Java certification success, Part 2: SCWCD

Skill Level: Introductory

Seema Manivannan (seema@whizlabs.com)
Java developer and trainer
Whizlabs

04 May 2004

Sun Certified Web Component Developer (SCWCD) is one of most coveted certifications in the J2EE domain. If you're considering the SCWCD certification, you need to be aware that it takes more than just learning the servlet and JSP technologies. It requires in-depth knowledge of the topics specified in the exam objectives, and Java programmer and certification trainer, Seema Manivannan of Whizlabs offers just that in this comprehensive tutorial. Seema covers the 13 main objectives of the SCWCD exam and provides a Q&A section to ensure you understand the concepts.

Section 1. Getting started

Preparing for SCWCD

Sun Certified Web Component Developer (SCWCD) is one of most coveted certifications in the J2EE domain. If you're considering the SCWCD certification, you need to be aware that it takes more than just learning the servlet and JSP technologies. It requires in-depth knowledge of the topics specified in the exam objectives. It is not uncommon for even experienced programmers to perform poorly in the exam due to the fact that they might not be well-versed in everything that is covered by the objectives. For the best chance at success, it is important to follow a learning approach that is guided by the exam objectives.

Should I take this tutorial?

The SCWCD certification exam can be taken only by Sun Certified Programmers for Java 2 platform.

This tutorial is intended for professionals experienced in developing Web applications using the Java technology servlet and Java Server Pages (JSP) APIs. As it is not a comprehensive tutorial for these technologies, it is not recommended for novices in this field. The aim of this tutorial is to provide precise coverage of the concepts tested in the SCWCD exam. It focuses solely on what you need to know to be successful in the exam.

The SCWCD certification consists of 13 main objectives dealing with servlets as well as JSP pages, using JavaBeans components in JSP pages, developing and using custom tags, and dealing with some important J2EE design patterns. The objectives are:

- The servlet model
- Structure and deployment of modern servlet Web apps
- The servlet container model
- Developing servlets to handle server-side exceptions
- Developing servlets using session management
- Developing secure Web applications
- Developing thread-safe servlets
- The JavaServer Pages technology model
- Developing reusable Web components
- Developing JSP pages using JavaBeans components
- Developing JSP pages using custom tags
- Developing a custom tag library
- J2EE design patterns

Each chapter of the tutorial deals with a single objective. The code snippets provided as examples are easy to understand. This tutorial does not elaborate much on each topic; rather, it helps you prepare for the exam by concentrating on the key points.

Each chapter contains mock questions in the pattern of the SCWCD exam. These questions demonstrate the use of the ideas covered in that objective. Explanations about the correct and incorrect choices are included to give you a better understanding of the concepts.

Section 2. The servlet model

HTTP methods

The HTTP methods indicate the purpose of an HTTP request made by a client to a server. The four most common HTTP methods are `GET`, `POST`, `PUT`, and `HEAD`. Let's look at the features of these methods and how they are triggered.

`GET` method

The `GET` method is used to retrieve a resource (like an image or an HTML page) from the server, which is specified in the request URL. When the user types the request URL into the browser's location field or clicks on a hyperlink, the `GET` method is triggered. If a tag is used, the method attribute can be specified as " `GET` " to cause the browser to send a `GET` request. Even if no method attribute is specified, the browser uses the `GET` method by default.

We can pass request parameters by having a query string appended to the request URL, which is a set of name-value pairs separated by an "&" character. For instance:

```
http://www.testserver.com/myapp/testservlet?studname=Tom&studno=123
```

Here we have passed the parameters `studname` and `studno`, which have the values "Tom" and "123" respectively. Because the data passed using the `GET` method is visible inside the URL, it is not advisable to send sensitive information in this manner. The other restrictions for the `GET` method are that it can pass only text data and not more than 255 characters.

`POST` method

The purpose of the `POST` method is to "post" or send information to the server. It is possible to send an unlimited amount of data as part of a `POST` request, and the type of data can be binary or text.

This method is usually used for sending bulk data, such as uploading files or updating databases. The method attribute of the `<form>` tag can be specified as " `POST` " to cause the browser to send a `POST` request to the server.

Because the request parameters are sent as part of the request body, it is not visible as part of the request URL, which was also the case with the `GET` method.

`PUT` method

The `PUT` method adds a resource to the server and is mainly used for publishing pages. It is similar to a `POST` request, because both are directed at server-side resources. However, the difference is that the `POST` method causes a resource on the server to process the request, while the `PUT` method associates the request data with a URL on the server.

The method attribute of the `<form>` tag can be specified as " `PUT` " to cause the browser to send a `PUT` request to the server.

HEAD method
The HEAD method is used to retrieve the headers of a particular resource on the server. You would typically use HEAD for getting the last modified time or content type of a resource. It can save bandwidth because the meta-information about the resource is obtained without transferring the resource itself.

The method attribute of the `<form>` tag can be specified as " HEAD " to cause the browser to send a HEAD request to the server.

Request handling methods in HttpServlet

When an HTTP request from a client is delegated to a servlet, the `service()` method of the `HttpServlet` class is invoked. The `HttpServlet` class adds additional methods, which are automatically called by the `service()` method in the `HttpServlet` class to aid in processing HTTP-based requests. You can override these methods in your servlet class to provide the handling logic for each HTTP request.

The methods are listed in the following table:

Table 1. HTTP Methods

HTTP Method	Handler method in HttpServlet class
GET	doGet()
POST	doPost()
HEAD	doHead()
PUT	doPut()

The methods take `HttpServletRequest` and `HttpServletResponse` as the arguments. All of them throw `ServletException` and `IOException`.

Servlet lifecycle

The servlet lifecycle consists of a series of events, which define how the servlet is loaded and instantiated, initialized, how it handles requests from clients, and how is it taken out of service.

Loading and instantiation
For each servlet defined in the deployment descriptor of the Web application, the servlet container locates and loads a class of the type of the servlet. This can happen when the servlet engine itself is started, or later when a client request is actually delegated to the servlet. After that, it instantiates one or more object instances of the servlet class to service the client requests.

Initialization

After instantiation, the container initializes a servlet before it is ready to handle client requests. The container initializes the servlet by invoking its `init()` method, passing an object implementing the `ServletConfig` interface. In the `init()` method, the servlet can read configuration parameters from the deployment descriptor or perform any other one-time activities, so the `init()` method is invoked once and only once by the servlet container.

Request handling
After the servlet is initialized, the container may keep it ready for handling client requests. When client requests arrive, they are delegated to the servlet through the `service()` method, passing the request and response objects as parameters. In the case of HTTP requests, the request and response objects are implementations of `HttpServletRequest` and `HttpServletResponse` respectively. In the `HttpServlet` class, the `service()` method invokes a different handler method for each type of HTTP request, `doGet()` method for `GET` requests, `doPost()` method for `POST` requests, and so on.

Removal from service
A servlet container may decide to remove a servlet from service for various reasons, such as to conserve memory resources. To do this, the servlet container calls the `destroy()` method on the servlet. Once the `destroy()` method has been called, the servlet may not service any more client requests. Now the servlet instance is eligible for garbage collection.

Retrieving request parameters and headers
Request parameters are stored by the servlet container as a set of name-value pairs. The following methods of the `ServletRequest` interface are used to retrieve the parameters sent by a client:

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

The `getParameter()` method returns a single value of the named parameter. For parameters that have more than one value, the `getParameterValues()` method is used. The `getParameterNames()` method is useful when the parameter names are not known; it gives the names of all the parameters as an `Enumeration`.

Retrieving request headers
The HTTP request headers can be retrieved using the following methods of the `HttpServletRequest` interface:

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

For instance:

```
public void doPost(HttpServletRequest req,HttpServletResponse res){
            Enumeration headers=req.getHeaderNames();
            while(headers.hasMoreElements()) {
                  String header=(String)headers.nextElement();
                      System.out.println("Header is "+header);
                 }
            }
```

Retrieving initialization parameters

The initialization parameters of a servlet can be retrieved using the following methods of the `ServletConfig` interface.

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

Setting the response

The following methods of the `ServletResponse` interface can be used to set the response that is sent back to the client.

```
public void setContentType(String type);
```

This method sets the content type of the response that is sent to the client. The default value is "text/html."

```
public java.io.PrintWriter getWriter();
public javax.servlet.ServletOutputStream getOutputStream();
```

The `getWriter()` method returns a `PrintWriter` object that can send character text to the client. The `getOutputStream()` method returns a `ServletOutputStream` suitable for writing binary data in the response. Either of these methods can be used to write the response, but not both. If you call `getWriter()` after calling `getOutputStream()` or vice versa, an `IllegalStateException` will be thrown.

For instance:

```
public void doGet(HttpServletRequest req,HttpServletResponse res) {
    res.setContentType("text/html");
    PrintWriter pw=res.getWriter();
    pw.println("Hello World");
    pw.close();
}
```

Setting response headers

Response headers provide additional information to the browser about the response received. Header information is stored as name value pairs. The following methods in the `HttpServletResponse` interface are available to set header information.

```
public void setHeader(String name, String value);
public void setIntHeader(String name, int value);
public void setDateHeader(String name, long value);
```

Redirecting requests
It is possible to send a temporary redirect message to the browser, which directs it to another URL. The method is provided by the `HttpServletResponse` interface.

```
public void sendRedirect( String location );
```

This method can accept relative URLs; the servlet container will convert the relative URL to an absolute URL before sending the response to the client. If this method is called after the response is committed, `IllegalStateException` is thrown.

Using the RequestDispatcher interface

The `RequestDispatcher` interface provides methods to include or forward a request to another resource, which can be a servlet, HTML file, or JSP file. These methods are:

```
public void forward(ServletRequest request, ServletResponse response);
public void include(ServletRequest request, ServletResponse response);
```

The `forward()` method allows a servlet to do some processing of its own before the request is sent to another resource that generates the response. The `forward()` method should not be called after the response is committed, in which case it throws `IllegalStateException`.

The `include()` method enables a servlet to include the content of another resource into its own response. Unlike in the case of the `forward()` method, the included resource cannot change the response status code or set headers.

```
public RequestDispatcher getRequestDispatcher(String path);
```

Object attributes

A servlet can store data in three different scopes: request, session, and context.

Data is stored as key value pairs, where the key is a `String` object and the value is any object. These data objects are called attributes.

The attribute values persist as long as the scope is valid. The `ServletRequest`, `HttpSession()`, and `ServletContext()` methods provide the following methods to `get`, `set`, and `remove` attributes:

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

public void setAttribute(String name, Object object)

public void removeAttribute(String name)
```

The attributes set within the request scope can be shared with other resources by forwarding the request. However, the attributes are available only for the life of the request. A servlet can share session attributes with other resources that are serving a request for the same client session. The attributes are available only while the client is still active. The context scope is common for all the resources that are part of the same Web application, so the objects stored within a context can be shared between all these resources. These are available throughout the life of the Web application.

The servlet model summary

In this section, you learned the servlet methods invoked in response to the different HTTP requests like `GET`, `POST`, and `PUT`. Next, you walked through the servlet lifecycle methods and the purpose of each one of them. You examined the interfaces and methods used for operations like reading parameters, setting headers, and so on, and discovered how to use the `RequestDispatcher` interface to include or forward to a Web resource. Finally, you looked at the methods to get and set object attributes in request, session, and context scopes.

Sample questions 2

Question 1:

You need to create a database connection in your application after reading the username, password, and database server URL from the deployment descriptor. Which will be the best place to do this?

Choices:

- **A.** Servlet constructor
- **B.** `init()` method
- **C.** `service()` method

- **D.** `doGet()` method
- **E.** `doPost()` method

Correct choice:

- **B**

Explanation:

The `init()` method is invoked once and only once by the container, so the creation of the database connection will be done only once, which is appropriate. The `service()`, `doGet()`, and `doPost()` methods might be called many times by the container.

The username, password, and URL are to be read from the deployment descriptor. These initialization parameters are contained in the `ServletConfig` object, which is passed to the `init()` method. That is why we need to use the `init()` method instead of the constructor for this purpose, even though the constructor is also called only once.

Question 2:

A user can select multiple locations from a list box on an HTML form. Which of the following methods can be used to retrieve all the selected locations?

Choices:

- **A.** `getParameter()`
- **B.** `getParameters()`
- **C.** `getParameterValues()`
- **D.** `getParamValues()`
- **E.** None of the above

Correct choice:

- **C**

Explanation:

The `getParameterValues(String paraName)` method of the `ServletRequest` interface returns all the values associated with a parameter. It returns an array of Strings containing the values. The `getParameter()` method returns just one of the values associated with the given parameter, so choice A is incorrect. There are no methods named `getParameters()` or `getParamValues()`, so choices B and D are incorrect.

Section 3. Structure and deployment of modern servlet Web apps

Web application defined

A Web application is a collection of servlets, JSP pages, static pages, classes, and other resources that can be packaged in a standard way and run on multiple containers from multiple vendors.

Application structure

A Web application exists in a structured hierarchy of directories, which is defined by the Java Servlet Specification. The root directory of the Web application contains all the public resources, such as images, HTML pages, and so on, stored directly or within subfolders.

A special directory called WEB-INF exists, which contains any files that are not publicly accessible to clients.

The WEB-INF directory is organized as follows:

- The /WEB-INF/web.xml deployment descriptor.
- The /WEB-INF/classes/ directory for servlet and utility classes. The container makes these classes available to the Web application class loader.
- The /WEB-INF/lib/ directory for JAR files. These files contain servlets, beans, and other utility classes useful to the Web application. The container adds all the JAR files from this directory to the Web application class path.

Web Archive (WAR) files

Web application directories can be packaged and signed into a Web Archive (WAR) format (that is, a JAR file with .war extension instead of .jar) file using the standard Java Archive tools. When the container sees the extension .war, it recognizes that it is a Web application archive, decompresses the file, and deploys the application automatically.

A META-INF directory will be present in the WAR file, which contains information useful to Java Archive tools. This directory must not be publicly accessible, though its contents can be retrieved in the servlet code using the `getResource` and `getResourceAsStream` calls on the `ServletContext` interface.

Deployment descriptor

The deployment descriptor must be a valid XML file, named web.xml, and placed in the WEB-INF subdirectory of the Web application. This file stores the configuration information of the Web application. The order in which the configuration elements must appear is important and is specified by the deployment descriptor DTD, which is available from java.sun.com (see Resources).

The root element of the deployment descriptor is the `<web-app>` element; all other elements are contained within it.

Specifying the servlet details

Each servlet is defined using a `<servlet>` element; it contains child elements that provide details about the servlet.

<servlet-name>: The servlet's unique name within the Web application is specified by the `<servlet-name>` element. The clients can access the servlet by specifying this name in the URL. It is possible to configure the same servlet class under different names.

<servlet-class>: The fully qualified class name used by the servlet container to instantiate the servlet is specified by the `<servlet-class>` element.

<init-param>: Each initialization parameter for a servlet is specified using an `<init-param>` element. It has two child elements -- `<param-name>` and `<param-value>` -- which give the name and value of the parameter. The value of the initialization parameter can be retrieved in the servlet code using the `getInitParameter()` method of the `ServletConfig` interface.

The following code demonstrates the use of the `<servlet>` element within the deployment descriptor:

```
<servlet>
  <servlet-name> TestServlet </servlet-name>
  <servlet-class> com.whiz.TestServlet </servlet-class>
  <init-param>
    <param-name>country</param-name>
    <param-value>India</param-value>
  </init-param>
</servlet>
```

This code causes the servlet container to instantiate a servlet class `com.whiz.TestServlet` and associates it with the name `TestServlet`. It has one initialization parameter named "country," which has the value "India."

Servlet mappings

In some cases, it might be required to map different URL patterns to the same servlet. For this, we use the `<servlet-mapping>` element.

The `<servlet-mapping>` element has two sub-elements: `<servlet-name>` and `<url-pattern>`. The `<servlet-name>` sub-element must match with one of the servlet names declared in the deployment descriptor. The `<url-pattern>` sub-element is the URL string to be mapped with the servlet.

Using URL paths
When a client request arrives for a particular servlet, the Web application that has the longest context path matching with the start of the request URL is chosen first. Then the requested servlet is chosen by the container by comparing the remaining part of the request URI with the mapped URLs. The mapping rules are as follows (the first successful match is taken):

1. If there is an exact match of the path of the request to the path of a servlet, that servlet is chosen.
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. An extension is defined as the part of the last segment after the last "." character. If the last segment in the URL path contains an extension (for instance, .jsp), the servlet container will try to match a servlet that handles requests for the extension.
4. If none of the previous three rules results 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.

Consider the following sets of servlet mappings in the deployment descriptor:

```
<servlet-mapping>
   <servlet-name>servlet1</servlet-name>
   <url-pattern>/my/test/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
   <servlet-name>servlet2</servlet-name>
   <url-pattern>/another </url-pattern>
  </servlet-mapping>

<servlet-mapping>
   <servlet-name>servlet3</servlet-name>
   <url-pattern>*.tst </url-pattern>
  </servlet-mapping>
```

A string beginning with a "/" character and ending with a "/*" postfix is used for path mapping. If the request path is /my/test/index.html, then servlet1 is invoked to handle the request. Here the match occurs as was described in step 2 above.

If the request path is /another, then servlet2 services the request. Here the matching occurs as was describe in step 1 above. But when the path is /another/file1.tst, servlet3 is chosen. This is because the URL mapping for servlet2 requires an exact match, which is not available, so the extension mapping as described above in step 3 is chosen.

However, if the request path is /my/test/new.tst, the request would be handled by servlet1 and not by servlet3 because a match occurs in step 2 itself.

Web Archive (WAR) files summary

In this section, you learned the structural details of a servlet Web application, including the directories to place the deployment descriptor, class files, JAR files, and so on. You also saw how the servlet details like name, class, and initialization parameters are specified in the deployment descriptor. Finally, you learned about how to specify mappings between URL patterns and the servlets to be invoked.

Sample questions 3

Question 1:

Which of the following are not child elements of the `<servlet>` element in the deployment descriptor?

Choices:

- **A.** `<servlet-mapping>`
- **B.** `<error-page>`
- **C.** `<servlet-name>`
- **D.** `<servlet-class>`
- **E.** `<init-param>`

Correct choices:

- **A** and **B**

Explanation:

The `<servlet-mapping>` and `<error-page>` elements are sub-elements of the `<web-app>` element.

The `<servlet-name>` element defines the name for the servlet, and the `<servlet-class>` element specifies the Java class name that should be used to instantiate the servlet. The `<init-param>` element is used to pass initialization parameters to the servlet. The `<servlet-mapping>` element is used to specify which URL patterns should be handled by the servlet. The `<error-page>` element can be used to specify the error pages to be used for certain exceptions or error codes.

Question 2:

Which of the following requests will not be serviced by `MyServlet` (assume that the Web application name is test)?

```
<servlet-mapping>
   <servlet-name> MyServlet  </servlet-name>
   <url-pattern> /my/my/*   </url-pattern>
</servlet-mapping>
```

Choices:

- **A.** /test/my/my/my
- **B.** /test/my/my/a/b
- **C.** /test/my/my/a.jsp
- **D.** /test/my/a.jsp
- **E.** /test/my/my.jsp

Correct choices:

- **D** and **E**

Explanation:

To match a request URL with a servlet, the servlet container identifies the context path and then evaluates the remaining part of the request URL with the servlet mappings specified in the deployment descriptor. It tries to recursively match the longest path by stepping down the request URI path tree a directory at a time, using the "/" character as a path separator, and determining if there is a match with a servlet. If there is a match, the matching part of the request URL is the servlet path and the remaining part is the path info. In this case, when the servlet encounters any request with the path "/test/my/my," it maps that request to `MyServlet`. In choices A, B, and C, this path is present, hence they are serviced by `MyServlet`. Choices and D and E do not have this complete path, so they are not serviced.

Section 4. The servlet container model

Context

An object that implements the `javax.servlet.ServletContext` interface represents the environment in which a Web application is running. All the servlets belonging to the same Web application share the same context.

There is one instance of the `ServletContext` interface associated with each Web application deployed into a servlet container. If the container is distributed over multiple JVMs, a Web application will have an instance of the `ServletContext` for each VM.

Context initialization parameters

We can specify initialization parameters for a servlet context, so that application-wide information can be shared by all the servlets that belong to the same Web application. The servlets can retrieve these initialization parameters by invoking the following methods of the `ServletContext` interface:

```
public String getInitParameter(String name);
public Enumeration getInitParameterNames();
```

The `getInitParameter()` method returns a `String` containing the value of the named context-wide initialization parameter, or null if the parameter does not exist. The parameter name passed is case sensitive. The `getInitParameterNames()` method returns the names of the context's initialization parameters as an `Enumeration` of `String` objects. An empty `Enumeration` is returned if the context has no initialization parameters.

The servlet context is initialized when the Web application is loaded, and is contained in the `ServletConfig` object that is passed to the `init()` method. Servlets extending the `GenericServlet` class (directly or indirectly) can invoke the `getServletContext()` method to get the context reference, because `GenericServlet` implements the `ServletConfig` interface.

Declaring initialization parameters
Each servlet context initialization parameter of a Web application must be declared within a `<context-param>` element. The sub-elements are `<param-name>`, `<param-value>`, and `<description>` which is optional.

The following code specifies the name of the company as the context parameter:

```
<context-param>
        <param-name>CompanyName</param-name>
        <param-value> IBM </param-value>
        <description> Name of the company </description>
</context-param>
```

Note that `<context-param>` is a direct sub-element of the `<web-app>` root element.

We can access the value of the `CompanyName` parameter from the servlet code as follows:

```
String name=getServletContext().getInitParameter("CompanyName");
```

Application events and listeners

It might be necessary to take actions in response to certain events like the starting or stopping of a Web application. The following section discusses some listener interfaces that define methods, invoked in response to important events. To receive notifications, the listener class must be configured in the deployment descriptor.

ServletContextListener
Implementations of the `ServletContextListener` interface receive notifications about changes to the servlet context of the Web application of which they are part. The following methods are defined in the `ServletContextListener`:

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

The `contextInitialized()` method is invoked when the Web application is ready for service and the `contextDestroyed()` method is called when it is about to shut down. The following code shows how we can use these methods to log the application events:

```
public void contextInitialized(ServletContextEvent e) {
e.getServletContext().log("Context initialized");
}

public void contextDestroyed(ServletContextEvent e) {
e.getServletContext().log("Context destroyed");
}
```

ServletContextAttributeListener
The `ServletContextAttributeListener` interface can be implemented to

receive notifications of changes to the servlet context attribute list. The following methods are provided by this interface:

```
void attributeAdded(ServletContextAttributeEvent scab)
void attributeRemoved(ServletContextAttributeEvent scab)
void attributeReplaced(ServletContextAttributeEvent scab)
```

The `attributeAdded()` method is invoked by the container whenever a new attribute is added. When an existing attribute is removed or replaced, the `attributeRemoved()` and `attributeReplaced()` methods are invoked respectively.

HttpSessionAttributeListener
We can store attributes in the `HttpSession` object, which are valid until the session terminates. The `HttpSessionAttributeListener` interface can be implemented in order to get notifications of changes to the attribute lists of sessions within the Web application:

```
void attributeAdded(HttpSessionBindingEvent se)
void attributeRemoved(HttpSessionBindingEvent se)
void attributeReplaced(HttpSessionBindingEvent se)
```

The `attributeAdded()` method is invoked by the container whenever a new attribute is added to a session. When an existing attribute is removed from a session or replaced, the `attributeRemoved()` and `attributeReplaced()` methods are invoked respectively.

Configuring the listeners

The `<listener>` element in the deployment descriptor can be used to configure the listener implementation classes so that the servlet container can pass the events to the matching notification methods. There should be a `<listener>` element for each custom listener class implementing the `ServletContextListener`, `ServletContextAttributeListener`, or `SessionAttributeListener` interface. We do not need to specify which class implements that interface. This will be found out by the servlet container itself.

The `<listener>` element has only one `<listener>` sub-element whose value is specified as the fully qualified class name of the listener class as shown in the following code:

```
<listener>
  <listener-class>com.whiz.MyServletContextListener
  </listener-class>
</listener>
```

```
<listener>
  <listener-class>com.whiz.MyServletContextAttributeListener
  </listener-class>
</listener>
```

Distributed applications

A Web application can be marked distributable, by specifying the `<distributable>` element within the `<web-app>` element. Then the servlet container distributes the application across multiple JVMs. Scalability and failover support are some of the advantages of distributing applications.

In cases where the container is distributed over many VMs, a Web application will have an instance of the `ServletContext` for each VM. However, the default `ServletContext` is non-distributable and must only exist in one VM. As the context exists locally in the JVM (where created), the `ServletContext` object attributes are not shared between JVMs. Any information that needs to be shared has to be placed into a session, stored in a database, or set in an Enterprise JavaBeans component. However, servlet context initialization parameters are available in all JVMs, because these are specified in the deployment descriptor. `ServletContext` events are not propagated from one JVM to another.

All requests that are part of a session must be handled by one virtual machine at a time. `HttpSession` events, like context events, may also not be propagated between JVMs.

Also note that because the container may run in more than one JVM, the developer cannot depend on static variables for storing an application state.

The servlet container model summary

Under the third objective, you learned the methods to retrieve the initialization parameters of the servlet context. You also looked into the various events and listeners at application and session levels, and learned about configuring initialization parameters and listeners in the deployment descriptor. Finally, you reviewed the behavior of servlet context and session in a distributed environment.

Sample questions 4

Question 1:

Following is the deployment descriptor entry for a Web application using servlet context initialization parameters:

```
<web-app>
    ...
  <context-param>
          <param-name>Bill Gates</param-name>
          // xxx
  </context-param>
    ...
</web-app>
```

Which of the following elements to be placed at "// xxx" is valid?

Choices:

- **A.** `<param-size>`
- **B.** `<param-type>`
- **C.** `<param-value>`
- **D.** `<param-class>`

Correct choice:

- **C**

Explanation:

The `<context-param>` element contains the declaration of a Web application's servlet context initialization parameters. The `<param-name>` element contains the name of a parameter. Each parameter name must be unique in the Web application. The `<param-value>` element contains the value of a parameter.

Here is the DTD for the `<context-param>` element:

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

The `<param-size>`, `<param-type>`, and `<param-class>` elements are invalid elements. Thus choice C is correct.

Question 2:

Which of the following methods will be called when a `ServletContext` object is created?

- **A.** `ServletContextListener.contextInitialized()`
- **B.** `ServletContextListener.contextCreated()`
- **C.** `HttpServletContextListener.contextCreated()`
- **D.** `HttpServletContextListener.contextInitialized()`
- **E.** None of the above

Correct choice:

- **A**

Explanation:

The `ServletContext` interface defines a set of methods that a servlet uses to communicate with its servlet container. It is not Http-specific, so the interface is `ServletContext` rather than `HttpServletContext`. The context is initialized at the time that the Web application is loaded. The method called when the context is initialized is `contextInitialized(ServletContextEvent sce)`. The method called when the context is destroyed is `contextDestroyed(ServletContextEvent sce)`. The `ServletContextEvent` object that is passed in the `contextInitialized()` and `contextDestroyed()` methods can be used to retrieve a reference to the `ServletContext` object of the application.

Section 5. Developing servlets to handle server-side exceptions

Exception handling

When a Web application causes errors at the server side, the errors must be handled appropriately and a suitable response must be sent to the end user. In this section, you will discuss the programmatic and declarative exception handling techniques used to provide presentable error pages.

Exception handling in code

The `HttpServletResponse` interface provides the following methods that will send an appropriate error page to the client to indicate some error condition on the server side.

```
public void sendError(int statusCode);
public void sendError(int statusCode, String message);
```

The first version of the `sendError()` method sends an error response page, showing the given status code. The second version also displays a descriptive message.

The following code demonstrates the use of the `sendError()` method, handling

`FileNotFoundException`.

```
public void doGet(HttpServletRequest req, HttpServletResponse res) {
try {
        // code that throws FileNotFoundException
}
catch (FileNotFoundException e) {
  res.sendError(res.SC_NOT_FOUND);
}
```

The `setStatus()` method provided by the `HttpServletResponse` interface can be used to send an HTTP status code to the client. It is used mostly for non-error status codes such as `SC_OK` or `SC_MOVED_TEMPORARILY`.

```
public void setStatus(int statusCode);
```

These methods throw an `IllegalStateException` if the response is already committed.

RequestDispatcher

When an error occurs, you can use `RequestDispatcher` to forward a request to another resource to handle the error. The error attributes can be set in the request before it is dispatched to the error page, as shown below:

```
public void doGet(HttpServletRequest req, HttpServletResponse res){
try {
   // Code that throws exception
}
catch (Exception ex) {
request.setAttribute("javax.servlet.error.exception," ex.getMessage());
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher("error.jsp");
rd.forward(request,response);
 }
}
```

Throwing exceptions

The service methods in the `servlet` class declare only `ServletException` and `IOException` in their throws clauses, so we can throw only the subclasses of `ServletException`, `IOException`, or `RuntimeException` from these methods. All other exceptions should be wrapped as `ServletException` and the root cause of the exception set to the original exception before being propagated.

Declarative handling of exceptions

It is possible to configure error pages in the deployment descriptor, corresponding to particular error codes or exceptions. If a servlet sets a status code to indicate an error on the response, or if the servlet generates an exception that is unhandled, the servlet container looks up the error page mappings and invokes the associated resource.

The `<error-page>` element, which is a direct child of the `<web-app>` element, contains a mapping between an error code or exception type and the path of a resource in the Web application.

The following configuration maps the error code 404 to error.jsp and `SQLException` to `ErrorServlet`:

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

  <error-page>
            <exception-type>java.sql.SQLException</exception-type>
            <location>/error/ErrorServlet</location>
  </error-page>
```

If the exception generated is `ServletException`, the container generates the root cause exception wrapped in it, and then it looks for a matching error-page mapping.

Logging errors

It might be required to report errors and other debug information from the Web application for later analysis.

The logging methods provided by the `GenericServlet` class and `ServletContext` interface are:

```
public void log(String message)
public void log(String message, Throwable t)
```

The first version of the `log()` method writes the specified message, while the second version writes an explanatory message and a stack trace for a given `Throwable` exception to the servlet log file. Note that the name and type of the servlet log file is specific to the servlet container.

Developing servlets to handle server-side exceptions summary

In this section, you learned about handling exceptions and generating appropriate responses. You saw the `sendError()` and `sendStatus()` methods for programmatically handling exceptions. Next, you discovered the ways to use the declarative approach, by mapping exceptions and error codes to appropriate error pages, in the deployment descriptor. You also learned about using `RequestDispatcher` for forwarding a request to an error page. Finally, you reviewed the methods for logging the exception and related messages to the applications log file.

Sample questions 5

Question 1:

To which of the following classes or interfaces do the `sendError()` and `setStatus()` methods belong?

Choices:

- **A.** `HttpServletRequest`
- **B.** `HttpServletResponse`
- **C.** `ServletRequest`
- **D.** `ServletResponse`
- **E.** None of the above

Correct choice:

- **B**

Explanation:

The above methods belong to the `HttpServletResponse` interface. The `sendError()` methods (there are two overloaded methods) return an error message to the client according to the status code. The `setStatus()` methods are similar to the `sendError()` methods. They set the HTTP status code to be included in the response. Note that the above methods also belong to the `HttpServletResponseWrapper` class (in the `javax.servlet.http` package). This class provides a convenient implementation of the `HttpServletResponse` interface that can be subclassed by developers wishing to adapt the response from a servlet.

Question 2:

Which of the following statements is true regarding the following deployment descriptor definitions for an error-page element?

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

2. <web-app>
    ...
   <error-page>
         <exception-type>java.sun.com.MyException</exception-type>
         <location>/404.html</location>
   </error-page>
    ...
   <web-app>
```

Choices:

- **A.** Both of the above declarations are correct
- **B.** None of the above declarations is correct
- **C.** Only **1** is correct
- **D.** Only **2** is correct

Correct choice:

- **A**

Explanation:

The `<error-page>` element contains a mapping between an error code or exception type to the path of a resource in the Web application. Here is the DTD definition for the `<error-page>` element:

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

The error-code contains an HTTP error code, ex: 404. The exception type contains a fully qualified class name of a Java exception type. The location element contains the location of the resource in the Web application.

According to the above DTD definition, the `<error-page>` tag must contain either the error-code or exception-type and location. Thus both of the declarations in the question are true.

Section 6. Developing servlets using session

management

Maintaining sessions

HTTP, being a stateless protocol, has its own disadvantages. Each client request is treated as a separate transaction. In Web applications, it becomes necessary for the server to remember the client state across multiple requests. This is made possible by maintaining sessions for client server interactions. When a user first makes a request to a site, a new session object is created and a unique session ID is assigned to it. The session ID, which is then passed as part of every request, matches the user with the session object. Servlets can add attributes to session objects or read the stored attribute values.

Session tracking gives servlets the ability to maintain state and user information across multiple page requests. The servlet container uses the `HttpSession` interface to create a session between an HTTP client and the server.

To retrieve the session, we can use the `getSession()` method of the `HttpServletRequest` interface:

```
HttpSession getSession()
HttpSession getSession(boolean create)
```

Both the methods return the current session associated with this request. The first method creates a new session, if there is no existing session. The second version creates a new session only if there is no existing session and the boolean argument is true.

Storing and retrieving session objects

The `Session` object has methods for adding, retrieving, and removing Java objects:

```
public void setAttribute(String name, Object value);
public Object getAttribute(String name);
public Enumeration getAttributeNames();
public void removeAttribute(String name);
```

The following example shows how a session is retrieved from the current request and an `Integer` attribute is written into the session:

```
public void doGet(HttpServletRequest request, HttpServletResponse
    response)throws ServletException, IOException {
      response.setContentType("text/html");
      HttpSession httpSession = request.getSession(true);
      session.setAttribute("no", new Integer(2));
    }
```

Session events and listeners

In the third objective, you learned about the `HttpSessionAttributeListener` and its features. Here we'll discuss two more listener interfaces related to sessions -- `HttpSessionBindingListener` and `HttpSessionListener`.

HttpSessionBindingListener

An object implementing this interface is notified when it is bound to or unbound from a session. It is not set in the deployment descriptor of the application because the container checks the interfaces implemented by the attribute whenever it is added or removed.

The methods provided by the interface are:

```
    void valueBound(HttpSessionBindingEvent event);
    void valueUnbound(HttpSessionBindingEvent event);
```

It is important to note the difference between `HttpSessionAttributeListener` and `HttpSessionBindingListener`. `HttpSessionAttributeListener` is implemented by an object that is interested in receiving events from all the sessions belonging to the application, while `HttpSessionBindingListener` is implemented by the object attributes for the particular session to which they are added or removed.

HttpSessionListener

Implementations of `HttpSessionListener` are notified when a session is created or destroyed. The methods provided by the interface are:

```
    public void sessionCreated(HttpSessionEvent e);
    public void sessionDestroyed(HttpSessionEvent e);
```

The implementing class of this interface needs to be configured using the `<listener>` element in the deployment descriptor.

The `HttpSessionEvent`, which is passed to these methods, provides the following method that returns the associated session:

```
public HttpSession getSession();
```

Terminating a session

Sessions may get invalidated automatically due to a session timeout or can be explicitly ended. When a session terminates, the session object and the information stored in it are lost permanently.

The `HttpSession` interface provides the following method to terminate a session explicitly:

```
public void invalidate();
```

This method throws an `InvalidStateException` if invoked on a session, which has been invalidated. The container will unbind any objects bound to the session before it destroys the session. The `valueUnbound()` method will be invoked on all the session attributes that implement the `HttpSessionBindingListener` interface.

Session timeout

It is possible to use the deployment descriptor to set a time period for the session. If the client is inactive for this duration, the session is automatically invalidated.

The `<session-timeout>` element defines the default session timeout interval (in minutes) for all sessions created in the Web application. A negative value or zero value causes the session never to expire.

The following setting in the deployment descriptor causes the session timeout to be set to 10 minutes:

```
<session-config>
  <session-timeout>10</session-timeout>
</session-config>
```

You can also programmatically set a session timeout period. The following method provided by the `HttpSession` interface can be used for setting the timeout period (in seconds) for the current session:

```
public void setMaxInactiveInterval(int seconds)
```

If a negative value is passed to this method, the session will never time out.

URL rewriting

Sessions are made possible through the exchange of a unique token known as *session id*, between the requesting client and the server. If cookies are enabled in the client browser, the session ID will be included in the cookie sent with the HTTP request/response.

For browsers that do not support cookies, we use a technique called *URL rewriting* to enable session handling. If URL rewriting is used, then the session ID should be appended to the URLs, including hyperlinks that require access to the session and also the responses from the server.

Methods of `HttpServletResponse` that support URL rewriting are:

```
public String encodeURL(String url)
public String encodeRedirectURL(String url)
```

The `encodeURL()` method encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.

The `encodeRedirectURL()` method encodes the specified URL for use in the `sendRedirect()` method of `HttpServletResponse`. This method also returns the URL unchanged if encoding is not required.

URL rewriting must be consistently used to support clients that do not support or accept cookies to prevent loss of session information.

Developing servlets using session management summary

Under the fifth objective, you learned the methods to store and retrieve session objects. Next, you saw how the various session events and the corresponding listener interfaces. You also learned about the ways in which a session can be invalidated. Finally, you examined how URL rewriting is needed to enable session handling for browsers that do not offer cookie support.

Sample questions 6

Question 1:

Which of the following will ensure that the session never gets invalidated automatically?

Choices:

- **A.** Specify a value of 0 for `<session-timeout>`

- **B.** Call the `setMaxInactiveInterval()` method passing a value of 0
- **C.** Call the `setMaxInactiveInterval()` method passing a value of -1
- **D.** Call the `setSessionTimeOut()` method passing a value of 0
- **E.** Specify a value of -1 for `<session-timeout>`

Correct choices:

- **A**, **C**, and **E**

Explanation:

A <session-timeout> value of 0 or less means that the session will never expire, so choices A and E are correct. The <session-timeout> element is a sub-element of `session-config`. The `setMaxInactiveInterval()` method of `HttpSession` specifies the number of seconds between client requests before the servlet container will invalidate this session. A negative value (not 0) is required to ensure that the session never expires, so choice C is also correct.

Question 2:

How should you design a class whose objects need to be notified whenever they are added to or removed from the session?

Choices:

- **A.** Design the class implementing the `SessionBinding` interface
- **B.** Design the class implementing the `HttpSessionBindingListener` interface
- **C.** Design the class implementing the `HttpSessionListener` interface
- **D.** Design the class implementing the `HttpSessionAttributeListener` interface
- **E.** Configure an `HttpSessionAttributeListener` object in the deployment descriptor

Correct choice:

- **B**

Explanation:

The task can be accomplished by designing a class implementing `HttpSessionBindingListener` and then overriding two methods: `valueBound(HttpSessionBindingEvent e)` and `valueUnbound(HttpSessionBindingEvent e)`. The first method will be called whenever we add this object to a session. `valueUnbound()` is called whenever this object is removed from the session.

`HttpSessionAttributeListener` is used to design a general session attribute listener object, which always receives a notification whenever any type of attribute is added to, removed from, or replaced from a session, so choices D and E are incorrect.

Choice A is incorrect as there is no such interface. Choice C is incorrect because `HttpSessionListener` is used to receive when a session is created or destroyed.

Section 7. Developing secure Web applications

Security

Secure communication is essential to protect sensitive data, including personal information, passed to and from a Web application. Here you'll explore the important security concepts and configurations to overcome the security issues in servlet-based Web applications.

Security issues

Authentication is the means by which communicating entities verify their identities to each other. The username/password combination is usually used for authenticating the user.

Authorization is the means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. Even though an account holder can log into a banking system successfully, he is authorized to access only his own account.

Data integrity proves that information has not been modified by a third party while in transit. The correctness and originality is usually verified by signing the transmitted information. Auditing is the process of keeping a record or log of system activities, so as to monitor users and their actions in the network, such as who accessed certain resources, which users logged on and off from the system, and the like.

Malicious code is a piece of software that is deliberately designed to cause harm to computer systems. This kind of code attacks vulnerable systems by exploiting potential security holes. Viruses, worms, and trojans are examples of malicious code.

A Web site may be attacked to extract sensitive information, to simply crash the server, or for many other reasons. A denial-of-service attack is characterized by an explicit attempt by hackers to prevent genuine users of a service from accessing a

Web site by overloading the server with too many fake requests.

Authentication mechanisms

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

- HTTP basic authentication
- HTTP digest authentication
- Form-based authentication
- HTTPS client authentication

HTTP basic authentication
In basic authentication, a Web server requests a Web client to authenticate the user. The Web client obtains the username and the password from the user through a login box and transmits them to the Web server. The Web server then authenticates the user in the specified realm. Though it is quite easy to set up, it is not secure because simple base64 encoding is used. It is supported by all the common browsers.

HTTP digest authentication
The HTTP digest authentication also gets the username/password details in a manner similar to that of basic authentication. However, the authentication is performed by transmitting the password in an encrypted form. Only some Web browsers and containers support it.

Form-based authentication
Form-based authentication allows a developer to control the look and feel of the login screens. The login form must contain fields for entering a username and password. These fields must be named j_username and j_password, respectively.

Form-based authentication has the same lack of security as basic authentication because the user password is transmitted as plain text and the target server is not authenticated. However, it is quite easy to implement and is supported by most of the common browsers.

HTTPS client authentication
End-user authentication using HTTP over SSL (HTTPS) requires the user to possess a public key certificate (PKC). All the data is transmitted after incorporating public key encryption. It is the most secure authentication type and is supported by all the common browsers.

Configuring the authentication mechanism

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. It has three sub-elements: `<auth-method>`, `<realm-name>`, and `<form-login-config>`.

The `<auth-method>` element is used to configure the authentication mechanism for the Web application. As a prerequisite to gaining access to any Web resources that are protected by an authorization constraint, a user must have authenticated using the configured mechanism. Legal values for this element are "BASIC," "DIGEST," "FORM," or "CLIENT-CERT."

The `<realm-name>` element specifies the realm name to be used; this is required only in the case of HTTP basic authorization.

The `<form-login-config>` element specifies the login page URL and the error page URL to be used, if form-based authentication is used.

The following setting in the deployment descriptor defines basic authentication:

```
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>student</realm-name>
 </login-config>
```

Security constraints

A security constraint determines who is authorized to access the resources of a Web application.

security-constraint
The `<security-constraint>` element is used to associate security constraints with one or more Web resource collections. The sub-elements of `<security-constraint>` are `<display-name>` `<web-resource-collection>`, `<auth-constraint>`, and `<user-data-constraint>`.

web-resource-collection
The `<web-resource-collection>` element specifies a collection of resources to which this security constraint will be applied. Its sub-elements are `<web-resource-name>`, `<description>`, `<url-pattern>`, and `<http-method>` as described here:

- `<web-resource-name>` specifies the name of the resource.
- `<description>` provides a description for the resource.
- `<url-pattern>` specifies the URL patterns of the resource to be accessed.
- `<http-method>` specifies the HTTP methods to which this constraint will be applied.

If no HTTP methods are specified, the security constraint applies to all the HTTP methods.

The following configuration specifies that the `POST()` method of `MarksServlet` will be subject to the security constraints of the application:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name> marks </web-resource-name>
    <url-pattern> /servlet/MarksServlet </url-pattern>
    <http-method>POST</http-method>
  </web-resource-collection>
</security-constraint>
```

auth-constraint
The `<auth-constraint>` element specifies which security roles can access the resources to which the security constraint applies. Its sub-elements are `<description>` and `<role-name>`.

The `<role-name>` element should be a name defined in the `<security-role>` element in the deployment descriptor. Note that role names are case sensitive.

The following code indicates that users belonging to the role "teacher" would be given access to the resources that are protected by the security constraint:

```
<auth-constraint>
    <description>Only for teachers</description>
    <role-name>teacher</role-name>
<auth-constraint>
```

To specify that all roles can access the secure resources, specify the asterisk (*) character:

```
<auth-constraint>
    <description> For all roles </description>
    <role-name>*</role-name>
<auth-constraint>
```

user-data-constraint
The `<user-data-constraint>` element specifies how the data transmitted between the client and the server should be protected. Its sub-elements are `<description>` and `<transport-guarantee>`.

The `<transport-guarantee>` element can contain any of three values: `NONE`, `INTEGRAL`, or `CONFIDENTIAL`. Here, `NONE` means no transport guarantee, `INTEGRAL` means data cannot be changed in transit, and `CONFIDENTIAL` means the contents of a transmission cannot be observed.

The following example demonstrates the use of the `<user-data-constraint>` element:

```
<user-data-constraint>
        <description> Integral Transmission </description>
        <transport-guarantee>INTEGRAL</transport-guarantee>
</user-data-constraint>
```

Security issues summary

The sixth objective deals with security in Web applications and such important security concepts like authorization and authentication. You also learned about the various authentication mechanisms like basic and digest. Finally, you saw how to declare the security constraints and authentication mechanisms in the deployment descriptor.

Sample questions 7

Question 1:

Which of the following authentication types uses public-key encryption as a security mechanism for a Web application?

Choices:

- **A.** BASIC
- **B.** DIGEST
- **C.** FORM
- **D.** CLIENT-CERT
- **E.** None of the above

Correct choice:

- **D**

Explanation:

HTTP basic authentication, which is based on a username and password, is the authentication mechanism defined in the HTTP/1.0 specification. It is not a secure authentication mechanism; it sends user information in simple base64 encoding. Hence choice A is incorrect.

Like the basic authentication type, HTTP digest authentication authenticates a user based on a username and password. It is more secure; the user information is encrypted before it's sent to the server. Hence choice B is incorrect.

In form-based authentication, the developer creates custom logic/error screens, the display of which are managed by the Web server. Hence choice C is incorrect.

End-user authentication (CLIENT-CERT) using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to possess a public key certificate (PKC). Hence choice D is correct.

Question 2:

Which of the following values may `<transport-guarantee>` contain?

Choices:

- **A.** NONE
- **B.** AUTHORIZED
- **C.** INTEGRAL
- **D.** AUTHENTICATED
- **E.** CONFIDENTIAL

Correct choice:

- **A**, **C**, and **E**

Explanation:

Choice C implies that the Web application requires the data transmission to have data integrity, whereas choice E implies that the Web application requires the data transmission to have data confidentiality.

Choice A implies that the application does not need any such guarantee. Plain HTTP is used when the value is set to `NONE`. HTTPS is used when the value is set to `INTEGRAL` or `CONFIDENTIAL`.

Choices B and D are incorrect because there are no such values for the `<transport-guarantee>` element.

Section 8. Developing thread-safe servlets

Thread-safe servlets

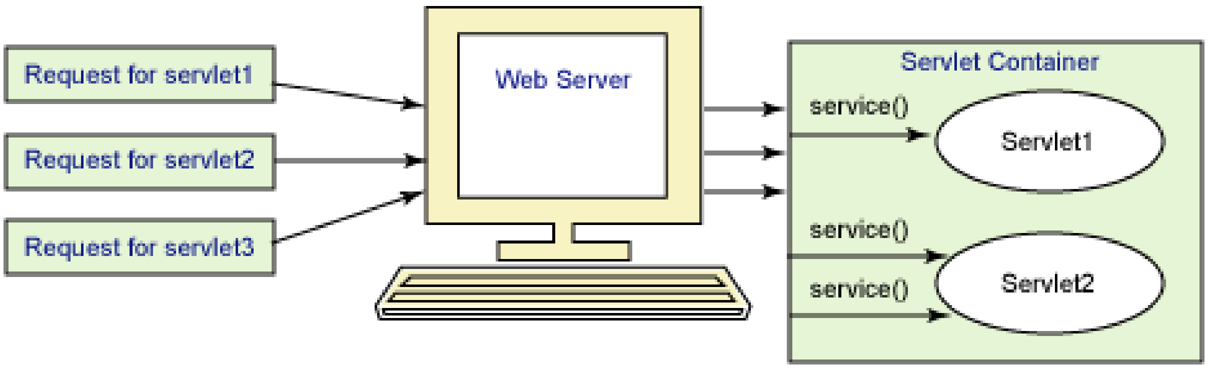
Typically, the servlet container loads only one instance of a servlet to process client requests.

A servlet instance may receive multiple requests simultaneously, and each time the `service()` method is executed in a different thread. In this section, we discuss what issues can arise when multiple threads execute servlet methods and how to develop thread-safe servlets.

Multi-threaded model

The multi-threaded model, which is used by default, causes the container to use only one instance per servlet declaration. By using a separate thread for each request, efficient processing of client requests is achieved.

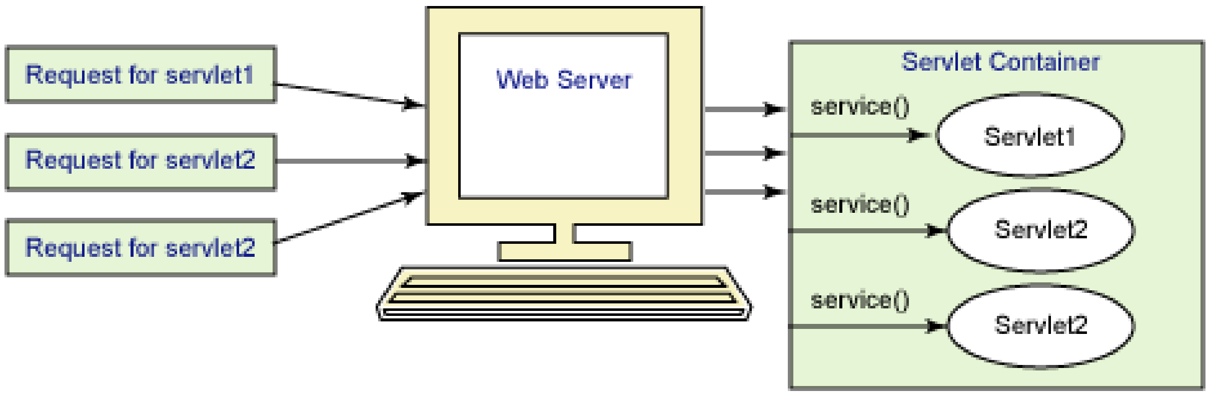
The figure below illustrates the multi-threaded model for servlets. One client request arrives for servlet1 and two for servlet2. The container spawns one thread for executing the `service()` method of servlet1 and two for the `service()` method of servlet2. All the threads execute simultaneously and the responses generated are returned to the clients.



SingleThreadModel interface

A very convenient way of ensuring that no two threads will execute concurrently in the servlet's `service()` method is to make the servlet implement the `SingleThreadModel` interface. The `SingleThreadModel` interface does not define any methods. The servlet container guarantees this by either synchronizing access to a single instance of the servlet or by maintaining a pool of servlet instances and dispatching each new request to a free servlet.

The figure below illustrates the situation when servlet2 implements the `SingleThreadModel` interface. Two client requests arrive for servlet2. Here the container uses a different instance of servlet2 to service each of the two requests.



However, this technique has its own disadvantages. If access to the servlet is synchronized, the requests get serviced one after the other, which can cause a severe performance bottleneck. Maintaining multiple servlet instances consumes time and memory.

Even though multiple threads cannot enter the `service()` method simultaneously, in this case thread safety issues are not completely taken care of. Static variables, attributes stored in session and context scopes, and so on are still being shared between multiple instances. Also, instance variables cannot be used to share data among multiple requests because the instances serving each request might be different.

Thread safety of variables and attributes

A servlet developer has to be aware of the effect of multiple threads on variables and attributes stored in different scopes.

Local variables
Local variables are always thread safe because each servlet has its own copy of these variables, so they cannot be used to share data between threads because their scope is limited to the method in which they are declared.

Instance variables
Instance variables are not thread safe in the case of the multi-threaded servlet model. In the case of servlets implementing `SingleThreadModel`, instance variables are accessed only by one thread at a time.

Static variables
Static variables are never thread safe. These variables are at class level, so they are shared between all instances. Hence these variables are not thread safe even if the servlet is implementing the `SingleThreadModel` interface. That is why they are usually used to store only constant/read-only data.

Context scope
The `ServletContext` object is shared by all the servlets of a Web application, so multiple threads can set and get attributes simultaneously from this object. Implementing the `SingleThreadModel` interface does not make any difference in

this case. Thus the context attributes are not thread safe.

Session scope
The `HttpSession` object is shared by multiple threads that service requests belonging to the same session, so the session attributes are also not thread safe. Just as the case with context attributes, the threading model has no impact on this behavior.

Request scope
The `ServletRequest` object is thread safe because it is accessible only locally within the `service()` method, so the request attributes are safe, irrespective of the threading model used.

Thread-safe servlets summary

In this section, you learned about the thread safety issues for servlets. First, you analyzed the implications of the multi-threaded servlet model, which is used by default, in the case of servlets. Then, you moved onto the significance of the single threaded model for servlets. You also identified the effect of multiple threads on variables and attributes under various scopes, which helps in developing thread safe Web applications.

Sample questions 8

Question 1:

Consider the following servlet code:

```
public class MyServlet extends HttpServlet
{
 final static int i=0;
 public void doGet(HttpServletRequest req, HttpServletResponse res)
 {
    private HttpSession session=req.getSession();
    private ServletContext ctx=getServletContext();
    synchronized(ctx)
       {
        Object obj=ctx.getAttribute();
        // code to alter obj
       }
 }
}
```

Which of the following variables in the above code are thread safe?

Choices:

- **A.** `i`

- **B.** `session`
- **C.** `ctx`
- **D.** `req`
- **E.** `obj`
- **F.** `res`

Correct choices:

- **A**, **C**, **D**, and **F**

Explanation:

The static variable `i` is thread safe because it is final (cannot be modified), or else it would not have been safe. Request and response objects are scoped only for the lifetime of the request, so they are also thread safe. Session and `ServletContext` objects can be accessed from multiple threads while processing multiple requests, so they are not thread safe. However, in this case, the `ServletContext` object is synchronized, so it can be accessed only by one thread at a time. `obj` is not thread safe because even though the `ServletContext` object is synchronized, its attributes are not. They need to be synchronized separately. Hence choices B and E are incorrect and choices A, C, D and F are correct.

Question 2:

Which of the following statements are true?

Choices:

- **A.** Multiple instances may be created on a servlet implementing `SingleThreadModel`
- **B.** No more than one instance is created for a servlet implementing `SingleThreadModel`
- **C.** Even static variables in a `SingleThreadModel` servlet are thread safe
- **D.** If no threading model is implemented, by default a servlet is executed in a multi-threaded model

Correct choices:

- **A** and **D**

Explanation:

When `SingleThreadModel` is implemented, the servlet container ensures that only one thread is executing the servlet's method at a time. So what will happen for multiple requests? In that case, the container may instantiate multiple instances of the servlet to handle multiple requests, so option A is correct and B is incorrect.

If the `SingleThreadModel` interface is not implemented, a servlet uses the multi-threaded model (that is, multiple threads can access the methods of the servlet). Static variables can be accessed through multiple instances of the same class, so they are not always thread safe. Hence choices B and C are incorrect and choices A and D are correct.

Section 9. The JavaServer pages technology model

JavaServer Pages

JavaServer Pages (JSP) technology is an extension of the Java Servlet API. JSP pages are typically comprised of static HTML/XML components, custom JSP tags, and Java code snippets known as scriptlets.

Even though JSP pages can contain business processing logic, they are mainly used for generating dynamic content in the presentation layer. Separation of business logic from presentation logic is one of the main advantages of this technology.

JSP tag types

JSP syntax can be classified into directives, declarations, scriptlets, expressions, standard actions, and comments.

Directives
A JSP directive provides information about the JSP page to the JSP engine. The types of directives are `page`, `include`, and `taglib` (a directive starts with a <%@ and ends with a %>):

- The `page` directive is used to define certain attributes of the JSP page:
  `<%@ page import="java.util.*, com.foo.*" %>`
- The `include` directive is used to include the contents of a file in the JSP page:
  `<%@ include file="/header.jsp" %>`
- The `taglib` directive allows us to use the custom tags in the JSP pages:
  `<%@ taglib uri="tlds/taglib.tld" prefix="mytag" %>`

Declarations

JSP declarations let you define variables and supporting methods that the rest of a JSP page may need.

To add a declaration, you must use the `<%!` and `%>` sequences to enclose your declarations, starting with a `<%!` and ending with a `%>`:

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

Here the variable `sum` is initialized only once when the JSP page is loaded.

Scriptlets

Scriptlets are fragments of code that are embedded within `<% ... %>` tags. They get executed whenever the JSP page is accessed:

```
<%
int count=0;
count++;
out.println("Count is "+count);
%>
```

Expressions

An expression is a Java expression that is evaluated when the JSP page is accessed and its value gets printed in the resultant HTML page. JSP expressions are within `<%= ... %>` tags and do not include semicolons:

```
<%=  count %>
```

The above expression prints out the value of the variable count.

Standard actions

JSP actions are instructions that control the behavior of the servlet engine. The six standard JSP actions are jsp:include, jsp:forward, jsp:useBean, jsp:setProperty, jsp:getProperty, and jsp:plugin. We will discuss actions in more detail in the following sections.

Comments

A JSP comment is of the form `<%-- Content to be commented --%>`. The body of the content is ignored completely.

JSP documents

JSP files can now use either JSP syntax or XML syntax within their source files. However, you cannot intermix JSP syntax and XML syntax in a source file.

JSP files using XML syntax are called JSP documents. All JSP documents have a `<jsp:root>` element within which all the other elements are enclosed:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" xmlns:prefix1="URI-for-taglib1"
        xmlns:prefix2="URI-for-taglib2" ...version="1.2">
    // contents of the JSP page
</jsp:root>
```

Let's view a sample JSP document:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
  <jsp:directive.page errorPage="error.jsp" />
  <jsp:directive.include file="test.jsp"/>
  <jsp:declaration> int count=10; </jsp:declaration>
  <jsp:text> Hello </jsp:text>
  <jsp:expression> count * 10 </jsp:expression>
  <jsp:scriptlet>
          int i=100;
          int j=11;
          out.println(i+j);
  </jsp:scriptlet>
</jsp:root>
```

You can see that the `<jsp:scriptlet>` tag is used for scriptlets, the `<jsp:expression>` tag is used for expressions, the `<jsp:declaration>` tag is used for declarations, and the `<jsp:text>` tag is used to embed text within a JSP document. The `page` directive is represented as `<jsp:directive.page>` and the `include` directive is represented as `<jsp:directive.include>`.

It is important to note that all the tags are case sensitive.

Page directive attributes

As discussed before, the page directives are used to define attributes that apply to the JSP page as a whole. These are passed onto the JSP container at translation time. Let's discuss the important page attributes that are relevant for the SCWCD exam.

import
The `import` attribute of a page directive is used to import a Java class into the JSP page. For instance:

```
<%@ page import="java.util.*, java.io.*,com.whiz.MyClass" %>
<%@ page import="com.whiz.TestClass" %>
```

It can appear multiple times in a translation unit.

session
The `session` attribute can have a value of true or false. It specifies whether the page should take part in an `HttpSession`. The default value is true. For instance:

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

errorPage
The `errorPage` attribute can be used to delegate the exception to another JSP page that has the error handling code. For instance:

```
<%@ page errorPage="error.jsp" %>
```

isErrorPage
The `isErrorPage` attribute specifies whether the current page can be the error handler for other JSP pages. The default value is false. For instance:

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

language
The `language` attribute specifies the language used by the JSP page; the default value is "java." For instance:

```
<%@ page language="java" %>
```

extends
The `extends` attribute specifies the superclass of the generated servlet class of the JSP page. The default value of this attribute is vendor-specific. For instance:

```
<%@ page extends="mypackage.MyServlet" %>
```

buffer
The `buffer` attribute gives the minimum size of the output buffer before the content is sent to the client. For instance:

```
<%@ page buffer="32kb" %>
```

autoFlush
The `autoFlush` attribute specifies whether the data in the buffer should be sent to the client as soon as the buffer is full. The default value is true. For instance:

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

JSP lifecycle

When a request is mapped to a JSP page for the first time, it translates the JSP page into a servlet class and compiles the class. It is this servlet that services the client requests.

A JSP page has seven phases in its lifecycle, as listed below in the sequence of occurrence:

- Translation
- Compilation
- Loading the class
- Instantiating the class
- `jspInit()` invocation
- `_jspService()` invocation
- `jspDestroy()` invocation

Translation
In this phase, the JSP page is read, parsed, and validated. If there are no errors, a Java file containing the servlet class is created.

Compilation
The Java file created in the translation phase is compiled into a class file. All the Java code is validated and syntax errors are reported in this phase.

Loading and instantiating
The servlet class is loaded into memory and instantiated, if the compilation is successful.

`jspInit()`
The `jspInit()` method is called only once in the life of the servlet. It is this method that we perform any initializations required for the servlet.

`_jspService`
The request and response objects are passed to this method when each client request is received for the JSP page. JSP scriptlets and expressions are processed and included in this method.

`jspDestroy()`
The `jspDestroy()` method is called when the servlet instance is taken out of service by the JSP engine. Any cleanup operation, such as releasing resources, can

be performed in this method. After this method is called, the servlet is unable to serve any client requests.

JSP implicit objects

The JSP container makes available nine implicit objects that can be used within scriptlets and expressions because they are defined in the `_jspService()` method of the generated servlet.

The nine implicit objects in the JSP API and their purpose are listed in the following table:

Table 2. Implicit objects

Object	Class	Purpose
`application`	`javax.servlet.ServletContext`	Refers to the Web application's environment in which the JSP is executed.
`config`	`javax.servlet.ServletConfig`	The initialization parameters given in the deployment descriptor can be retrieved from this object.
`exception`	`java.lang.Throwable`	Available for pages that set the page directive attribute `isErrorPage` to true. It can be used for exception handling.
`Out`	`javax.servlet.jsp.JspWriter`	Refers to the output stream of the JSP page.
`page`	`java.lang.Object`	Refers to the current instance of the servlet generated from the JSP page.
`pageContext`	`javax.servlet.jsp.PageContext`	Provides certain convenience methods and stores references to the implicit objects.
`request`	Subtype of `javax.servlet.ServletRequest`	Refers to the current request passed to the `_jspService()` method.
`response`	Subtype of	Refers to the

	`javax.servlet.ServletResponse`	Response sent to the client. It is also passed to the `_jspService()` method.

Conditional and iterative statements

For generating dynamic content based on conditions, we can use conditional statements, such as if/else blocks. For performing repetitive tasks, there are iterative statements using for or while loops. Conditional and iterative statements can span across multiple scriptlets, so that we can include HTML code in between.

For instance, the following scriptlet code uses a conditional statement to check whether a user's password is valid. If it is valid, the marks are printed using an iterative statement.

```
<% if(passwordValid)
         {
       %>
        Welcome, <%= username  %>
       <%
        for(int i=0; i<10; i++)
            {
       %>
          Printing <%=marks[i] %>
       <%
        }
       }

       %>
```

Be careful not to leave out the curly braces at the beginning and end of the Java fragments.

JavaServer Pages summary

In this section, you saw the basics of the JavaServer Pages (JSP) model. You learned about the various tag types and their purposes, and the various page directive attributes. Next, you identified the different phases in the JSP page lifecycle, followed by the nine implicit objects in the JSP API and the purpose of each of them. Finally, we saw how conditional and iteration statements can span across multiple scriptlets.

Sample questions 9

Question 1:

What will be the result of accessing the following JSP page, if the associated session does not have an attribute named `str`?

```
<%!
    String str;
    public void jspInit()
    {
     str = (String)session.getAttribute("str");
    }
%>
```

The string is: <%= `str` %>.

Choices:

- **A.** "null" is printed
- **B.** `NullPointerException` is thrown
- **C.** Code does not compile
- **D.** None of the above

Correct choice:

- **C**

Explanation:

The JSP engine declares and initializes nine objects in the `_jspService()` method. These implicit object variables are `application`, `session`, `request`, `response`, `out`, `page`, `pageContext`, `config`, and `exception`. Because they are declared locally to the `_jspService()` method, they are not accessible within the `jspInit()` method, which means this code will not compile. If this code was within the `jspService()` method, it would have compiled without errors and printed "null." Hence choices A, B, and D are incorrect, and choice C is correct.

Question 2:

What will be the result of an attempt to access this JSP page?

```
<% x=10;% >
<% int x=5;% >
<%! int x; %>
x= <%=x%>
x= <%=this.x%>
```

The string is <%= `str` %>.

Choices:

- **A.** Code does not compile because x is used before declaration
- **B.** Prints x=5 followed by x=10
- **C.** Prints x=10 followed by x=5
- **D.** Prints x=10 followed by x=0
- **E.** None of the above

Correct choice:

- **B**

Explanation:

This declaration will create an instance variable x and initialize it to 0. Then in the `service()` method, you modify it to 10. Then you declare a local variable named x and give it the value 5. When you print x, it prints the local version of value 5. When you say `this.x`, you refer to the instance variable x, which prints 10. Hence choices A, C, and D are incorrect, and choice B is correct.

Section 10. Developing reusable Web components

Reusing Web components

Reusing Web components reduces redundancy of code and makes your application more maintainable. There are two mechanisms to reuse content in a JSP page: the `include` directive and the `<jsp:include>` action. The `include` directive is for including the contents of a Web component statically, while the `<jsp:include>` action enables dynamic inclusion.

Using the include directive

If the inclusion of the component happens when the JSP page is translated into a servlet class, it is static inclusion. Changes made to the included file later will not affect the results of the main JSP page.

The JSP syntax for the include directive is:

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

The XML syntax is:

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

If the relative URL starts with "/", the path is relative to the JSP application's context. If the relative URL starts with a directory or file name, the path is relative to the JSP file.

The included file can be a JSP page, HTML file, XML document, or text file. If the included file is a JSP page, its JSP elements are translated and included (along with any other text) in the JSP page. Once the included file is translated and included, the translation process resumes with the next line of the including JSP page. For instance, the following JSP page includes the content of the file another.jsp:

```
<html>
     <head>
      <title>JSP Include directive</title>
     </head>
     <body>
    <%
      This content is statically included.<br />
      <%@ include file="another.jsp" %>
     </body>
</html>
```

The including and included pages can access variables and methods defined in the other page; they even share the implicit JSP objects. However, the file attribute of the include directive cannot be an expression. For instance, the following code is not allowed:

```
<% String url="date.html"; %>
<%@ include file="<%=url%>" %>
```

The file attribute cannot pass parameters to the included page, so the following code is illegal:

```
<%@ include file="new.jsp?name=ram" %>
```

The `include` directive is typically used to include banner content, date, copyright information, or any such content that you might want to reuse in multiple pages.

Using the <jsp:include> action

The `<jsp:include>` action allows you to include either a static or dynamic resource in a JSP page. If the resource is static, its content is included in the calling JSP page. If the resource is dynamic, the request is delegated to the included component. When the `<jsp:include>` action is finished, control returns to the including page and the JSP container continues processing the page.

Dynamically included pages do not share the variables and methods of the including page. The syntax for the `jsp:include` element is:

```
<jsp:include page="{relativeURL | <%= expression %>}" flush="true" />
```

The relative URL can be absolute or relative to the current JSP file. Here is an example, demonstrating the use of the `<jsp:include>` action:

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

Note that the value of the page attribute can be an expression that evaluates to a String, representing the relative URL, as shown here:

```
<% String url = "another.jsp" %>
<jsp:include page=">%=url%> " />
```

Because the `<jsp:include>` element handles both types of resources, you can use it when you don't know whether the resource is static or dynamic.

`<jsp:forward>` action
The mechanism for transferring control to another Web component from a JSP page is provided by the `jsp:forward` element. The forwarded component, which can be an HTML file, a JSP file, or a servlet, sends the reply to the client. The syntax is:

```
<jsp:forward page="{relativeURL | <%= expression %>}" />
```

For instance, the following code forwards the request to main.jsp:

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

The remaining portion of the forwarding JSP file, after the `<sp:forward>` element, is not processed. Note that if any data has already been sent to a client, the `<jsp:forward>` element will cause an `IllegalStateException`.

Passing parameters in dynamic inclusion
When an include or forward action takes place, the original request object is available to the target page. We can append parameters to the request object using the `<jsp:param>` element. The included component should be dynamic, such as a

JSP or a servlet that can process the passed request parameters. For instance, the request received by another.jsp has two additional parameters:

```
<jsp:include page="another.jsp">
        <jsp:param name="username" value="Tom" />
        <jsp:param name="ssn" value="<%=ssnString%>" />
</jsp:include>
```

As you can see in this example, the values passed can be static or dynamic.

Developing reusable Web components summary

In this section, which covered the ninth objective, you learned the ways to include the content of another Web component into a JSP page. You saw how to include the component statically -- that is, at translation time of the page -- and the ways to dynamically include or forward the content of components when the JSP page is requested.

Sample questions 10

Question 1:

Consider the following code segment:

```
<%! String filePath = "Helloworld.jsp"%>
<%@ include file="<%= filePath %>"%>
```

This will include the content of Helloworld.jsp within the current JSP file. Select the right choice.

Choices:

- **A.** True
- **B.** False

Correct choice:

- **B**

Explanation:

When you include a file using the `include` directive, the inclusion processing is done at translation time. But request-time attribute values are evaluated at request time, not translation time. Therefore, the attribute value of the file cannot be an expression, it must be a string literal value. Also remember that file attribute values cannot pass any parameters to the included page, so the following example is invalid:

```
<%@ include file="Helloworld.jsp?planet=Earth" %>
```

Question 2:

Which of the following can be used to include the file another.jsp in the file test.jsp, assuming that there are no errors?

File 1: test.jsp

```
<% String str="hello"; % >

// line 1

<%= str%>
 %>
```

File 2: another.jsp

```
<% str+="world"; %>
```

Choices:

- **A.** `<jsp:directive.include file="another.jsp"/>`
- **B.** `<%@ include page="another.jsp" %>`
- **C.** `<%@ include file="another.jsp" %>`
- **D.** `<jsp:include page="another.jsp"/>`
- **E.** `<jsp:include file="another.jsp"/>`

Correct choice:

- **C**

Explanation:

Here, another.jsp does not declare the variable `str`, so it cannot compile on its own. Note that when a JSP file is dynamically included, it is compiled separately, so the variables are not shared between the including files and the included one. In this case, dynamic inclusion is not possible, so choices C and D are incorrect (D also has an invalid attribute). Choice A is incorrect because XML syntax and JSP syntax cannot be used on the same page. Choice B is incorrect because the valid attribute for the `include` directive is `file` and not `page`.

Section 11. Developing JSP pages using JavaBeans components

JavaBeans components

JavaBeans components (or beans) are Java classes that are portable, reusable, and can be assembled into applications. JSP pages can contain processing and data access logic in the form of scriptlets. However, if there is a lot of business logic to be handled that way, it makes the JSP page cluttered and difficult to maintain. Instead, you can encapsulate the processing logic within beans and use them with JSP language elements.

Any Java class can be a bean, if it adheres to the following design rules:

- For each readable property of data type "proptype," the bean must have a method of the following signature:

```
public proptype getProperty() { }
```

- For each writable property of data type "proptype," the bean must have a method of the following signature:

```
public setProperty(proptype x) { }
```

In addition, the class must also define a constructor that takes no parameters. For instance, the following class encapsulates user information and exposes it using getter and setter methods.

```
public class User {
   private String name;
   private String password;
   public User() {
   }
   public void setName(String name) {
        this.name=name;
   }
   public String getName() {
        return name;
   }
   public void setPassword(String password) {
        this.password=password;
   }
   public String getPassword() {
        return password;
   }
}
```

Declaring the bean

The `<jsp:useBean>` action is used to declare that the JSP page will use a bean that is stored within and accessible from the specified scope.

For instance, the following tag declares a bean of type `UserBean` and of `id user`, in application scope:

```
    <jsp:useBean id="user" class="UserBean" scope="application"/>
```

The value of the `id` attribute is the identifier used to reference the bean in other JSP elements and scriptlets. The scope of the bean can be `application`, `session`, `request`, or `page`. The `id` attribute is mandatory, while `scope` is optional. The default value of `scope` is `page`.

The other possible attributes are `class`, `type`, and `beanName`. A subset of these attributes needs to be present in the `<jsp:useBean>` action in one of the following combinations:

- `class`
- `class` and `type`
- `type`
- `beanName` and `type`

Using class attribute
The following tag uses the `class` attribute. This causes the JSP engine to try and locate an instance of the `UserBean` class with the `id` user, in the application scope. If it is unable to find a matching instance, a new instance is created with the `id user`, and stored in the application scope.

```
    <jsp:useBean id="user" class="UserBean" scope="application"/>
```

Using class and type attributes
If `class` and `type` are both used as in the code below, the JSP engine tries to locate a bean instance of type `PersonBean`. However, if a matching bean could not be found, a new instance is created by instantiating the `UserBean` class. In this

case, `UserBean` has to be a class that is of type `PersonBean`.

```
<"jsp:useBean id="user" type="PersonBean" class="UserBean" scope="application"/>
```

Using type attribute
The `type` attribute can be used without `class` or `beanName` attributes as in the case below:

```
<jsp:useBean id="user" type="PersonBean" scope="session"/>
```

This will cause the JSP engine to look for a bean of the given type within the mentioned scope. In this case, if no existing bean matches the type, no new bean instance will be created and an `InstantiationException` is thrown.

Using type and beanName attributes
The `beanName` attribute can refer to a class name or the name of a serialized file containing the bean. When you use `beanName`, the bean is instantiated by the `java.beans.Beans.instantiate` method. If the `beanName` represents a serialized template, it reads the serialized form using a class loader. Otherwise, the bean is normally instantiated.

```
<"jsp:useBean id="user" type="PersonBean" beanName="User.ser" scope="session"/>
```

JavaBeans code in servlets

The JSP attribute scopes are `request`, `session`, and `application`. We have seen how to declare a bean within one of these scopes. As we already know, JSP code gets translated into a servlet and then compiled before execution.

Let's discuss the equivalent servlet code generated for beans declared in different scopes. In the servlet, objects of type `HttpServletRequest`, `HttpSession`, and `ServletContext` implement the `request`, `session`, and `application` scopes, respectively.

Consider the given bean declared within the `request` scope:

```
"jsp:useBean id="user" class="UserBean" scope="request"/>
```

Within the `service()` method, the equivalent servlet code would be as follows:

```
UserBean user=(UserBean)request.getAttribute("user");
If(user==null)
        {
        user=new UserBean();
        request.setAttribute("user",user);
        }
```

Now consider the code if the bean is declared in the `session` scope:

```
<jsp:useBean id="user" class="UserBean" scope="session" />
```

Here we need to obtain a reference to the current session by invoking the `getSession()` method of the `request` object. The generated servlet code would be as shown here:

```
HttpSession session=request.getSession();
UserBean user=(UserBean)session.getAttribute("user");
if(user==null)
   {
    user=new UserBean();
    session.setAttribute("user",user);
}
```

If the scope is application level, the bean is set as an attribute of the `ServletContext` object. The code generated would be:

```
ServletContext context=getServletContext();
UserBean user=(UserBean)context.getAttribute("user");
If(user==null)
        {
        user=new UserBean();
        context.setAttribute("user",user);
        }
```

Setting bean properties

We can set the property of a bean by using the `<jsp:setProperty>` action. It has four attributes: `name`, `property`, `value`, and `param`.

The `name` attribute refers to the `id` of the bean and the `property` attribute refers to the bean property that is to be set. These are mandatory attributes.

The `value` attribute specifies the value to be specified for the bean property. The `param` attribute can be the name of a request parameter whose value can be used to set the bean property. It is obvious that the `value` and `param` attributes would never be used together.

The following code sets the `name` property of `UserBean` to the value `Tom`:

```
<jsp:setProperty name="user" property="name" value="Tom" />
```

To set the value of the `name` property using the request parameter `username`, we do the following:

```
<jsp:setProperty name="user" property="name" param="username" />
```

Assume that the `request` parameter has the same name as the bean property that is to be set. In this case, the above code can be changed like this:

```
<jsp:setProperty name="user" property="name" />
```

Now let's see the code to set all the bean properties from the request parameter values:

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

If there is no matching request parameter for a particular property, the value of that property is not changed. This does not cause any errors.

Getting bean properties

To retrieve bean properties and print them to the output stream, we use the `<jsp:getProperty>` action. It has two attributes, `name` and `propertyname`, which are both mandatory.

The following code causes the value of the bean property `"name"` to be printed out:

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

Accessing JavaBeans components from JSP code

It is possible to access JavaBeans components from JSP code using the `id` attribute that is specified in the bean declaration by the `<jsp:useBean>` action.

For instance, in the following code we need to invoke a bean method, `isLoggedIn()`, to check if the user login was successful. For this, we refer to the bean in the scriptlet using its `id` attribute:

```
<jsp:useBean id="user" class="UserBean"  scope="session" />

<%
  if(user.isLoggedIn()) {
%>

<jsp:forward page="userhome.jsp" />

<% } else { %>

 <jsp:forward page="error.jsp" >

 <% } %>
```

Here, we forward the user to the home page if he is already logged in, and to the error page if he is not.

Developing JSP pages using JavaBeans components summary

In this section, you learned how to declare and use JavaBeans components in JSP pages. You reviewed the JSP actions for setting and getting bean properties. You saw the servlet code generated for JavaBeans components declared in different scopes: `request`, `session`, and `application`. Finally, you learned about accessing declared beans from JSP scriptlets.

Sample questions 11

Question 1:

A user fills out a form in a Web application. The information is then stored in a JavaBeans component, which is used by a JSP page. The first two lines of code for the JSP page are as follows:

```
<jsp:useBean id="userBean" class="myapp.UserBean" scope="request"/>
<jsp:setProperty name="useBean" //XXX  />
```

Which of the following should be placed in the position //XXX to parse all the form element values to the corresponding JavaBeans component property (assuming that the form input elements have the corresponding variables -- with the same name -- in the JavaBeans component)?

Choices:

- **A.** `param="*"`
- **B.** `param="All"`

- **C.** `property="*"`
- **D.** `property="All"`
- **E.** None of the above

Correct choice:

- **C**

Explanation:

The `jsp:setProperty` action is used in conjunction with the `jsp:useBean` action to set the value of bean properties used by the JSP page. The property of the JavaBeans component can also be set as follows:

```
<jsp:setProperty name="myBean" property="name" value="<%=expression %>" />
```

When developing beans for processing form data, you can follow a common design pattern by matching the names of the bean properties with the names of the form input elements. You also need to define the corresponding getter/setter methods for each property within the bean. The advantage in this is that you can now direct the JSP engine to parse all the incoming values from the HTML form elements that are part of the `request` object, then assign them to their corresponding bean properties with a single statement, like this:

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

This runtime magic is possible through a process called *introspection*, which lets a class expose its properties on request. The introspection is managed by the JSP engine and implemented through the Java reflection mechanism. This feature alone can be a lifesaver when processing complex forms containing a significant number of input elements.

If the names of your bean properties do not match those of the form's input elements, they can still be mapped explicitly to your property by naming the parameter as:

```
<jsp:setProperty name="user" property="address" param="parameterName" />
```

Hence choices A, B, D, and E are incorrect, and choice C is correct.

Question 2:

Which of the following uses of the `<jsp:useBean>` tag for a JSP page that uses the `java.sun.com.MyBean` JavaBeans component are correct?

Choices:

- **A.** `<jsp:useBean id = "java.sun.com.MyBean" scope="page" />`
- **B.** `<jsp:useBean id = "MyBean" class="java.sun.com.MyBean" />`
- **C.** `<jsp:useBean id = "MyBean" type = "java.lang.String" scope="page" />`
- **D.** `<jsp:useBean id = "MyBean" beanName="java.sun.com.MyBean" scope="page" />`
- **E.** `<jsp:useBean id = "MyBean" beanName="java.sun.com.MyBean" className="MyBean" type = "java.lang.String" scope="page" />`

Correct choices:

- **B** and **C**

Explanation:

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`.

The syntax of the `jsp:useBean` action is as follows:

```
<jsp:useBean id="name" scope="page|request|session|application" beandetails />
```

where `beandetails` can be one of:

```
class="className"
class="className" type="typeName"
beanName="beanName" type="typeName"
type="typeName"
```

The description of various attributes are:

- `id`: The case-sensitive name used to identify the object instance.
- `scope`: The scope within which the reference is available. The default value is page.
- `class`: The fully qualified (including package name) class name.

- `beanName`: The name of a Bean, as you would supply to the `instantiate()` method in the `java.beans.Beans` class. This attribute can also be a request time expression.
- `type`: This optional attribute specifies the type of class, and follows standard Java programming casting rules. The type must be a superclass, an interface, or the class itself. The default value is the same as the value of the class attribute.

From the rules above, we can say:

- Either class or type must be present. Thus choice A is incorrect, and choices B and C are correct.
- If present, `beanName` must be accompanied by `type`. Thus choice D is incorrect.
- Both `beanName` and `class` can't be present. Thus choice E is also incorrect.

Section 12. Developing JSP pages using custom tags

Custom tags

JSP technology uses XML-like tags to encapsulate the logic that dynamically generates the content for the page. Besides the standard JSP tags, it is possible for the JSP developer to create custom tags, which encapsulate complex scripting logic. Using custom tags instead of scriptlets promotes reusability, flexibility, and clarity of the JSP page.

Tag libraries

JSP custom tags are distributed in the form of tag libraries. A tag library defines a set of related custom tags and contains the tag handler objects. These handler objects are instances of classes that implement some special interfaces in the `javax.servlet.jsp.tagext` package. The JSP engine invokes the appropriate methods of these classes when it encounters custom tags in the page.

The tag library needs to be imported into the JSP page before its tags can be used.

Tag library directive
A tag library can be declared by including a `taglib` directive in the page before any custom tag is used:

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

The `uri` attribute refers to a URI that uniquely identifies the tag library descriptor (TLD) that describes the set of custom tags associated with the named tag prefix.

The prefix that precedes the custom tag name is given by the prefix attribute. You cannot use the tag prefixes `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, and `sunw`, as these are reserved by Sun Microsystems. You can use more than one taglib directive in a JSP page, but the prefix defined in each must be unique.

Tag library descriptor file names must have the extension `. tld` and are stored in the WEB-INF directory of the WAR or in a subdirectory of WEB-INF.

We'll now discuss the possible values of the `uri` attribute. The value of the `uri` attribute can be the absolute path to the TLD file as shown below:

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

You can specify a logical name, which is mapped to the actual path of the TLD file in the deployment descriptor as shown here:

- `<%@ taglib uri="/mylib" prefix="test" %>`

The `<taglib>` element in the deployment descriptor can be used to map the logical name to the absolute path of the TLD file. It has two elements: `<taglib-uri>` specifies the logical name, and `<taglib-location>` gives the TLD file path.

For instance, the following mapping can be used to map the short name /mylib to /WEB-INF/mylib.tld.

```
<taglib>
  <taglib-uri>/mylib</taglib-uri>
  <taglib-location> /WEB-INF/tld/mylib.tld </taglib-location>
</taglib>
```

It is possible to specify a logical name even without configuring in the deployment descriptor. The container reads the TLD files present in the packaged JAR files present in the /WEB-INF/lib directory. For each TLD file that contains information about its own URI, the JSP Engine automatically creates a mapping between the given URI and the actual location of the TLD file.

We can also give the path to a packaged JAR file as the value for the `uri` attribute. In this case, the JAR file must have the tag handler classes for all the tags in the library. The TLD file must be placed in the META-INF directory of the JAR file.

```
<%@ taglib uri="/WEB-INF/lib/mylib.jar" prefix="test" %>
```

The classes implementing the tag handlers can be stored in an unpacked form in the WEB-INF/classes subdirectory of the Web application. They can also be packaged into JAR files and stored in the WEB-INF/lib directory of the Web application.

Using custom tags

JSP custom tags are written using XML syntax. The syntax for using a custom tag is `<prefix:tagName>`, where the prefix is the value of the prefix attribute in the taglib directive, which declares the tag library, and `tagName` is the name specified for the tag in the corresponding TLD file. Let's see how the different tag types are used in the JSP page.

Empty tag
A custom tag with no body is called an *empty tag* and is expressed as follows:

```
<prefix:tag />
```

Tag with attributes
A custom tag can have attributes, which can be used to customize the functionality of the tag. The details of the attributes of a tag are specified in the TLD file. For instance, the tag named welcome prints the message "Welcome Tom," because we passed "Tom" as the value for the name attribute.

```
<test:welcome name="Tom" />
```

Tags with JSP code as body

A custom tag can contain JSP content in the form of static text, HTML, and JSP elements like scriptlets, between the start and end tag. Such a tag is called a *body tag*.

For instance, the following tag gets the username from the request parameter and prints an appropriate welcome message.

```
<test:welcomeyou>
<% String yourName=request.getParameter("name"); %>
  Hello <B> <%= yourName %> </B>
</test:welcomeyou>
```

For body tags with attributes, the processing of the body by the tag handler can be customized based on the value passed for the attribute:

```
<test:hello loopcount=3>
   Hello World !
</test:hello>
```

Here, the tag processes the body iteratively; the number of iterations is given by the value of the `loopcount` attribute.

Nested tags
A tag can be nested within a parent tag, as illustrated below:

```
<test:myOuterTag>
   <H1>This is the body of myOuterTag</H1>
  <test:repeater repeat=4>
     <B>Hello World! </B>
  </test:repeater>
</test:myOuterTag>
```

The nested JSP tag is first evaluated and the output becomes part of the evaluated body of the outer tag. It is important to note that the opening and closing tags of a nested tag and its parent tag must not overlap.

Developing JSP pages using custom tags summary

Custom tags are useful in reducing the amount of scriptlets in the JSP pages, thereby allowing better separation of business logic and presentation logic. In this section, you learned about tag libraries and the role of the taglib directive in informing the JSP engine about the custom tags used in the page. You also learned the different types of custom tags and how to use them in a JSP page.

Sample questions 12

Question 1:

Consider the following mapping in the web.xml file:

```
<taglib>
    <taglib-uri>/myTagLib</taglib-uri>
    <taglib-location>/location/myTagLib.tld</taglib-location>
</taglib>
```

How would you correctly specify the above tag library in your JSP page?

Choices:

- **A.** `<%@ taglib uri="/myTagLib" id="myLib" %>`
- **B.** `<%@ taglib uri="/myTagLib" prefix="myLib" %>`
- **C.** `<%@ taglib name="/myTagLib" prefix="myLib" %>`
- **D.** `<%@ taglib uri="/myTagLib" name="myLib" %>`

Correct choice:

- **B**

Explanation:

The `taglib` directive is used to declare a tag library in a JSP page. It has two attributes: `uri` and `prefix`. The value of `uri` is the same as the value of the `<taglib-uri>` element in the deployment descriptor, where it has been mapped to the location of the library's TLD file. If this mapping is not used, then the `uri` attribute must directly point to the TLD file using a root relative URI such as uri="/location/myTagLib.tld." The `prefix` attribute is used to identify the tags from this library, used in the JSP page. Hence choices A, C, and D are incorrect and choice B is correct.

Question 2:

Which of the following XML syntaxes would you use to import a tag library in a JSP document?

Choices:

- **A.** `<jsp:directive.taglib>`
- **B.** `<jsp:root>`
- **C.** `<jsp:taglib>`
- **D.** None of the above

Correct choice:

- **B**

Explanation:

In XML format, the tag library information is provided in the root element itself:

```
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:test="sample.tld"
    version="1.2">
    .....
</jsp:root>
```

The attribute value pair `xmlns:test="sample.tld"` tells the JSPEngine that the page uses custom tags of prefix myLib and the location of the tld file. Hence choices A, C, and D are incorrect and choice B is correct.

Section 13. Developing a custom tag library

Custom tag library

You've already seen how to declare a tag library in a JSP page and how to use the custom tags belonging to that library. In this section, you'll learn how to develop custom tag libraries. To develop a tag library, you need to declare the tags in the tag library descriptor (TLD) file and implement the tag handler classes.

Tag library descriptor

A tag library descriptor (TLD) is an XML file whose elements describe a particular tag library. All tag definitions must be nested inside the `<taglib>` element in the TLD.

The `<uri>` element uniquely identifies the tag library; its value can be specified for the `uri` attribute in the taglib directive for the library. The JSP engine implicitly creates a mapping between the `uri` and the actual location of the file:

```
<taglib>
  <tlib-version> 1.0  <tlib-version>
  <jsp-version>1.2  <jsp-version>
  <short-name>  test <short-name>
  <uri> http://www.whizlabs.com/testLib </uri>
  <tag>
      <name> welcome</name>
      <tag-class> whiz.MyTag</tag-class>
      <body-content> empty</body-content>
      <attribute>
        <name>uname</name>
        <required> true</required>
        <rtexprvalue> false</rtexprvalue>
      </attribute>
   </tag>
</taglib>
```

Defining tags
Each tag is defined by a `<tag>` element. The mandatory elements `<name>` and `<tag-class>` specify the unique tag name and the tag handler class, respectively.

If a tag accepts attributes, then the `<tag>` element should have one or more `<attribute>` sub-elements.

We can indicate that an attribute is mandatory by specifying true value for the `<required>` element. If a value is not supplied for the attribute when the tag is used, this causes an error in the JSP page.

The `<rtexprvalue>` element specifies whether the attribute can accept request-time expression values. The default is false.

```
<test:welcome uname="<%=request.getParameter("username") %>" />
```

The `<body-content>` element can have one of the following values: `empty`, `JSP`, or `tagdependent`. For tags without a body or empty tags, we specify the value for this element as "empty."

All the tag usage examples shown are valid for empty tags.

```
<test:mytag />
<test:mytag uname="Tom" />
<test:mytag></test:mytag>
```

For tags that can have valid JSP code (can be plain text, HTML, scripts, custom tags) in their body, we specify the value for `<body-content>` as "JSP."

The following code illustrates the use of a tag with JSP code in its body:

```
<test:hello loopcount=3>
  Hello World !
</test:hello>
```

When the `<body-content>` tag has the value "tagdependent," the body may contain non-JSP content like SQL statements. For instance:

```
<test:myList>
select name,age from users
</test:myList>
```

When the `<body-content>` tag has the value "tagdependent" or "JSP," the body of the tag may be empty.

Tag handler interfaces

The tag handlers must implement `Tag`, `BodyTag`, or `IterationTag` interfaces. These interfaces are contained in the `javax.servlet.jsp.tagext` package.

Tag handler methods defined by these interfaces are called by the JSP engine at various points during the evaluation of the tag.

Tag interface
The Tag interface defines the basic protocol between a tag handler and JSP container. It is the base interface for all tag handlers and declares the main lifecycle methods of the tag.

- The `setPageContext()` method is called first by the container. The `pageContext` implicit object of the JSP page is passed as the argument.
- The `setParent()` method is called next, which sets the parent of the tag handler.
- For each attribute of the custom tag, a setter method is invoked next.
- The `doStartTag()` method can perform initializations. It returns the value `EVAL_BODY_INCLUDE` or `SKIP_BODY`.
- The `doEndTag()` method can contain the cleanup code for the tag. It returns the value `EVAL_PAGE` or `SKIP_PAGE`.
- The `release()` method is called when the tag handler object is no longer required.

IterationTag interface
The `IterationTag` interface extends Tag by defining one additional method that controls the reevaluation of its body.

- `IterationTag` provides a new method: `doAfterBody()`.
- If `doStartTag()` returns `SKIP_BODY`, the body is skipped and the container calls `doEndTag()`.
- If `doStartTag()` returns `EVAL_BODY_INCLUDE`, the body of the tag is evaluated and included, and the container invokes `doAfterBody()`.
- The `doAfterBody()` method is invoked after every body evaluation to control whether the body will be reevaluated.
- If `doAfterBody()` returns `IterationTag.EVAL_BODY_AGAIN`, then the body will be reevaluated. If `doAfterBody()` returns `Tag.SKIP_BODY`, then the body will be skipped and `doEndTag()` will be evaluated instead.

BodyTag interface
The `BodyTag` interface extends `IterationTag` by defining additional methods that let a tag handler manipulate the content of evaluating its body:

- The `doStartTag()` method can return `SKIP_BODY`, `EVAL_BODY_INCLUDE`, or `EVAL_BODY_BUFFERED`.
- If `EVAL_BODY_INCLUDE` or `SKIP_BODY` is returned, then evaluation happens as in `IterationTag`.
- If `EVAL_BODY_BUFFERED` is returned, `setBodyContent()` is invoked, `doInitBody()` is invoked, the body is evaluated, `doAfterBody()` is invoked, and then, after zero or more iterations, `doEndTag()` is invoked. The `doAfterBody()` element returns `EVAL_BODY_AGAIN` or `EVAL_BODY_BUFFERED` to continue evaluating the page and `SKIP_BODY` to stop the iteration.

Accessing the implicit objects from tag handlers

The tag handler classes can access all the objects that are available to the JSP page, in which the corresponding custom tags are being used. The `PageContext` object that is passed to the `setPageContext()` method by the JSP engine, represents the `pageContext` implicit object.

The `PageContext` class has the following methods to access the three JSP implicit objects: `request()`, `session()`, and `application()`.

Table 2. Methods of PageContext to access JSP implicit objects

Implicit object	Method Name	Return Type
`Request`	`getRequest()`	`ServletRequest`
`Session`	`getSession()`	`HttpSession`
`Application`	`getServletContext()`	`ServletContext`

Developing a custom tag library summary

In this section, you learned how to develop custom tag libraries. First, we walked through the tag library descriptor and identified the various elements of a TLD file. Next you learned about the Tag extension API for writing custom tag handlers. Finally, you examined the important lifecycle methods of the three interfaces: `Tag`, `IterativeTag`, and `BodyTag`.

Sample questions 13

Question 1:

Which of the following methods can return the `SKIP_PAGE` constant?

Choices:

- **A.** `doStartTag()`
- **B.** `doAfterBody()`
- **C.** `doEndTag()`
- **D.** `release()`

Correct choice:

- **C**

Explanation:

Depending on the return value of the `doStartTag()` method, the container calls the `doEndTag()` method. `doEndTag()` decides whether to continue evaluating the rest of the JSP page or not. It returns one of the two constants defined in the `Tag` interface: `EVAL_PAGE` or `SKIP_PAGE`.

A return value of `Tag.EVAL_PAGE` indicates that the rest of the JSP page must be evaluated and the output must be included in the response. A return value of `Tag.SKIP_PAGE` indicates that the rest of the JSP page must not be evaluated at all and that the JSP engine should return immediately from the current `_jspService()` method.

Question 2:

Which of the following statements is not true?

Choices:

- **A.** The container invokes the `release()` method on a tag handler object when it is no longer required
- **B.** The `setPageContext()` method is the first method that is called in a custom tag lifecycle
- **C.** The `doAfterBody()` is the only method defined by the `IterationTag` interface
- **D.** The `setBodyContent()` method is called only if the `doStartTag()` method returns `EVAL_BODY_INCLUDE` or `EVAL_BODY_BUFFERED`

Correct choice:

- **D**

Explanation:

The `setBodyContent()` method is called and the `bodyContent` object is set only if `doStartTag()` returns `EVAL_BODY_BUFFERED`. The container may reuse a tag instance if a custom tag occurs multiple times in a JSP page. The container calls the `release()` method only when the tag is to be permanently removed from the pool. This method can be used to release the tag handler's resources. The `setPageContext()` method is the first method called in the lifecycle of a custom tag. The JSP container calls this method to pass the `pageContext` implicit object of the JSP page in which the tag appears. The `doAfterBody()` method is the only method defined by the `IterationTag` interface. It gives the tag handler a chance to reevaluate its body.

Section 14. J2EE design patterns

J2EE design patterns

Design patterns are abstractions of solutions to commonly experienced design problems in software development. J2EE design patterns concentrate on the various issues encountered in the architecture and implementation of enterprise applications.

In this section, you'll learn about the five important J2EE design patterns covered in the SCWCD exam.

Value Objects

In an Enterprise JavaBeans (EJB) application, each invocation on a session bean or an entity bean is usually a remote method invocation across the network layer. Such invocations on the enterprise beans create an overhead on the network. If the server receives multiple calls to retrieve or update single attribute values from numerous clients, system performance would be degraded significantly.

A *Value Object* is a serializable Java object that can be used to retrieve a group of related data using just one remote method invocation. After the enterprise bean returns the Value Object, it is locally available to the client for future access.

If a client wishes to update the attributes, it can do it on the local copy of the Value Object and then send the updated object to the server. However, update requests from multiple clients can corrupt the data.

The Value Object is also known as a *Transfer Object* or *Replicate Object*.

Model-view-controller

Consider an application that needs to support multiple client types like WAP clients, browser-based clients, and so on. If we use a single controlling component to interact with the user, manage business processing, and manage the database, it affects the flexibility of the system. Whenever support for a new type of view needs to be added, the whole application will need to be redesigned. Also the business logic will need to be replicated for each client type.

As a solution to this problem, the model-view-controller (MVC) architecture divides applications into three layers -- model, view, and controller -- and decouples their respective responsibilities.

The model represents business data and operations that manage the business data. The model notifies views when it changes and provides the ability for the view to query the model about its state. Typically, entity beans would play the role of model in the case of enterprise applications.

The view handles the display styles and user interactions with the system. It updates data presentation formats when the model changes. A view also forwards user input to a controller. In J2EE applications, the view layer would include JSP and servlets.

A controller dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a standalone application, user inputs include text inputs and button clicks. In a Web application, users communicate by sending HTTP requests to the Web tier. Session beans or servlets would represent the controller layer.

Business Delegate

In a J2EE application, the client code needs to utilize the services provided by the business components. If the presentation tier components are made to access the business tier directly, there are some disadvantages. Whenever the business services API changes, all the client components would need to be altered accordingly. Also, the client code needs to be aware of the location of the business services.

The Business Delegate object helps to minimize coupling between clients and the business tier. This object encapsulates access to a business service, thereby hiding the implementation details of the service, such as lookup and access mechanisms. If the interfaces of the business service changes, only the Business Delegate object needs to be modified and the client components are not affected.

Using the Business Delegate can free the client from the complexities of handling remote method calls. For instance, this object can translate network exceptions into user-friendly application exceptions.

The Business Delegate may cache business service results. This improves performance by reducing the number of remote calls across the network. The Business Delegate object is also called *client-side facade* or *proxy*.

Front Controller

In the presentation layer of a Web application, multiple user requests need to be handled and forwarded to the appropriate resource for processing. The navigation steps vary according to the user actions. Also, the resources need to ensure that the user has been authenticated and is authorized to access the particular resource.

If the responsibility to control the navigation, authentication, and other processing is left to the views, it can give rise to certain problems. Each view component needs to maintain information about the next or previous component in the navigational sequence, which causes unnecessary dependency between components. Whenever there is a modification in the processing logic, changes will need to be made in many view components. Security code for authentication and authorization get mixed up with the presentation code.

Front Controller is a controlling component that holds the common processing logic that occurs within the presentation tier. It handles client requests and manages security, state management, error handling, and navigation. The Front Controller centralizes control logic that might otherwise be duplicated, and dispatches the requests to appropriate worker components.

As a component that provides the initial single point of entry for all client requests, it is also known as Front Component. Multiple Front Controllers can be designed for different business use cases, which together manage the workflow of a Web application.

Data Access Object

Most Web applications use a persistent storage mechanism to store data. The data access methods may differ for different types of data sources, which might range from relational databases to legacy systems. Even within an RDBMS environment, the actual syntax and format of the SQL statements may vary depending on the particular database product. Also, there might be applications that use more than one data source.

The coupling between the business tier and the database tier can cause difficulties in migrating the application from one data source to another. When this happens, all the business components that access the data source need to be altered accordingly. To overcome these dependencies, the business tier can interact with data sources through a Data Access Object (DAO).

The DAO implements the access mechanism required to work with the data source. The business component that relies on the DAO uses the simpler and uniform interface exposed by the DAO for its clients. By acting as an adapter between the component and the data source, the DAO enables isolation of the business components from the data source type, data access method, and connectivity details. Thus the data access logic is uniform and centralized, and database dependencies are minimized by the use of this pattern.

J2EE design patterns summary

This final section covered some of the most important design patterns used for Web applications: Value Objects, Model-View-Controller, Business Delegate, Front Controller, and Data Access Objects. You learned the important characteristics of each pattern and problems that are resolved by them.

Sample questions 14

Question 1:

Your Web application that handles all aspects of credit card transactions requires a component that would receive the requests and dispatch them to appropriate JSP pages. It should manage the workflow and coordinate sequential processing. Centralized control of use cases is preferred. Which design pattern would be best suited to address these concerns?

Choices:

- **A.** MVC
- **B.** Business Delegate
- **C.** Front Component
- **D.** Value Object
- **E.** Facade

Correct choice:

- **C**

Explanation:

Front Component or Front Controller is the design pattern best suited to handle the given requirements. The Front Controller is a component that provides a common point of entry for all client requests. It dispatches the requests to appropriate JSP pages and controls sequential processing. The control of use cases is centralized and a change in the sequence of steps affects only the Front Controller Component. The requirements only specify that workflow should be controlled, so MVC is not the right choice. (If asked about controlling and presenting the data in multiple views, however, MVC should be chosen.) Hence choices A, B, D, and E are incorrect and choice C is correct.

Question 2:

Consider a Web application where the client tier needs to exchange data with enterprise beans. All access to an enterprise bean is performed through remote interfaces to the bean. Every call to an enterprise bean is potentially a remote method call with network overhead.

In a normal scenario, to read every attribute value of an enterprise bean, the client would make a remote method call. The number of calls made by the client to the enterprise bean impacts network performance.

Which of the following design patterns is most suited to solve the above problem?

Choices:

- **A.** Data Access Object
- **B.** Model View Controller
- **C.** Value Object
- **D.** Business Delegate

Correct choice:

- **C**

Explanation:

In the scenario explained above, a single method call is used to send and retrieve the Value Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Value Object, populate it with its attribute values, and pass it by value to the client.

When an enterprise bean uses a Value Object, the client makes a single remote method invocation to the enterprise bean to request the Value Object instead of numerous remote method calls to get individual bean attribute values. Hence choices A, B, and D are incorrect and choice C is correct.

Section 15. Wrap-up

Summing up the tutorial

In this tutorial, we covered a wide range of topics, as defined by the objectives of the SCWCD exam. Real work experience in Java-based Web technologies needs to be combined with a systematic learning pattern based on the test objectives to perform well in the exam. Applying and experimenting with new concepts can reinforce what you learn and in turn build your confidence. The sample exam questions given at the end of each chapter in this tutorial provide insight into what you can expect in the actual exam.

I hope this tutorial has been beneficial in your preparation for the SCWCD exam, and I wish you the best of luck on your exam.

Resources

Learn

- Take " Java certification success, Part 1: SCJP " by Pradeep Chopra (*developerWorks*, November 2003).
- Here you can find the DTD for the Servlet 2.3 deployment descriptor.
- You can also refer to the JSP Documentation.
- Here are some useful JSP tutorials from Sun:
  - JSP Tutorial
  - JSP Short Course
  - JSP Java Beans Tutorial
- You can explore the following tutorials on servlets:
  - Servlet Tutorial
  - Fundamentals of Java Servlets
- Learn how custom tags are developed and used. Check out the following links:
  - Tag Libraries Tutorial
  - Jakarta Taglibs Tutorial
- These SCWCD certification guides will help you focus on the exam topics:
  - *SCWCD Certification Study Kit* (Manning Publications, July 2002) by Hanumant Deshmukh and Jignesh Malavia
  - *Professional SCWCD Certification* (Wrox Press, November 2002) by Daniel Jepp and Sam Dalton
- Read more books on servlet and JSP technologies.
  - *Java Servlet Programming* (O'Reilly, April 2001) by Jason Hunter with William Crawford
  - *Professional JSP* (Wrox Press, April 2001) by Simon Brown, Robert Burdick, Jayson Falkner, Ben Galbraith, Rod Johnson, Larry Kim, Casey Kochmer, Thor Kristmundsson, Sing Li, Dan Malks, Mark Nelson, Grant Palmer, Bob Sullivan, Geoff Taylor, John Timney, Sameer Tyagi, Geert Van Damme, and Steve Wilkinson.
- Take a look at the J2EE design patterns.
- Check out this comprehensive article on SCWCD certification.
- You can practice and assess your knowledge using the following:

- Whizlabs SCWCD Exam Simulator
- Java Ranch SCWCD Mock Exam
- Sun JSP Quiz

Get products and technologies

- You can download the Servlet 2.3 Specifications and JSP 1.2 Specifications.
- Download Apache Tomcat 4.0 Server to experiment with servlets and JSP pages.

About the author

Seema Manivannan
Seema Manivannan has a Bachelor of Technology degree in Electrical and Electronics Engineering and a PG in Advanced Computing from C-DAC. Her work experience includes software development, teaching, and content development in Java programming and related technologies. She holds SCJP, SCWCD, and SCBCD certifications. She has been with Whizlabs for over two years, where she has co-authored the Sun certification exam simulators. She is an experienced corporate trainer and conducts Instructor-led online training for the SCJP, SCWCD, and SCBCD certification exams for Whizlabs. She is also the moderator of the Whizlabs SCBCD discussion forum. You can reach her at seema@whizlabs.com.