEndTag) is executed, and explain what the return value for each event method means; and write a tag handler class.

The classes and interfaces used to implement classic tag handlers are contained in the `javax.servlet.jsp.tagext` package. Classic tag handlers implement either the `Tag`, `IterationTag`, or `BodyTag` interface. Interfaces can be used to take an existing Java object and make it a tag handler. For newly created classic tag handlers, you can use the `TagSupport` and `BodyTagSupport` classes as base classes. These classes and interfaces are contained in the `javax.servlet.jsp.tagext` package.

Tag handler methods defined by the `Tag` and `BodyTag` interfaces are called by the JSP page's servlet at various points during the evaluation of the tag. When the start element of a custom tag is encountered, the JSP page's servlet calls methods to initialize the appropriate handler and then invokes the handler's `doStartTag` method. When the end element of a custom tag is encountered, the handler's `doEndTag` method is invoked for all but simple tags. Additional methods are invoked in between when a tag handler needs to manipulate the body of the tag.

Table 10.1. Tag Handler Methods

Tag Type	Interface	Methods
Basic	`Tag`	`doStartTag`, `doEndTag`
Attributes	`Tag`	`doStartTag`, `doEndTag`, `setAttribute1`, ..., `setAttributeN`, `release`
Body	`Tag`	`doStartTag`, `doEndTag`, `release`
Body, iterative evaluation	`IterationTag`	`doStartTag`, `doAfterBody`, `doEndTag`, `release`
Body, manipulation	`BodyTag`	`doStartTag`, `doInitBody`, `doAfterBody`, `doEndTag`, `release`

A tag handler has access to an API that allows it to communicate with the JSP page. The entry points to the API are two objects: the JSP context (`javax.servlet.jsp.JspContext`) for simple tag handlers and the page context (`javax.servlet.jsp.PageContext`) for classic tag handlers. `JspContext` provides access to implicit objects. `PageContext` extends `JspContext` with HTTP-specific behavior. A tag handler can retrieve all the other implicit objects (`request`, `session`, and `application`) accessible from a JSP page through these objects. In addition, implicit objects can have named attributes associated with them. Such attributes are accessed using `[set|get]Attribute` methods.

The `Tag` interface defines the basic protocol between a tag handler and a JSP page's servlet. It defines the life cycle and the methods to be invoked when the start and end tags are encountered.

The JSP page's servlet invokes the `setPageContext`, `setParent`, and attribute setting methods before calling `doStartTag`. The JSP page's servlet also guarantees that `release` will be invoked on the tag handler before the end of the page.

Here is a typical tag handler method invocation sequence:

```
ATag t = new ATag();
t.setPageContext(...);
t.setParent(...);
t.setAttribute1(value1);
t.setAttribute2(value2);
t.doStartTag();
t.doEndTag();
t.release();
```

The `BodyTag` interface extends `IterationTag` by defining additional methods that let a tag handler manipulate the content of evaluating its body:

- `setBodyContent` - Creates body content and adds to the tag handler

- `doInitBody` - Called before evaluation of the tag body. This method will not be invoked for empty tags or for non-empty tags whose `doStartTag()` method returns `SKIP_BODY` or `EVAL_BODY_INCLUDE`.

A typical invocation sequence is:

```
t.doStartTag();
out = pageContext.pushBody();
t.setBodyContent(out);
// perform any initialization needed after body content is set
t.doInitBody();
t.doAfterBody();
// while IterationTag.doAfterBody() returns EVAL_BODY_AGAIN we
// iterate body evaluation
...
t.doAfterBody();
t.doEndTag();
out = pageContext.popBody();
t.release();
```

If the tag handler does not need to manipulate the body, the tag handler should implement the `Tag` interface. If the tag handler implements the `Tag` interface and the body of the tag needs to be evaluated, the `doStartTag` method needs to return `Tag.EVAL_BODY_INCLUDE`; otherwise it should return `Tag.SKIP_BODY`.

If a tag handler needs to iteratively evaluate the body, it should implement the `IterationTag` interface. The tag handler should return `IterationTag.EVAL_BODY_AGAIN` from `IterationTag.doAfterBody` method if it determines that the body needs to be evaluated again.

If the tag handler needs to manipulate the body, the tag handler must implement `BodyTag` (or be derived from `BodyTagSupport`).

When a tag handler implements the `BodyTag` interface, it must implement the `doInitBody` and the `IterationTag.doAfterBody` methods. These methods manipulate body content passed to the tag handler by the JSP page's servlet.

Body content supports several methods to read and write its contents. A tag handler can use the body content's `getString` or `getReader` methods to extract information from the body, and the `writeOut`

(out) method to write the body contents to an out stream. The writer supplied to the writeOut method is obtained using the tag handler's getPreviousOut method. This method is used to ensure that a tag handler's results are available to an enclosing tag handler.

If the body of the tag needs to be evaluated, the doStartTag method needs to return BodyTag.EVAL_BODY_BUFFERED; otherwise, it should return Tag.SKIP_BODY.

The doInitBody method is called after the body content is set but before it is evaluated. You generally use this method to perform any initialization that depends on the body content.

The IterationTag.doAfterBody method is called AFTER the body content is evaluated. IterationTag.doAfterBody must return an indication of whether to continue evaluating the body. Thus, if the body should be evaluated again, as would be the case if you were implementing an iteration tag, IterationTag.doAfterBody should return IterationTag.EVAL_BODY_AGAIN; otherwise, IterationTag.doAfterBody should return Tag.SKIP_BODY.

The following example reads the content of the body (which contains a SQL query) and passes it to an object that executes the query. Since the body does not need to be reevaluated, IterationTag.doAfterBody returns Tag.SKIP_BODY:

```
public class QueryTag extends BodyTagSupport {
        public int doAfterBody() throws JspTagException {
                BodyContent bc = getBodyContent();
                // get the bc as string
                String query = bc.getString();
                // clean up
                bc.clearBody();
                try {
                        Statement stmt = connection.createStatement();
                        result = stmt.executeQuery(query);
                } catch (SQLException e) {
                        throw new JspTagException("QueryTag: " +
                                e.getMessage());
                }
                return SKIP_BODY;
        }
}
```

When you extend TagSupport, the doStartTag and doEndTag methods use the following return values (defined as constants):

- doStartTag returns one of the following values:
  - Tag.EVAL_BODY_INCLUDE - Allow body text (including JSP code) between the start and end tags. Note, however, that body text is not available to the doEndTag method.
  - Tag.SKIP_BODY - Ignore body text. Any text between the start and end tags is not evaluated or displayed.
- doEndTag returns one of the following values:
  - Tag.EVAL_PAGE - Continue evaluating the page.
  - Tag.SKIP_PAGE - Ignore the remainder of the page.

The following tag handler outputs messages from the doStartTag and doEndTag methods. These messages form the contents of the tag in the JSP page:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class SimpleTag extends TagSupport {

        /**
        * Executes when the tag is started.
        */
        public int doStartTag() throws JspException {
                try {
                        pageContext.getOut().print("Hello from doStartTag()");
                        // Allow text in the body of the tag.
                        return EVAL_BODY_INCLUDE;
                } catch(IOException ioe) {
                        throw new JspException(ioe.getMessage());
                }
        }

        /**
        * Executes with the end tag.
        */
        public int doEndTag() throws JspException {
                try {
                        pageContext.getOut().print("Hello from doEndTag()");
                        // Continue evaluating the page.
                        return EVAL_PAGE;
                } catch(IOException ioe) {
                        throw new JspException(ioe.getMessage());
                }
        }
}
```

If your custom tag must modify body content, extend the `BodyTagSupport` class. It implements `BodyTag`. Provides the `doInitBody` and `IterationTag.doAfterBody` methods. Extend this class when your tag handler must modify body content. The `doStartTag` method can return `BodyTag.EVAL_BODY_BUFFERED` in addition to `Tag.EVAL_BODY_INCLUDE` and `Tag.SKIP_BODY`.

The `BodyContent` object is a subclass of `JspWriter`. `JspWriter` is the writer used internally for the JSP `out` variable. The `BodyContent` object is available to `doInitBody`, `IterationTag.doAfterBody`, and `doEndTag` through the `bodyContent` variable. You can integrate this object's contents with the original `JspWriter` in the `doEndTag` method. The `BodyContent` object contains methods that you use to write output as well as methods to read, clear, and retrieve its contents. For example, you can use `bodyContent.getString()` to retrieve the writer's contents, optionally modifying the contents before integrating them with the original `JspWriter`.

The servlet container invokes the `doInitBody` method if the `doStartTag` method returns `BodyTag.EVAL_BODY_BUFFERED`. Use this method to initialize the body content, if necessary. Then the tag handler processes the body, and invokes the `IterationTag.doAfterBody` method.

The `IterationTag.doAfterBody` method returns `Tag.SKIP_BODY` or `IterationTag.EVAL_BODY_AGAIN`. If it returns `IterationTag.EVAL_BODY_AGAIN`, the servlet container loops back and re-executes the body. This lets you loop over repetitive data, such as enumerations and database result sets.

The following example shows using the `doInitBody` and `IterationTag.doAfterBody` methods. It also shows how to integrate `bodyContent` output with the output stream:

```
public class TestBody extends BodyTagSupport {
        public int doStartTag() throws JspException {
                try {
                        pageContext.getOut().print("doStartTag()");
                        return EVAL_BODY_BUFFERED;
                } catch(IOException ioe) {
                        throw new JspException(ioe.getMessage());
                }
        }

        public void doInitBody() throws JspException {
                try {
                        // Note, that this is a different writer than the one
                        // you have in doStartTag and doEndTag.
                        bodyContent.print("doInitBody()");
                } catch(IOException ioe) {
                        throw new JspException(ioe.getMessage());
                }
        }

        public int doAfterBody() throws JspException {
                try {
                        // Note, that this is a different writer than the one
                        // you have in doStartTag and doEndTag.
                        bodyContent.print("doAfterBody()");
                        // return IterationTag.EVAL_BODY_AGAIN;
                        // Use this to loop
                        return SKIP_BODY;
                } catch(IOException ioe) {
                        throw new JspException(ioe.getMessage());
                }
        }

        public int doEndTag() throws JspException {
                try {
                        // Write from bodyContent writer to original writer.
                        pageContext.getOut().print(bodyContent.getString());
                        // Now we're back to the original writer.
                        pageContext.getOut().print("doEndTag()");
                        return EVAL_PAGE;
                } catch(IOException ioe) {
                        throw new JspException(ioe.getMessage());
                }
        }
}
```

Using the `PageContext` API, write tag handler code to access the JSP implicit variables and access web application attributes.

A `PageContext` is an object that provides a context to store references to objects used by the page, encapsulates implementation-dependent features, and provides convenience methods. A JSP page implementation class can use a `PageContext` to run unmodified in any compliant JSP container while taking advantage of implementation-specific improvements like high performance `JspWriters`.

The PageContext provides a number of facilities to the page/component author and page implementor, including:

- a single API to manage the various scoped namespaces
- a number of convenience API's to access various public objects
- a mechanism to obtain the `JspWriter` for output
- a mechanism to manage session usage by the page
- a mechanism to expose page directive attributes to the scripting environment
- mechanisms to forward or include the current request to other active components in the application

- a mechanism to handle errorpage exception processing

```
public abstract class JspContext {

        public abstract void setAttribute(String name, Object value);
        public abstract void setAttribute(String name, Object value, int scope);
        public abstract Object getAttribute(String name);
        public abstract Object getAttribute(String name, int scope);
        public abstract Object findAttribute(String name);
        public abstract void removeAttribute(String name);
        public abstract void removeAttribute(String name, int scope);
        public abstract int getAttributesScope(String name);
        public abstract Enumeration getAttributeNamesInScope(int scope);
        public abstract JspWriter getOut();

}
```

```
public abstract class PageContext extends JspContext {

        public abstract javax.servlet.http.HttpSession getSession();
        public abstract java.lang.Object getPage();
        public abstract javax.servlet.ServletRequest getRequest();
        public abstract javax.servlet.ServletResponse getResponse();
        public abstract java.lang.Exception getException();
        public abstract javax.servlet.ServletConfig getServletConfig();
        public abstract javax.servlet.ServletContext getServletContext();

        public abstract void forward(java.lang.String relativeUrlPath)
                throws javax.servlet.ServletException, java.io.IOException;
        public abstract void include(java.lang.String relativeUrlPath)
                throws javax.servlet.ServletException, java.io.IOException;
        public abstract void handlePageException(java.lang.Exception e)
                throws javax.servlet.ServletException, java.io.IOException;

}
```

Given a scenario, write tag handler code to access the parent tag and an arbitrary tag ancestor.

An object created by the enclosing tag handler of a group of nested tags is available to all inner tag handlers. This form of object sharing has the advantage that it uses a private namespace for the objects, thus reducing the potential for naming conflicts. To access an object created by an enclosing tag, a tag handler must first obtain its enclosing tag with the static method `TagSupport.findAncestorWithClass(from, class)` or the `TagSupport.getParent()` method. The former method should be used when a specific nesting of tag handlers cannot be guaranteed. Once the ancestor has been retrieved, a tag handler can access any statically or dynamically created objects. Statically created objects are members of the parent. Private objects can also be created dynamically. Such objects can be stored in a tag handler with the `setValue` method and retrieved with the `getValue` method.

The following example illustrates a tag handler that supports both the named and private object approaches to sharing objects. In the example, the handler for a query tag checks whether an attribute named `connectionId` has been set. If the connection attribute has been set, the handler retrieves the connection object from the page context. Otherwise, the tag handler first retrieves the tag handler for the enclosing tag, and then retrieves the connection object from that handler:

```
public class QueryTag extends BodyTagSupport {
        public int doStartTag() throws JspException {
                String cid = getConnectionId();
                Connection connection;
                if (cid != null) {
                        // there is a connection id, use it
                        connection =(Connection)pageContext.getAttribute(cid);
                } else {
                        ConnectionTag ancestorTag =
                                (ConnectionTag) findAncestorWithClass(this,
                                ConnectionTag.class);
                        if (ancestorTag == null) {
                                throw new JspTagException("A query without
                                        a connection attribute must be nested
                                        within a connection tag.");
                        }
                        connection = ancestorTag.getConnection();
                        ...
                }
        }
}
```

The query tag implemented by this tag handler could be used in either of the following ways:

```
<tt:connection cid="con01" ... >
        ...
</tt:connection>

<tt:query id="balances" connectionId="con01">
        SELECT account, balance FROM acct_table
        WHERE customer_number = ?
        <tt:param value="${requestScope.custNumber}" />
</tt:query>

<tt:connection ... >
        <tt:query cid="balances">
                SELECT account, balance FROM acct_table
                WHERE customer_number = ?
                <tt:param value="${requestScope.custNumber}" />
        </tt:query>
</tt:connection>
```

The TLD for the tag handler indicates that the `connectionId` attribute is optional with the following declaration:

```
<tag>
        ...
        <attribute>
                <name>connectionId</name>
                <required>false</required>
        </attribute>
</tag>
```

Describe the semantics of the "Simple" custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.

JSP 2.0 introduces a new type of tag extension called a Simple Tag Extension. Simple Tag Extensions can be written in one of two ways:

- In Java, by defining a class that implements the `javax.servlet.jsp.tagext.SimpleTag` interface. This class is intended for use by advanced page authors and tag library developers who need the flexibility of the Java language in order to write their tag handlers. The `javax.servlet.jsp.tagext.SimpleTagSupport` class provides a default implementation for all methods in `SimpleTag`.

- In JSP, using tag files. This method can be used by page authors who do not know Java. It can also be used by advanced page authors or tag library developers who know Java but are producing tag libraries that are presentation -centric or can take advantage of existing tag libraries.

In addition to being simpler to work with, Simple Tag Extensions do not directly rely on any servlet APIs, which allows for potential future integration with other technologies. This is facilitated by the `JspContext` class, which `PageContext` now extends. `JspContext` provides generic services such as storing the `JspWriter` and keeping track of scoped attributes, whereas `PageContext` has functionality specific to serving JSPs in the context of servlets. Whereas the `Tag` interface relies on `PageContext`, `SimpleTag` only relies on `JspContext`. Unlike classic tag handlers, `SimpleTag` does not extend `Tag`.

```
package javax.servlet.jsp.tagext;

public interface SimpleTag extends JspTag {
        public void doTag()throws JspException, java.io.IOException;
        public void setParent(JspTag parent);
        public JspTag getParent();
        public void setJspContext(JspContext pc);
        public void setJspBody(JspFragment jspBody);
}
```

Most `SimpleTag` handlers should extend `javax.servlet.jsp.tagext.SimpleTagSupport`. This is the convenience class, similar to `TagSupport` or `BodyTagSupport`. There are also some helpful methods included in this class that include:

- `public JspFragment getJspBody()` - returns the body passed in by the container via `setJspBody`. The `JspFragment` encapsulates the body of the tag. If the `JspFragment` is `null`, it indicates that tag has a body content type of empty.

- `public static final JspTag findAncestorWithClass(JspTag from, java.lang.Class klass)` - finds the instance of a given class type that is closest to a given instance. This method uses the `getParent()` method from the `Tag` and/or `SimpleTag` interfaces. This method is used for coordination among cooperating tags. While traversing the ancestors, for every instance of `TagAdapter` (used to allow collaboration between classic `Tag` handlers and `SimpleTag` handlers) encountered, the tag handler returned by `TagAdapter.getAdaptee()` is compared to `klass`. In a case where the tag handler matches this class, and not its `TagAdapter`, is returned.

The body of a Simple Tag, if present, is translated into a JSP Fragment and passed to the `setJspBody` method. The tag can then execute the fragment as many times as needed. Because JSP fragments do not support scriptlets, the `<body-content>` of a `SimpleTag` cannot be "JSP". A TLD is invalid if it specifies "JSP" as the value for `<body-content>` for a tag whose handler implements the `SimpleTag` interface. JSP containers are recommended to but not required to produce an error if "JSP" is specified in this case.

During the translation phase, various pieces of the page are translated into implementations of the `javax.servlet.jsp.tagext.JspFragment` abstract class, before being passed to a tag handler.

JSP fragments are pieces of JSP code. Written using standard JSP syntax, a fragment is translated into an implementation of the interface `JspFragment` for use by tag handlers. A JSP fragment is used to represent the body of a tag for use with a `SimpleTag` handler. The `JspFragment` interface declares only two methods: `invoke` and `getJspContext`.

```
package javax.servlet.jsp.tagext;

public abstract class JspFragment {
        public abstract void invoke(java.io.Writer out)
                throws JspException, java.io.IOException;
        public abstract JspContext getJspContext();
}
```

`invoke` causes the fragment to be executed, writing the output produced to the `Writer` passed to it. You can invoke a fragment as many times as needed. `invoke` can throw a `SkipPageException`, which signals that the fragment has determined that the remainder of the page doesn't need to be evaluated.

The following example shows the simple custom tag:

```
public class MySimpleTag extends SimpleTagSupport {
        public void doTag()     throws JspException, IOException {
                getJspContext().getOut().write("Hello, World !");
        }
}
```

The lifecycle of a Simple Tag Handler is straightforward and is not complicated by caching semantics. Once a Simple Tag Handler is instantiated by the Container, it is executed and then discarded. The same instance must not be cached and reused.

The following lifecycle events take place for the simple tag handler (in the same order):

1. A new tag handler instance is created each time the tag is encountered by the container. This is done by calling the zero argument constructor on the corresponding implementation class. It is important to note that a new instance must be created for each tag invocation.

2. The `setJspContext()` and `setParent()` methods are invoked on the tag handler. If the value being passed is '`null`', then the `setParent()` method need not be called. In the case of tag files, a `JspContext` wrapper is created so that the tag file can appear to have its own page scope. Calling `getJspContext()` must return the wrapped `JspContext`.

3. The container calls the setters for each attribute defined for this tag in the order in which they appear in the JSP page or Tag File. If the attribute value is an expression language expression or a runtime expression, it gets evaluated first and is then passed on to the setter. On the other hand if the attribute is a dynamic-attribute then `setDynamicAttribute()` is called.

4. The `setJspBody()` method is called by the container to set the body of this tag, as a `JspFragment`. A value of `null` is passed to `setJspBody()` if the tag is declared to have a `<body-content>` of `empty`.

5. The `doTag()` method is called by the container. All tag logic, iteration and body evaluations occur in this method.

6. All variables are synchronized after the `doTag()` method returns.

The following example simply writes the body to the output stream. When the `Writer` given to invoke is `null`, the output from `invoke` goes to the `JspWriter` associated with the `JspContext` of the tag handler:

```

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class MySimpleTag extends SimpleTagSupport {
        public void doTag() throws JspException, IOException {
                getJspBody().invoke(null);
        }
}
```

The `invoke` method directs all output to a supplied writer or to the `JspWriter` returned by the `getOut` method of the `JspContext` associated with the tag handler if the writer is `null`.

```
<%@ taglib uri="/mytag" prefix="mytag" %>
<html>
        <body>
                <mytag:MySimpleTag>
                        Hello, World !
                </mytag:MySimpleTag>
        </body>
</html>
```

`SimpleTagSupport` provides the convenience method `getJspBody()` to return the `JspFragment` generated for the body content. The following example demonstrates that a `SimpleTag` can obtain a copy of the body so that it can use or MANIPULATE it:

```
public class MySimpleTag extends SimpleTagSupport {
        public void doTag() throws JspException, IOException  {
                StringWriter sw = new StringWriter();
                getJspBody().invoke(sw);
                getJspContex().getOut().write(sw.toString());
        }
}
```

is equivalent to:

```
public class MySimpleTag extends SimpleTagSupport {
        public void doTag() throws JspException, IOException  {
                StringWriter sw = new StringWriter();
                jspBody().invoke(sw);
                getJspContex().getOut().write(sw.toString());
        }
}
```

Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.

Writing a traditional custom tag requires two steps:

- Writing and compiling a tag handler.
- Defining the tag that is associated with the tag handler.

Tag files simplify the process. First, tag files don't need to be compiled. They are compiled as they are invoked. Also, tag files allow tag extensions to be written using only JSP syntax. This means someone who does not know Java can also write tag extensions. Secondly, a `tag` element in a tag library descriptor describes the name to be used in a JSP page to reference the custom action. Using tag files, the name of a custom action is the same as the tag file representing the action. Therefore, you don't need a tag library descriptor at all. A tag file looks like a JSP page. It can have directives, scripts, EL expressions, and standard and custom tags. A tag file has the `.tag` or `.tagx` extension and can also include other files that contain a common resource. An include file for a tag file has a `.tagf` extension.

To work, tag files must be placed in the `WEB-INF/tags` directory under your application directory OR a subdirectory under it. The container converts each tag file found in (or uder) `WEB-INF/tags` into a tag handler.

A number of implicit objects are available from inside of a tag file. You can access these objects from a script or an EL expression:

Table 10.2. The Tag Files implicit objects

Object	Type
`request`	`javax.servlet.http.HttpServletRequest`
`response`	`javax.servlet.http.HttpServletResponse`
`out`	`javax.servlet.jsp.JspWriter`
`session`	`javax.servlet.http.HttpSession`
`application`	`javax.servlet.ServletContext`
`config`	`javax.servlet.ServletConfig`
`jspContext`	`javax.servlet.jsp.JspContext`

This is the example of a tag library in which the tag file simply writes a `String` to the implicit `out` object:

```
<%–  WEB-INF/tags/myExample.tld –%>

<%
        out.println("Hello, World !");
%>
```

In JSP you need the `taglib` directive as usual, with the `prefix` attribute to identify your tag library throughout the page. NOTE, instead of a `uri` attribute, you have a `tagdir` attribute. The `tagdir` attribute refers to the `WEB-INF/tags` directory OR ANY subdirectory below `WEB-INF/tags`:

```

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

<simpleTag:myExample />
```

Combined with the expression language (EL), you can really build a scriptless JSP page very rapidly. The following example accepts an attribute from a calling JSP page and converts it to the upper case:

```

<%– WEB-INF/tags/myExample2.tld –%>

<%@ attribute name="someAttribute" %>
<%
        someAttribute = someAttribute.toUpperCase();
        out.println(someAttribute);
%>
```

The following JSP page that uses the tag file:

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

<simpleTag:myExample2 someAttribute="hello" />
```

example, which is Java-code-free:

```
<%— WEB-INF/tags/myExample3.tag —%>
<%@ variable name-given="myVar" scope="AT_BEGIN" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:set var="myVar" value="3"/>

After: ${myVar}

<jsp:doBody/>
```

To use the tag file, use the following JSP page:

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

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

<c:set var="myVar" value="1"/>

Before: ${myVar} <br>

<simpleTag:myExample3/>
```

Chapter 11. J2EE Patterns

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.

- Intercepting Filter.
- Model-View-Controller.
- Front Controller.
- Service Locator.
- Business Delegate.
- Transfer Object.

see: http://java.sun.com/blueprints/corej2eepatterns/

Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View-Controller, Front Controller, Service

Locator, Business Delegate, and Transfer Object.

- Intercepting Filter.
- Model-View-Controller.
- Front Controller.
- Service Locator.
- Business Delegate.
- Transfer Object.

see: http://java.sun.com/blueprints/corej2eepatterns/

Appendixes

Appendix A. First Appendix

First Section

sdsds

Second Section

sdsds

Third Section

sdsds

Bibliography

[Servlet_2.4] *Java Servlet Specification Version 2.4 (http://java.sun.com/products/servlet)*.

[JSP_2.0] *JavaServer Pages Specification Version 2.0 (http://java.sun.com/products/jsp)*.

[JavaRanch] *JavaRanch.com forum for SCWCD Certification*.

[HTTP-1.0] *Internet RFC: HTTP/1.0 - RFC 1945 (ftp://ftp.isi.edu/in-notes/rfc1945.txt)* .

[HTTP-1.1] *Internet RFC: HTTP/1.1 - RFC 2616 (ftp://ftp.isi.edu/in-notes/rfc2616.txt)* .