
UNIT 3 JAVA SERVER PAGES-I

Structure	Page Nos.
3.0 Introduction	52
3.1 Objectives	53
3.2 Overview of JSP	53
3.3 Relation of Applets and Servlets with JSP	56
3.4 Scripting Elements	58
3.5 JSP Expressions	59
3.6 JSP Scriptlets	59
3.7 JSP Declarations	60
3.8 Predefined Variables	61
3.9 Creating Custom JSP Tag Libraries using Nested Tags	65
3.10 Summary	69
3.11 Solutions/Answers	70
3.12 Further Readings/References	73

3.0 INTRODUCTION

Nowadays web sites are becoming very popular. These web sites are either static or dynamic. With a *static web page*, the client requests a web page from the server and the server responds by sending back the requested file to the client. Therefore with a static web page receives an exact replica of the page that exists on the server.

But these days web site requires a lot more than static content. Therefore, these days' dynamic data is becoming very important to everything on the Web, from online banking to playing games. *Dynamic web pages* are created at the time they are requested and their content gets based on specified criteria. For example, a Web page that displays the current time is dynamic because its content changes to reflect the current time. Dynamic pages are generated by an application on the server, receiving input from the client, and responding appropriately.

Therefore, we can conclude that in today's environment, dynamic content is critical to the success of any Web site. In this unit we will learn about Java Server Pages (JSP) i.e., an exciting new technology that provides powerful and efficient creation of dynamic contents. It is a presentation layer technology that allows static Web content to be mixed with Java code. JSP allows the use of standard HTML, but adds the power and flexibility of the Java programming language. JSP does not modify static data, so page layout and "look-and-feel" can continue to be designed with current methods. This allows for a clear separation between the page design and the application. JSP also enables Web applications to be broken down into separate components. This allows HTML and design to be done without much knowledge of the Java code that is generating the dynamic data. As the name implies, JSP uses the Java programming language for creating dynamic content. Java's object-oriented design, platform independence, and protected-memory model allow for rapid application development. Built-in networking and enterprise Application Programming Interfaces (APIs) make Java an ideal language for designing client-server applications. In addition, Java allows for extremely efficient code reuse by supporting the Java Bean and Enterprise Java Bean component models.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the need of JSP;
- understand the functioning of JSP;
- understand the relation of applets and servlets with JSP;
- know about various elements of JSP;
- explain various scripting elements of JSP;
- explain various implicit objects of JSP, and
- understand the concept of custom tags and process of creating custom tag libraries in JSP.

3.2 OVERVIEW OF JSP

As you have already studied in previous units, servlets offer several improvements over other server extension methods, but still suffer from a lack of presentation and business logic separation. Therefore, developers created some servlet-based environments that provided the desired separation. Some of these servlet-based environments gained considerable acceptance in the marketplace e.g., FreeMarker and WebMacro. Parallel to the efforts of these individual developers, the Java community worked to define a standard for a servlet-based server pages environment. The outcome was what we now know as JSP. Now, let us look at a brief overview of JSP: JSP is an extremely powerful choice for Web development. It is a technology using server-side scripting that is actually translated into servlets and compiled before they are run. This gives developers a scripting interface to create powerful Java Servlets.

JSP uses server-side scripting that is actually translated into servlets and compiled before they are run

JSP pages provide tags that allow developers to perform most dynamic content operations without writing complex Java code. Advanced developers can add the full power of the Java programming language to perform advanced operations in JSP pages.

Server Pages

The goal of the server pages approach to web development is to support dynamic content without the performance problems or the difficulty of using a server API. The most effective way to make a page respond dynamically would be to simply modify the static page. Ideally, special sections to the page could be added that would be changed dynamically by the server. In this case pages become more like a page *template* for the server to process before sending. These are no longer normal web pages—they are now *server pages*.

The most popular server page approaches today are Microsoft Active Server Pages (ASP), JSP from Sun Microsystems Inc., and an open-source approach called PHP. Now, as you know, server pages development simplifies dynamic web development by allowing programmers to embed bits of program logic directly into their HTML pages. This embedded program logic is written in a simple scripting language, which depends on what your server supports. This scripting language could be VBScript, JavaScript, Java, or something else. At runtime, the server interprets this script and returns the results of the script's execution to the client. This process is shown in

Figure 1. In this Figure, the client requests a server page; the server replaces some sections of a template with new data, and sends this newly modified page to the client.

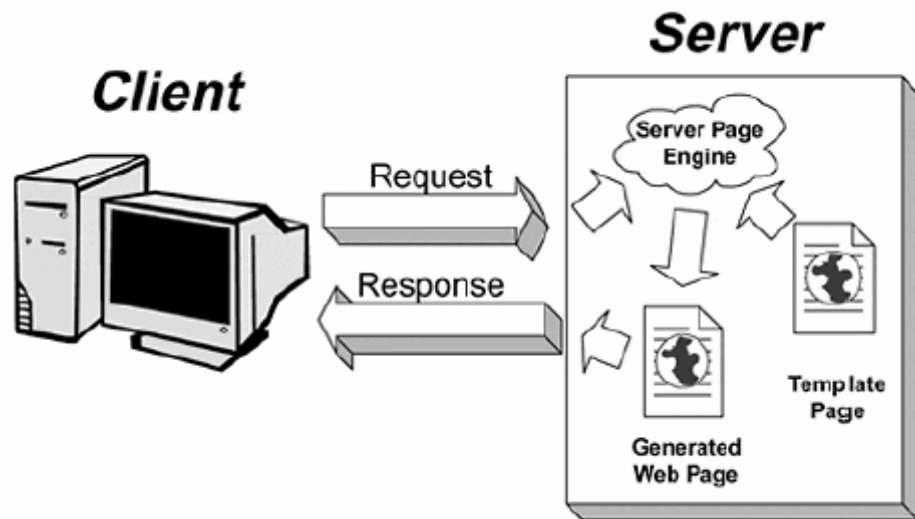


Figure 1: Server page

Separating Business and Presentation Logic

One of the greatest challenges in web development is in cleanly separating presentation and business logic. Most of the web server extension methods have suffered from this obstacle.

What does it mean to separate these layers? To start with, we can partition any application into two parts:

- **Business logic**

It is the portion of the application that solves the business need, e.g., the logic to look into the user's account, draw money and invest it in a certain stock. Implementing the business logic often requires a great deal of coding and debugging, and is the task of the programmer.

- **Presentation layer**

Presentation layer takes the results from the business logic execution and displays them to the user. The goal of the presentation layer is to create dynamic content and return it to the user's browser, which means that those responsible for the presentation layer are graphics designers and HTML developers.

Now, the question arises that if, applications are composed of a presentation layer and a business logic layer, what separates them, and why would we want to keep them apart? Clearly, there needs to be interaction between the presentation layer and the business logic, since, the presentation layer presents the business logic's results. But how much interaction should there be, and where do we place the various parts? At one extreme, the presentation and the business logic are implemented in the same set of files in a tightly coupled manner, so there is no separation between the two. At the other extreme, the presentation resides in a module totally separate from the one implementing the business logic, and the interaction between the two is defined by a set of well-known interfaces. This type of application provides the necessary

separation between the presentation and the business logic. But this separation is so crucial. Reason is explained here:

In most cases the developers of the presentation layer and the business logic are different people with different sets of skills. Usually, the developers of the presentation layer are graphics designers and HTML developers who are not necessarily skilled programmers. Their main goal is to create an easy-to-use, attractive web page. The goal of programmers who develop the business logic is to create a stable and scalable application that can feed the presentation layer with data. These two developers differ in the tools they use, their skill sets, their training, and their knowledge. When the layers aren't separated, the HTML and program code reside in the same place, as in CGI. Many sites built with those techniques have code that executes during a page request and returns HTML. Imagine how difficult it is to modify the User Interface if the presentation logic, for example HTML, is embedded directly in a script or compiled code. Though developers can overcome this difficulty by building template frameworks that break the presentation away from the code, this requires extra work for the developer since the extension mechanisms don't natively support such templating. Server pages technologies are not any more helpful with this problem. Many developers simply place Java, VBScript, or other scripting code directly into the same page as the HTML content. Obviously, this implies maintenance challenges as the server pages now contain content requiring the skills of both content developers and programmers. They must check that each updating of content to a specific server goes through without breaking the scripts inside the server page. This check is necessary because the server page is cluttered with code that only the business developer understands. This leaves the presentation developer walking on eggshells out of concern for preserving the work of the business logic developer. Worse, this arrangement can often cause situations in which both developers need to modify a single file, leaving them the tedious task of managing file ownership. This scenario can make maintaining a server pages-based application an expensive effort.

Separating these two layers is a problem in the other extension mechanisms, but the page-centric nature associated with server pages applications makes the problem much more pronounced. JSP separates the presentation layer (i.e., web interface logic) from the business logic (i.e. back-end content generation logic) so that web designers and web developers can work on the same web page without getting in each other's way.

Static and Dynamic contents in a JSP page

JSP pages usually contain a mixture of both static data and dynamic elements. *Static data* is never changed in the server page, and *dynamic elements* will always be interpreted and replaced before reaching the client.

JSP uses HTML or XML to incorporate static elements in a web page. Therefore, format and layout of the page in JSP is built using HTML or XML.

As well as these static elements a JSP page also contains some elements that will be interpreted and replaced by the server before reaching the client. In order to replace sections of a page, the server needs to be able to recognise the sections it needs to change. For this purpose a JSP page usually has a special set of tags to identify a portion of the page that should be modified by the server. JSP uses the `<%` tag to note the start of a JSP section, and the `%>` tag to note the end of a JSP section. JSP will interpret anything within these tags as a special section. These tags are known as scriptlets.

When the client requests a JSP page, the server translates the server page and client receives a document as HTML. This translation process used at server is displayed in

Figure 2. Since, the processing occurs on the server, the client receives what appears to be static data. As far as the client is concerned there is no difference between a server page and a standard web page. This creates a solution for dynamic pages that does not consume client resources and is completely browser neutral.

Resulting HTML

Template

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.0 Final//EN">
<HTML>
<HEAD>
<TITLE>A simple date example</TITLE>
</HEAD>
<BODY COLOR=#ffffff>
  The time on the server is
  Wed Aug 17 17:10:05 PST 2006
</BODY>
</HTML>
```

Server Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.0 Final//EN">
<HTML>
<HEAD>
<TITLE>A simple date example</TITLE>
</HEAD>
<BODY COLOR=#ffffff>
  The time on the server is
  <%= new java.util.Date() %>
</BODY>
</HTML>
```



Scriptlets

Figure 2: A Server Page into HTML Data

3.3 RELATION OF APPLETS AND SERVLETS WITH JSP

Now, in this topic we shall compare applets, servlets and JSP and shall try to make a relationship among these.

Let us start with the **Applets**. These are small programs that are downloaded into a Java Enabled Web Browser, like, Netscape Navigator or Microsoft Internet Explorer. The browser will execute the applet on the client's machine. The downloaded applet has very limited access to the client machine's file system and network capabilities. These limitations ensure that the applet can't perform any malicious activity on the client's machine, such as deleting files or installing viruses. By default, a downloaded applet cannot read or write files from the file system, and may use the network only to communicate back to the server of origin. Using security certificates, applets can be given permission to do anything on the user's machine that a normal Java application can do. This may be impractical for Extranet applications; however, as users may require support to give these permissions or may not trust an organization enough to grant such permission.

Applets greatly *enhance* the user interface available through a browser. Applets can be created to act exactly like any other client-server GUI application including menus, popup dialog windows, and many other user-friendly features not otherwise available in a web browser environment.

But main *problem* with applet is it's long setup time over modems. Applets need to be downloaded over the Internet. Instead of just downloading the information to be displayed, a browser must download the whole application to execute it. The more functionality the applet provides, the longer it will take to download. Therefore, applets are best suited for applications that either run on an Intranet, or are small enough to download quickly and don't require special security access.

Next, **Servlet** is a Java program that runs in conjunction with a Web Server. A servlet is executed in response to an HTTP request from a client browser. The servlet executes and then returns an HTML page back to the browser.

Some major advantages of servlets are:

- Servlets handle multiple requests. Once a servlet is started it remains in memory and can handle multiple HTTP requests. In contrast, other server side script e.g. CGI program ends after each request and must be restarted for each subsequent request, reducing performance.
- Servlets support server side execution. Servlets do not run on the client, all execution takes place on the server. While, they provide the advantages of generating dynamic content, they do not levy the same download time requirement as applets.

Major *problem* with servlets is their limited functionality. Since they deliver HTML pages to their clients, the user interface available through a servlet is limited by what the HTML specification supports.

Next, as you know, a **JSP** is text document that describes how a server should handle specific requests. A JSP is run by a JSP Server, which interprets the JSP and performs the actions the page describes. Frequently, the *JSP server compiles the JSP into a servlet* to enhance performance. The server would then periodically check the JSP for changes and if there is any change in JSP, the server will recompile it into a servlet. *JSPs have the same advantages and disadvantages as servlets when compared to applets.*

- **JSP is Easier to Develop and Maintain than Servlets**

To the developer, JSPs look very similar to static HTML pages, except that they contain special tags used to identify and define areas that contain Java functionality. Because of the close relationship between JSPs and the resulting HTML page, JSPs are easier to develop than a servlet that performs similar operations. Because they do not need to be compiled, JSPs are easier to maintain and enhance than servlets.

- **JSP's Initial Access Slower than Servlets**

However, because they need to be interpreted or compiled by the server, response time for initial accesses may be slower than servlets.

Check Your Progress 1

Give right choice for the following:

- 1) JSP uses server-side scripting that is actually translated into ----- and compiled before they are run
 - a) Applet
 - b) Servlets
 - c) HTML
- 2) Presentation layer defines -----
 - a) Web interface logic
 - b) Back-end content generation logic

Explain following question in brief

- 3) What is JSP ? Explain its role in the development of web sites.

.....
.....
.....

3.4 SCRIPTING ELEMENTS

Now, after going through the basic concepts of JSP, we will understand different types of tags or scripting elements used in JSP.

A JSP page contains HTML (or other text-based format such as XML) mixed with elements of the JSP syntax.

There are five basic types of elements, as well as a special format for comments. These are:

1. **Scriptlets :**The Scriptlet element allows Java code to be embedded directly into a JSP page.
2. **Expressions:**An expression element is a Java language expression whose value is evaluated and returned as a string to the page.
3. **Declarations:** A declaration element is used to declare methods and variables that are initialized with the page.
4. **Actions:** Action elements provide information for the translation phase of the JSP page, and consist of a set of standard, built-in methods. Custom actions can also be created in the form of custom tags. This is a new feature of the JSP 1.1 specification.
5. **Directives:** Directive elements contain global information that is applicable to the whole page.

The first three elements—Scriptlets, Expressions, and Declarations—are collectively called **scripting elements**.

There are two different *formats* in which these elements can be used in a JSP page:

- **JSP Syntax**

The first type of format is called the JSP syntax. It is based on the syntax of other Server Pages, so it might seem very familiar. It is symbolized by: `<% script %>`. The JSP specification refers to this format as the “friendly” syntax, as it is meant for hand-authoring.

JSP Syntax: `<% code %>`

- **XML Standard Format**

The second format is an XML standard format for creating JSP pages. This format is symbolized by: `<jsp:element />`.

XML syntax would produce the same results, but JSP syntax is recommended for authoring.

XML Syntax: <jsp:scriptlet > code </jsp:scriptlet>

Now, we will discuss about these scripting elements in detail.

3.5 JSP EXPRESSIONS

JSP Syntax: <%= code %>

XML Syntax: <jsp:expression > code </jsp:expression>

Printing the output of a Java fragment is one of the most common tasks utilized in JSP pages. For this purpose, we can use the `out.println()` method. But having several `out.println()` method tends to be cumbersome. Realizing this, the authors of the JSP specification created the Expression element. The Expression element begins with the standard JSP start tag followed by an equals sign (<%=).

Look at example 3.1. In this example, notice that the `out.println()` method is removed, and immediately after the opening JSP tag there is an equals symbol.

Example 3.1 date.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">
```

```
<HTML>
```

```
  <HEAD>
```

```
    <TITLE>Current Date</TITLE>
```

```
  </HEAD>
```

```
  <BODY>
```

```
    The current date is:
```

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

```
  </BODY>
```

```
</HTML>
```

Expression element



3.6 JSP SCRIPTLETS

JSP Syntax: <% code %>

XML Syntax: <jsp:scriptlet > code </jsp:scriptlet>


Scriptlets are the most common JSP syntax element. As you have studied above, a scriptlet is a portion of regular Java code embedded in the JSP content within <% ... %> tags. The Java code in scriptlets is executed when the user asks for the page. Scriptlets can be used to do absolutely anything the Java language supports, but some of their more common tasks are:

- Executing logic on the server side; for example, accessing a database.
- Implementing conditional HTML by posing a condition on the execution of portions of the page.
- Looping over JSP fragments, enabling operations such as populating a table with dynamic content.

A simple use of these scriptlet tags is shown in Example 3.2. In this example you need to notice the two types of data in the page, i.e., static data and dynamic data. Here, you need not to worry too much about what the JSP page is doing; that will be covered in later chapters.

Example 3.2 simpleDate.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">
<HTML>
<HEAD>
    <TITLE>A simple date example</TITLE>
</HEAD>
<BODY COLOR=#ffffff>
    The time on the server is
    <%= new java.util.Date() %>
</BODY>
</HTML>
```



When the client requests this JSP page, the client will receive a document as HTML. The translation process used at server is displayed in *Figure 2*.

3.7 JSP DECLARATIONS

JSP Syntax: `<%! code %>`

XML Syntax: `<jsp:declaration> code </jsp:declaration>`

The third type of Scripting element is the Declaration element. The purpose of a declaration element is to initialize variables and methods and make them available to other Declarations, Scriptlets, and Expressions. Variables and methods created within Declaration elements are effectively global. The syntax of the Declaration element begins with the standard JSP open tag followed by an exclamation point (`<%!`). The Declaration element must be a complete Java statement. It ends with a semicolon, just as the Scriptlet element does.

Look at example 3.3. In this example an instance variable named `Obj` and the initialization and finalisation methods `jspInit` and `jspDestroy`, have been done using declaration element.

Example 3.3 Declaration element

```
<%!
    private ClasJsp Obj;
    public void jspInit()
    {
        ...
    }
    public void jspDestroy()
    {
        ...
    }
%>
```

Check Your Progress 2

Give right choice for the following:

- 1) ----- contain global information that is applicable to the whole page.
 - a) Actions elements
 - b) Directive elements
 - c) Declarations elements
 - d) Scriptlets elements
- 2) For incorporating Java code with HTML, we will use -----.
 - a) Actions elements
 - b) Directive elements
 - c) Declarations elements
 - d) Scriptlets elements

Explain the following question in brief

- 3) Explain various scripting elements used in JSP.

.....

.....

.....

3.8 PREDEFINED VARIABLES

To simplify code in JSP expressions and scriptlets, Servlet also creates several objects to be used by the JSP engine; these are sometimes called implicit objects (or predefined variables). Many of these objects are called directly without being explicitly declared. These objects are:

1. The out Object
2. The request Object
3. The response Object
4. The pageContext Object
5. The session object
6. The application Object
7. The config Object
8. The page Object
9. The exception Object

- **The out Object**

The major function of JSP is to describe data being sent to an output stream in response to a client request. This output stream is exposed to the JSP author through the implicit out object. The out object is an instantiation of a `javax.servlet.jsp.JspWriter` object. This object may represent a direct reference to the output stream, a filtered stream, or a nested `JspWriter` from another JSP. Output should never be sent directly to the output stream, because there may be several output streams during the lifecycle of the JSP.

The initial `JspWriter` object is instantiated differently depending on whether the page is buffered or not. By default, every JSP page has buffering turned on, which almost

always improves performance. Buffering be easily turned off by using the buffered= 'false' attribute of the page directive.

A buffered out object collects and sends data in blocks, typically providing the best total throughput. With buffering the PrintWriter is created when the first block is sent, actually the first time that flush() is called.

With unbuffered output the PrintWriter object will be immediately created and referenced to the out object. In this situation, data sent to the out object is immediately sent to the output stream. The PrintWriter will be created using the default settings and header information determined by the server.

In the case of a buffered out object the OutputStream is not established until the first time that the buffer is flushed. When the buffer gets flushed depends largely on the autoFlush and bufferSize attributes of the page directive. It is usually best to set the header information before anything is sent to the out object. It is very difficult to set page headers with an unbuffered out object. When an unbuffered page is created the OutputStream is established almost immediately.

The sending headers after the OutputStream has been established can result in a number of unexpected behaviours. Some headers will simply be ignored, others may generate exceptions such as IllegalStateException.

The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions. In JSP these exceptions need to be explicitly caught and dealt with. More about the out object is covered in Chapter 6.

Setting the autoFlush= 'false' attribute of the page directives will cause a buffer overflow to throw an exception.

- **The request Object**

Each time a client requests a page the JSP engine creates a new object to represent that request. This new object is an instance of javax.servlet.http.HttpServletRequest and is given parameters describing the request. This object is exposed to the JSP author through the request object.

Through the request object the JSP page is able to react to input received from the client. Request parameters are stored in special name/value pairs that can be retrieved using the request.getParameter(name) method.

The request object also provides methods to retrieve header information and cookie data. It provides means to identify both the client and the server, e.g., it uses request.getRequestURI() and request.getServerName() to identify the server.

The request object is inherently limited to the request scope. Regardless of how the page directives have set the scope of the page, this object will always be recreated with each request. For each separate request from a client there will be a corresponding request object.

- **The response Object**

Just as the server creates the request object, it also creates an object to represent the response to the client.

The object is an instance of `javax.servlet.http.HttpServletResponse` and is exposed to the JSP author as the response object.

The response object deals with the stream of data back to the client. The out object is very closely related to the response object. The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP author can add new cookies or date stamps, change the MIME content type of the page, or start “server-push” methods. The response object also contains enough information on the HTTP to be able to return HTTP status codes, such as forcing page redirects.

- **The pageContext Object**

The pageContext object is used to represent the entire JSP page. It is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The application, config, session, and out objects are derived by accessing attributes of this object. The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the `errorPageURL`, and page scope. The pageContext object does more than just act as a data repository. It is this object that manages nested JSP pages, performing most of the work involved with the forward and include actions. The pageContext object also handles uncaught exceptions.

From the perspective of the JSP author this object is useful in deriving information about the current JSP page's environment. This can be particularly useful in creating components where behavior may be different based on the JSP page directives.

- **The session object**

The session object is used to track information about a particular client while using stateless connection protocols, such as HTTP. Sessions can be used to store arbitrary information between client requests.

Each session should correspond to only one client and can exist throughout multiple requests. Sessions are often tracked by URL rewriting or cookies, but the method for tracking of the requesting client is not important to the session object.

The session object is an instance of `javax.servlet.http.HttpSession` and behaves exactly the same way that session objects behave under Java Servlets.

- **The application Object**

The application object is direct wrapper around the `ServletContext` object for the generated Servlet. It has the same methods and interfaces that the `ServletContext` object does in programming Java Servlets.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the `jspDestroy()` method, the JSP page is recompiled, or the JVM crashes. Information stored in this object remains available to any object used within the JSP page.

The application object also provides a means for a JSP to communicate back to the server in a way that does not involve “requests”. This can be useful for finding out information about the MIME type of a file, sending log information directly out to the servers log, or communicating with other servers.

- **The config Object**

The config object is an instantiation of `javax.servlet.ServletConfig`. This object is a direct wrapper around the `ServletConfig` object for the generated servlet. It has the same methods and interfaces that the `ServletConfig` object does in programming Java Servlets. This object allows the JSP author access to the initialisation parameters for the Servlet or JSP engine. This can be useful in deriving standard global information, such as the paths or file locations.

- **The page Object**

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. When the JSP page is first instantiated the page object is created by obtaining a reference to this object. So, the page object is really a direct synonym for this object.

However, during the JSP lifecycle, this object may not refer to the page itself. Within the context of the JSP page, the page object will remain constant and will always represent the entire JSP page.

- **The exception Object**

The error handling method utilises this object. It is available only when the previous JSP page throws an uncaught exception and the `<% @ pageerrorPage= " ..." %>` tag was used. The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

A summarized picture of these predefined variables (implicit objects) is given in *Table 4*.

Table 4: Implicit Objects in JSP

Variable	Class	Description
out	<code>javax.servlet.jsp.JspWriter</code>	The output stream.
request	Subtype of <code>javax.servlet.HttpServletRequest</code>	The request triggering the execution of the JSP page.
response	Subtype of <code>javax.servlet.HttpServletResponse</code>	The response to be returned to the client. Not typically used by JSP page authors.
pageContext	<code>javax.servlet.jsp.PageContext</code>	The context for the JSP page. Provides a single API to manage the various scoped attributes described in Sharing Information. This API is used extensively when implementing tag handlers.
Session	<code>javax.servlet.http.HttpSession</code>	The session object for the client..
application	<code>javax.servlet.ServletContext</code>	The context for the JSP page's servlet and any Web components contained in the same application.
config	<code>javax.servlet.ServletConfig</code>	Initialization information for the JSP page's servlet.
page	<code>java.lang.Object</code>	The instance of the JSP page's servlet processing the current request. Not typically used by JSP page authors.
exception	<code>java.lang.Throwable</code>	Accessible only from an error page.

3.9 CREATING CUSTOM JSP TAG LIBRARIES USING NESTED TAGS

Ok up to now, you have studied the basic elements of JSP. Now in this topic we will learn about the creation of custom tag libraries in JSP.

As you already know, a **tag** is a group of characters read by a program for the purpose of instructing the program to perform an action. In the case of HTML tags, the program reading the tags is a web browser, and the actions range from painting words or objects on the screen to creating forms for data collection.

In the same way, a **custom tag** is a user-defined JSP language element. Custom JSP tags are also interpreted by a program; but, unlike HTML, JSP tags are interpreted on the server side not client side. The program that interprets custom JSP tags is the runtime engine in your application server as Tomcat, JRun, WebLogic etc. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The web container then invokes those operations when the JSP page's servlet is executed.

Custom tags have a rich set of *features*. They can

- Be customised via attributes passed from the calling page.
- Access all the objects available to JSP pages.
- Modify the response generated by the calling page.
- Communicate with each other. You can create and initialize a JavaBeans component, create a variable that refers to that bean in one tag, and then use the bean in another tag.
- Be nested within one another, allowing for complex interactions within a JSP page.

Custom Tag Syntax

The syntax of custom tag is exactly the same as the syntax of JSP actions. A slight difference between the syntax of JSP actions and custom tag is that the JSP action prefix is `jsp`, while a custom tag prefix is determined by the prefix attribute of the `taglib` directive used to instantiate a set of custom tags. The prefix is followed by a colon and the name of the tag itself.

As shown below, the format of a standard custom tag looks like:

```
<utility:repeat number= "12">Hello World!</utility:repeat>
```

Here, a tag library named `utility` is referenced. The specific tag used is named `repeat`. The tag has an attribute named `number`, which is assigned a value of `"12"`. The tag contains a body that has the text `"Hello World!"`, and then the tag is closed.

The Components That Make Up a Tag Library

To use custom JSP tags, you need to define three separate components:

- a) **Tag handler class** that defines the tag's behaviour,
- b) **Tag library descriptor file** that maps the XML element names to the tag implementations and
- c) **The JSP file** that uses the tag library.

Now in the following section, we will read an overview of each of these components, and learn how to build these components for various styles of tags.

- **The Tag Handler Class**

To define a new tag, first you have to define a Java class that tells the system what to do when it sees the tag. This class must implement the `javax.servlet.jsp.tagext.Tag` interface. This is usually accomplished by extending the `TagSupport` or `BodyTagSupport` class. Example 3.4 is an example of a simple tag that just inserts “Custom tag example (coreservlets.tags.ExampleTag)” into the JSP page wherever the corresponding tag is used.

Don’t worry about understanding the exact behavior of this class. For now, just note that it is in the `coreservlets.tags` class and is called `ExampleTag`. Thus, with Tomcat 3.1, the class file would be in `install_dir/webapps/ROOT/WEB-INF/classes/coreservlets/tags/ExampleTag.class`.

Example 3.4 ExampleTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Very simple JSP tag that just inserts a string
 * (“Custom tag example...”) into the output.
 * The actual name of the tag is not defined here;
 * that is given by the Tag Library Descriptor (TLD)
 * file that is referenced by the taglib directive
 * in the JSP file.
 */

public class ExampleTag extends TagSupport {
    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.print(“Custom tag example “ +
                “(coreservlets.tags.ExampleTag)”);
        } catch(IOException ioe) {
            System.out.println(“Error in ExampleTag: ” + ioe);
        }
        return(SKIP_BODY);
    }
}
```

- **The Tag Library Descriptor File**

After defining a tag handler, your next task is to identify the class to the server and to associate it with a particular XML tag name. This task is accomplished by means of a tag library descriptor file (in XML format) like the one shown in Example 3.5. This

file contains some fixed information, an arbitrary short name for your library, a short description, and a series of tag descriptions. The bold part of the example is the same in virtually all tag library descriptors.

Don't worry about the format of tag descriptions. For now, just note that the tag element defines the main name of the tag (really tag suffix, as will be seen shortly) and identifies the class that handles the tag. Since, the tag handler class is in the `coreservlets.tags` package, the fully qualified class name of `coreservlets.tags.ExampleTag` is used. Note that this is a class name, not a URL or relative path name. The class can be installed anywhere on the server that beans or other supporting classes can be put. With Tomcat 3.1, the standard base location is `install_dir/webapps/ROOT/WEB-INF/classes`, so `ExampleTag` would be in `install_dir/webapps/ROOT/WEB-INF/classes/coreservlets/tags`. Although it is always a good idea to put your servlet classes in packages, a surprising feature of Tomcat 3.1 is that tag handlers are required to be in packages.

Example 3.5 `csajsp-taglib.tld`

```
<?xml version= "1.0" encoding= "ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN "
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->
<taglib>
  <!-- after this the default space is
    "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
  -->
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>
  <tag>
    <name>example</name>
    <tagclass>coreservlets.tags.ExampleTag</tagclass>
    <info>Simplest example: inserts one line of output</info>
    <bodycontent>EMPTY</bodycontent>
  </tag>
  <!-- Other tags defined later... -->
</taglib>
```


- **The JSP File**

Once, you have a tag handler implementation and a tag library description, you are ready to write a JSP file that makes use of the tag. Example 3.6 shows a JSP file. Somewhere before the first use of your tag, you need to use the taglib directive. This directive has the following form:

```
<%@ taglib uri= “...” prefix= “...” %>
```

The required uri attribute can be either an absolute or relative URL referring to a tag library descriptor file like the one shown in Example 3.5. To complicate matters a little, however, Tomcat 3.1 uses a web.xml file that maps an absolute URL for a tag library descriptor to a file on the local system. I don't recommend that you use this approach.

The prefix attribute, also required, specifies a prefix that will be used in front of whatever tag name the tag library descriptor defined. For example, if the TLD file defines a tag named tag1 and the prefix attribute has a value of test, the actual tag name would be test:tag1. This tag could be used in either of the following two ways, depending on whether it is defined to be a container that makes use of the tag body:

```
<test:tag1>
    Arbitrary JSP
</test:tag1>
or just
<test:tag1 />
```

To illustrate, the descriptor file of Example 3.5 is called csajsp-taglib.tld, and resides in the same directory as the JSP file shown in Example 3.6. Thus, the taglib directive in the JSP file uses a simple relative URL giving just the filename, as shown below.

```
<%@ taglib uri= “csajsp-taglib.tld” prefix= “csajsp” %>
```

Furthermore, since the prefix attribute is csajsp (for Core Servlets and JavaServer Pages), the rest of the JSP page uses csajsp:example to refer to the example tag defined in the descriptor file.

Example 3.6 SimpleExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<% @ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
<TITLE><csajsp:example /></TITLE>
<LINK REL=STYLESHEET
    HREF="JSP-Styles.css"
    TYPE="text/css">
</HEAD>
<BODY>
<H1><csajsp:example /></H1>
<csajsp:example />
</BODY>
</HTML>
```



Figure 3: Custom tag example

Check Your Progress 3

Give right choice for the following:

- 1) ----- object is an instantiation of a `avax.servlet.jsp.JspWriter` object.
 - a) page
 - b) config
 - c) out
 - d) response
- 2) ----- object is used to track information about a particular client while using stateless connection protocols.
 - a) request
 - b) session
 - c) out
 - d) application

Explain following questions in brief

- 3) What are various implicit objects used with JSP..

.....

.....

.....
- 4) What is a custom tags. in JSP? What are the components that make up a tag library in JSP?

.....

.....

.....

3.10 SUMMARY

In this unit, we first studied the static and dynamic web pages. With a static web page, the client requests a web page from the server and the server responds by sending back the requested file to the client, server doesn't process the requested page at its

end. But dynamic web pages are created at the time they are requested and their content gets based on specified criteria. These pages are generated by an application on the server, receiving input from the client, and responding appropriately. For creation of these dynamic web pages, we can use JSP; it is a technology using server-side scripting that is actually translated into servlets and compiled before they are run. It separates the presentation layer (i.e., web interface logic) from the business logic (i.e., back-end content generation logic) so that web designers and web developers can work on the same web page without getting in each other's way.

There are five basic types of elements in JSP. These are *scriptlets*, *expressions*, *declarations*, *actions* and *directives*. Among these elements the first three elements, i.e., scriptlets, expressions, and declarations, are collectively called scripting elements. Here **scriptlet** (`<%...%>`) element allows Java code to be embedded directly into a JSP page, **expression element** (`<%=...%>`) is used to print the output of a Java fragment and **declaration element** (`<%! code %>`) is used to initialise variables and methods and make them available to other declarations, scriptlets, and expressions. Next, we discussed about the implicit objects of JSP. Various implicit objects of JSP are out, request, response, pageContext, session, application, config, page and exception object. Here, **out object** refers to the output stream, **request object** contains parameters describing the request made by a client to JSP engine, **response object** deals with the stream of data back to the client, **pageContext object** is used to represent the entire JSP page, **session object** is used to track information about a particular client while using stateless connection protocols such as HTTP, **application object** is a representation of the JSP page through its entire lifecycle, **config object** allows the JSP author access to the initialisation parameters for the servlet or JSP engine, **page object** is an actual reference to the instance of the page and **exception object** is a wrapper containing the exception thrown from the previous page. Finally we studied about the custom tags. These are user-defined JSP language elements. Unlike HTML, these custom tags (JSP tags) are interpreted on the server side not client side. To use custom JSP tags, you need to define three separate components, i.e., tag handler class, tag library descriptor file and the JSP file. Here, tag handler class defines the tag's behaviour, tag library descriptor file maps the XML element names to the tag implementations and the JSP file uses the tag library.

3.11 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) b
- 2) a
- 3) JSP is an exciting new technology that provides powerful and efficient creation of dynamic contents. It allows static web content to be mixed with Java code. It is a technology using server-side scripting that is actually translated into servlets and compiled before they are run. This gives developers a scripting interface to create powerful Java Servlets.

Role of JSP in the development of websites:

In today's environment, dynamic content is critical to the success of any web site. There are a number of technologies available for incorporating the dynamic contents in a site. But most of these technologies have some problems. Servlets offer several improvements over other server extension methods, but still suffer from a lack of presentation and business logic separation. Therefore the Java community worked to define a standard for a servlet-based server pages environment and the outcome was what we now know as JSP. JSP separates the

presentation layer (i.e., web interface logic) from the business logic (i.e., back-end content generation logic) so that web designers and web developers can work on the same web page without getting in each other's way.

Check Your Progress 2

- 1) b
- 2) d
- 3) There are five basic types of elements used in JSP. These are:

(i) Scriptlets

JSP Syntax: `<% code %>`

XML Syntax: `<jsp:scriptlet > code </jsp:scriptlet>`

The Scriptlet element allows Java code to be embedded directly into a JSP page.

(ii) Expressions

JSP Syntax: `<%= code %>`

XML Syntax: `<jsp:expression > code </jsp:expression>`

An expression element is a Java language expression whose value is evaluated and returned as a string to the page.

(iii) Declarations

JSP Syntax: `<%! code %>`

XML Syntax: `<jsp:declaration> code </jsp:declaration>`

A declaration element is used to declare methods and variables that are initialized with the page.

(iv) Actions

Action elements provide information for the translation phase of the JSP page, and consist of a set of standard, built-in methods. Custom actions can also be created in the form of custom tags. This is a new feature of the JSP 1.1 specification.

(v) Directives

Directive elements contain global information that is applicable to the whole page.

The first three elements — Scriptlets, Expressions, and Declarations — are collectively called **scripting elements**.

Check Your Progress 3

- 1) c
- 2) c
- 3) To simplify code in JSP expressions and scriptlets, servlet creates several objects to be used by the JSP engine; these are sometimes called implicit objects. These objects are:

i) **The out object:** It refers to the output stream

- ii) **The request object:** It contains parameters describing the request made by a client to JSP engine.
 - iii) **The response object:** This object deals with the stream of data back to the client.
 - iv) **The pageContext object:** This object is used to represent the entire JSP page.
 - v) **The session object:** This object is used to track information about a particular client while using stateless connection protocols such as HTTP.
 - vi) **The application object:** This object is a representation of the JSP page through its entire lifecycle.
 - vii) **The config object:** This object allows the JSP author access to the initialization parameters for the servlet or JSP engine.
 - viii) **The page object:** This object is an actual reference to the instance of the page.
 - ix) **The exception object:** This object is a wrapper containing the exception thrown from the previous page.
- 4) A custom tag is a user-defined JSP language element. Custom JSP tags are also interpreted by a program; but, unlike HTML, JSP tags are interpreted on the server side not client side. The program that interprets custom JSP tags is the runtime engine in the application server as Tomcat, JRun, WebLogic, etc. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The web container then invokes those operations when the JSP page's servlet is executed.

Custom tags have a rich set of *features*. They can

- Be customized via attributes passed from the calling page.
- Access all the objects available to JSP pages.
- Modify the response generated by the calling page.
- Communicate with each other.

To use custom JSP tags, we need to define three separate components:

- a) **Tag handler class** that defines the tag's behaviour,
- b) **Tag library descriptor file** that maps the XML element names to the tag implementations, and
- c) **The JSP file** that uses the tag library.

3.12 FURTHER READINGS/REFERENCES

- Phil Hanna, *JSP: The Complete Reference*
- Hall, Marty, *Core Servlets and JavaServer Pages*
- Annunziato Jose & Fesler, Stephanie, *Teach Yourself JavaServer Pages in 24 Hours*,
- Bruce W. Perry, *Java Servlet & JSP, Cookbook* (Paperback).