**JSP**

**What are implicit objects in JSP?**

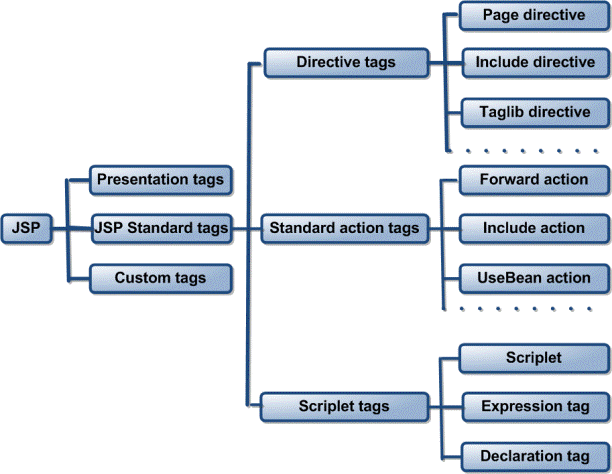
Implicit objects in JSP are the Java objects that the JSP Container makes available to developers in each page. These objects need not be declared or instantiated by the JSP author. They are automatically instantiated by the container and are accessed using standard variables; hence, they are called implicit objects.The implicit objects available in JSP are as follows:

* request
* response
* pageContext
* session
* application
* out
* config
* page
* exception

The implicit objects are parsed by the container and inserted into the generated servlet code. They are available only within the jspService method and not in any declaration.

**What are the different types of JSP tags?**

The different types of JSP tags are as follows:



**What are JSP directives?**

* JSP directives are messages for the JSP engine. i.e., JSP directives serve as a message from a JSP page to the JSP container and control the processing of the entire page
* They are used to set global values such as a class declaration, method implementation, output content type, etc.
* They do not produce any output to the client.
* Directives are always enclosed within <%@ ….. %> tag.
* Ex: page directive, include directive, etc.

**11.What is page directive?**

* A page directive is to inform the JSP engine about the headers or facilities that page should get from the environment.
* Typically, the page directive is found at the top of almost all of our JSP pages.
* There can be any number of page directives within a JSP page (although the attribute – value pair must be unique).
* The syntax of the include directive is: <%@ page attribute="value">
* Example:<%@ include file="header.jsp" %>

**12.What are the attributes of page directive?**

There are thirteen attributes defined for a page directive of which the **important** attributes are as follows:

* **import**: It specifies the packages that are to be imported.
* **session**: It specifies whether a session data is available to the JSP page.
* **contentType**: It allows a user to set the content-type for a page.
* **isELIgnored**: It specifies whether the EL expressions are ignored when a JSP is translated to a servlet.

**13.What is the include directive?**

There are thirteen attributes defined for a page directive of which the **important** attributes are as follows:

* The include directive is used to statically insert the contents of a resource into the current JSP.
* This enables a user to reuse the code without duplicating it, and includes the contents of the specified file at the translation time.
* The syntax of the include directive is as follows:  
  <%@ include file = "FileName" %>
* This directive has only one attribute called file that specifies the name of the file to be included.

**14.What are the JSP standard actions?**

* The JSP standard actions affect the overall runtime behavior of a JSP page and also the response sent back to the client.
* They can be used to include a file at the request time, to find or instantiate a JavaBean, to forward a request to a new page, to generate a browser-specific code, etc.
* Ex: include, forward, useBean,etc. object

**15.What are the standard actions available in JSP?**

The standard actions available in JSP are as follows:

* **<jsp:include>**: It includes a response from a servlet or a JSP page into the current page. It differs from an include directive in that it includes a resource at request processing time, whereas the include directive includes a resource at translation time.
* **<jsp:forward>**: It forwards a response from a servlet or a JSP page to another page.
* **<jsp:useBean>**: It makes a JavaBean available to a page and instantiates the bean.
* **<jsp:setProperty>**: It sets the properties for a JavaBean.
* **<jsp:getProperty>**: It gets the value of a property from a JavaBean component and adds it to the response.
* **<jsp:param>**: It is used in conjunction with <jsp:forward>;, <jsp:, or plugin>; to add a parameter to a request. These parameters are provided using the name-value pairs.
* **<jsp:plugin>**: It is used to include a Java applet or a JavaBean in the current JSP page.

**16.What is the <jsp:useBean> standard action?**

The <jsp:useBean> standard action is used to locate an existing JavaBean or to create a JavaBean if it does not exist. It has attributes to identify the object instance, to specify the lifetime of the bean, and to specify the fully qualified classpath and type.

**17.What are the scopes available in <jsp:useBean>?**

The scopes available in <jsp:useBean> are as follows:

* **page scope:**: It specifies that the object will be available for the entire JSP page but not outside the page.
* **request scope**: It specifies that the object will be associated with a particular request and exist as long as the request exists.
* **application scope**: It specifies that the object will be available throughout the entire Web application but not outside the application.
* **session scope**: It specifies that the object will be available throughout the session with a particular client.

**18.What is the <jsp:forward> standard action?**

* The <jsp:forward> standard action forwards a response from a servlet or a JSP page to another page.
* The execution of the current page is stopped and control is transferred to the forwarded page.
* The syntax of the <jsp:forward> standard action is :    
  <jsp:forward page="/targetPage" />  
  Here, targetPage can be a JSP page, an HTML page, or a servlet within the same context.
* If anything is written to the output stream that is not buffered before <jsp:forward>, an IllegalStateException will be thrown.

*Note* : Whenever we intend to use <jsp:forward> or <jsp:include> in a page, buffering should be enabled. By default buffer is enabled.

**19.What is the <jsp:include> standard action?**

The <jsp:include> standard action enables the current JSP page to include a static or a dynamic resource at runtime. In contrast to the include directive, the include action is used for resources that change frequently. The resource to be included must be in the same context.The syntax of the <jsp:include> standard action is as follows:  
<jsp:include page="targetPage" flush="true"/>   
Here, targetPage is the page to be included in the current JSP.

**20.What is the difference between include directive and include action?**

|  |  |
| --- | --- |
| **Include directive** | **Include action** |
| The *include* directive, includes the content of the specified file during the translation phase–when the page is converted to a servlet. | The *include* action, includes the response generated by executing the specified page (a JSP page or a servlet) during the request processing phase–when the page is requested by a user. |
| The include directive is used to statically insert the contents of a resource into the current JSP. | The include standard action enables the current JSP page to include a static or a dynamic resource at runtime. |
| Use the include directive if the file changes rarely. It’s the fastest mechanism. | Use the include action only for content that changes often, and if which page to include cannot be decided until the main page is requested. |

**21.Differentiate between pageContext.include and jsp:include?**

The <jsp:include> standard action and the pageContext.include() method are both used to include resources at runtime. However, the pageContext.include() method always flushes the output of the current page before including the other components, whereas <jsp:include> flushes the output of the current page only if the value of flush is explicitly set to true as follows:

<jsp:include page="/index.jsp" flush="true"/>

**22.What is the jsp:setProperty action?**

You use jsp:setProperty to give values to properties of beans that have been referenced earlier. You can do this in two contexts. First, you can use jsp:setProperty after, but outside of, a jsp:useBean element, as below:

<jsp:useBean id="myName" ... />

...

<jsp:setProperty name="myName" property="myProperty" ... />

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.   
  
A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

<jsp:useBean id="myName" ... >

...

<jsp:setProperty name="myName"

property="someProperty" ... />

</jsp:useBean>

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

**23.What is the jsp:getProperty action?**

The <jsp:getProperty> action is used to access the properties of a bean that was set using the <jsp:getProperty> action. The container converts the property to a String as follows:

* If it is an object, it uses the toString() method to convert it to a String.
* If it is a primitive, it converts it directly to a String using the valueOf() method of the corresponding Wrapper class.
* The syntax of the <jsp:getProperty> method is: <jsp:getProperty name="Name" property="Property" />

Here, name is the id of the bean from which the property was set. The property attribute is the property to get. A user must create or locate a bean using the <jsp:useBean> action before using the <jsp:getProperty> action.

**24.What is the <jsp:param> standard action?**

The <jsp:param> standard action is used with <jsp:include> or <jsp:forward> to pass parameter names and values to the target resource. The syntax of the <jsp:param> standard action is as follows:   
<jsp:param name="paramName" value="paramValue"/>

**25.What is the jsp:plugin action ?**

This action lets you insert the browser-specific OBJECT or EMBED element needed to specify that the browser run an applet using the Java plugin.

**26.What are scripting elements?**

JSP scripting elements let you insert Java code into the servlet that will be generated from the current JSP page. There are three forms:

1. **Expressions** of the form <%= expression %> that are evaluated and inserted into the output,
2. **Scriptlets** of the form <% code %> that are inserted into the servlet's service method,
3. **Declarations** of the form <%! code %> that are inserted into the body of the servlet class, outside of any existing methods.

**27.What is a scriptlet?**

A scriptlet contains Java code that is executed every time a JSP is invoked. When a JSP is translated to a servlet, the scriptlet code goes into the service() method. Hence, methods and variables written in scriptlets are local to the service() method. A scriptlet is written between the **<% and %>** tags and is executed by the container at request processing time.

**28.What are JSP declarations?**

As the name implies, JSP declarations are used to declare class variables and methods in a JSP page. They are initialized when the class is initialized. Anything defined in a declaration is available for the whole JSP page. A declaration block is enclosed between the **<%! and %>** tags. A declaration is not included in the service() method when a JSP is translated to a servlet.

**29.What is a JSP expression?**

A JSP expression is used to write an output without using the out.print statement. It can be said as a shorthand representation for scriptlets. An expression is written between the **<%= and %>** tags. It is not required to end the expression with a semicolon, as it implicitly adds a semicolon to all the expressions within the expression tags.

**30.How is scripting disabled?**

Scripting is disabled by setting the scripting-invalid element of the deployment descriptor to true. It is a subelement of jsp-property-group. Its valid values are true and false. The syntax for disabling scripting is as follows:

<jsp-property-group>

<url-pattern>\*.jsp</url-pattern>

<scripting-invalid>true</scripting-invalid>

</jsp-property-group>

**Servlet**

In this text I will try to give you an **overview** of Java servlets.

A Java Servlet is a Java object that responds to HTTP requests. It runs inside a Servlet container. Here is an illustration of that:

|  |
| --- |
| Servlets inside a Java Servlet Container |
| **Servlets inside a Java Servlet Container** |
| A Servlet is part of a Java web application. A Servlet container may run multiple web applications at the same time, each having multiple servlets running inside. Here is an llustration of that: Web applications with multiple servlets inside a Java Servlet container |
| **Web applications with multiple servlets inside a Java Servlet container** |

A Java web application can contain other components than servlets. It can also contain Java Server Pages (JSP), Java Server Faces (JSF) and Web Services. This tutorial is about Java Servlets only, though.

## HTTP Request and Response

The browser sends an HTTP request to the Java web server. The web server checks if the request is for a servlet. If it is, the servlet container is passed the request. The servlet container will then find out which servlet the request is for, and activate that servlet. The servlet is activated by calling the Servlet.service() method.

Once the servlet has been activated via the service() method, the servlet processes the request, and generates a response. The response is then sent back to the browser.

## Servlet Containers

Java servlet containers are usually running inside a Java web server. A few common well known, free Java web servers are:

* [Jetty](http://jetty.codehaus.org/jetty/)
* [Tomcat](http://tomcat.apache.org/)

**Servlet life cycle**

A servlet follows a certain life cycle. The servlet life cycle is managed by the servlet container. The life cycle contains the following steps:

1. Load Servlet Class.
2. Create Instance of Servlet.
3. Call the servlets init() method.
4. Call the servlets service() method.
5. Call the servlets destroy() method.

Step 1, 2 and 3 are executed only once, when the servlet is initially loaded. By default the servlet is not loaded until the first request is received for it. You can force the container to load the servlet when the container starts up though. See [web.xml Servlet Configuration](http://tutorials.jenkov.com/java-servlets/web-xml.html#load-on-startup) for more details about that.

Step 4 is executed multiple times - once for every HTTP request to the servlet.  
Step 5 is executed when the servlet container unloads the servlet.  
Each step is described in more detail below:

|  |
| --- |
| The Java Servlet life cycle |
| **The Java Servlet life cycle** |

## Load Servlet Class

Before a servlet can be invoked the servlet container must first load its class definition. This is done just like any other class is loaded.

## Create Instance of Servlet

When the servlet class is loaded, the servlet container creates an instance of the servlet.

Typically, only a single isntance of the servlet is created, and concurrent requests to the servlet are executed on the same servlet instance. This is really up to the servlet container to decide, though. But typically, there is just one instance.

## Call the Servlets init() Method

When a servlet instance is created, its init() method is invoked. The init() method allows a servlet to initialize itself before the first request is processed.

You can specify init parameters to the servlet in the web.xml file. See [web.xml Servlet Configuration](http://tutorials.jenkov.com/java-servlets/web-xml.html#initParams) for more details.

## Call the Servlets service() Method

For every request received to the servlet, the servlets service() method is called. For HttpServlet subclasses, one of the doGet(), doPost() etc. methods are typically called.

As long as the servlet is active in the servlet container, the service() method can be called. Thus, this step in the life cycle can be executed multiple times.

## Call the Servlets destroy() Method

When a servlet is unloaded by the servlet container, its destroy() method is called. This step is only executed once, since a servlet is only unloaded once.

A servlet is unloaded by the container if the container shuts down, or if the container reloads the whole web application at runtime.

**Servlet Example**

A Java Servlet is just an ordinary Java class which implements the interface

javax.servlet.Servlet;

The easiest way to implement this interface is to **extend** either the class GenericServlet or HttpServlet.

import javax.servlet.GenericServlet;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import java.io.IOException;

public class SimpleServlet extends GenericServlet {

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

// do something in here

}

}

When an HTTP request arrives at the web server, targeted for your Servlet, the web server calls your Servlet's service() method.

The service() method then reads the request, and generates a response which is sent back to the client (e.g. a browser).

Here is an example service() implementation:

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

String yesOrNoParam = request.getParameter("param");

if("yes".equals(yesOrNoParam) ){

response.getWriter().write(

"<html><body>You said yes!</body></html>");

}

if("no".equals(yesOrNoParam) ){

response.getWriter().write(

"<html><body>You said no!</body></html>");

}

}

This service() method first reads the request parameter "param". Then it checks if the param is equal to the text "yes" or "no", and writes an HTML response back to the browser.

**HttpServlet**

The javax.servlet.http.HttpServlet class is a slightly more advanced base class than the GenericServlet shown in the [Simple Servlet](http://tutorials.jenkov.com/java-servlets/simple-servlet.html) example.

The HttpServlet class reads the HTTP request, and determines if the request is an HTTP GET, POST, PUT, DELETE, HEAD etc. and calls one the corresponding method.

To respond to e.g. HTTP GET requests only, you will extend the HttpServlet class, and override the doGet() method only. Here is an example:

public class SimpleHttpServlet extends HttpServlet {

protected void doGet( HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

response.getWriter().write("<html><body>GET response</body></html>");

}

}

## Session

It is possible to obtain the session object from the HttpRequest object too.

The session object can hold information about a given user, between requests. So, if you set an object into the session object during one request, it will be available for you to read during any subsequent requests within the same session time scope.

Here is how you access the session object from the HttpRequest object:

HttpSession session = request.getSession();

I will not get into more detail about the session object here. It is covered in more detail in its own text.

## ServletContext

You can access the ServletContext object from the HttpRequest object too. The ServletContext contains meta information about the web application. For instance, you can access context parameters set in the web.xml file, you can forward the request to other servlets, and you can store application wide parameters in the ServletContext too.

Here is how you access the ServletContext object from the HttpRequest object:

ServletContext context = request.getSession().getServletContext();

As you can see, you have to first get the session object, to get access to the ServletContext object.

I will not get into more detail about the ServletContext object here. It will be covered in more detail in its own text.

## Redirecting to a Different URL

You can redirect the browser to a different URL from your servlet. You cannot send any data back to the browser when redirecting. Here is how you redirect:

response.sendRedirect("http://jenkov.com");

**RequestDispatcher**

The RequestDispatcher class enables your servlet to "call" another servlet from inside another servlet. The other servlet is called as if an HTTP request was sent to it by a browser.

You can obtain a RequestDispatcher from the HttpServletRequest object, like this:

protected void doPost(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

RequestDispatcher requestDispatcher =

request.getRequestDispatcher("/anotherURL.simple");

}

The above code obtains a RequestDispatcher targeted at whatever Servlet (or JSP) that is mapped to the URL /anotherUrl.simple.

You can call the RequestDispatcher using either its include() or forward() method:

requestDispatcher.forward(request, response);

requestDispatcher.include(request, response);

By calling either the include() or forward() method the servlet container activates whatever Servlet is mapped to the URL the RequestDispatcher.

The activated servlet has access to the same request as the servlet calling it, and will write to the same response as your current servlet. That way you can merge the output of servlets into a single repsonse.

There is a little difference between calling the forward() and include() method.

The forward() method intended for use in **forwarding** the request, meaning after the response of the calling servlet has been committed. You cannot merge response output using this method.

The include() method merges the response written by the calling servlet, and the activated servlet. This way you can achieve "server side includes" using the include().

Personally, I mostly use the include() method.

**HttpSession**

The HttpSession object represents a user session. A user session contains information about the user across multiple HTTP requests.

When a user enters your site for the first time, the user is given a unique ID to identify his session by. This ID is typically stored in a cookie or in a request parameter.

Here is how you access the session object:

protected void doPost(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

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

}

You can store values in the session object, and retrieve them later. First, let's see how you can store values in the session object:

session.setAttribute("userName", "theUserName");

This code sets an attribute named "userName", with the value "theUserName".

To read the value again, you do this:

String userName = (String) session.getAttribute("userName");

Values stored in the session object are stored in the memory of the servlet container.

**Sessions and Clusters**

If you have an architecture with 2 web servers in a cluster, keep in mind that values stored in the session object of one server, may not be available in the session object on the other server. So, if a user's requests are divided evenly between the two servers, sometimes session values may be missing.

The solution to this problem would be one of:

1. Do not use session attributes.
2. Use a session database, into which session attributes are written, and from which it is read.
3. Use sticky session, where a user is always sent to the same server, throughout the whole session.

**ServletContext**

The ServletContext is an object that contains meta informaton about your web application. You can access it via the HttpRequest object, like this:

ServletContext context = request.getSession().getServletContext();

## Context Attributes

Just like in the session object you can store attributes in the servlet context. Here is how:

context.setAttribute("someValue", "aValue");

You can access the attributes again like this:

Object attribute = context.getAttribute("someValue");

The attributes stored in the ServletContext are available to all servlets in your application, and between requests and sessions. That means, that the attributes are available to all visitors of the web application. Session attributes are just available to a single user.

The ServletContext attributes are still stored in the memory of the servlet container. That means that the same problems exists as does with the session attributes, in server clusters.

**Web.xml**

For a Java servlet to be accessible from a browser, you must tell the servlet container what servlets to deploy, and what URL's to map the servlets to. This is done in the web.xml file of your Java web application.

If you are not familiar with the directory layout and contents of a Java web application, see my tutorial on the topic: [Java Web Application Directory Layout](http://tutorials.jenkov.com/java-web-apps/web-app-directory-layout.html)

Here is a list of the topics covered in this text about the web.xml servlet configuration:

1. [Configuring and Mapping a Servlet](http://tutorials.jenkov.com/java-servlets/web-xml.html#configuration)
2. [Servlet Init Parameters](http://tutorials.jenkov.com/java-servlets/web-xml.html#initParams)
3. [Servlet Load-on-Startup](http://tutorials.jenkov.com/java-servlets/web-xml.html#load-on-startup)
4. [Context Parameters](http://tutorials.jenkov.com/java-servlets/web-xml.html#contextParams)

## Configuring and Mapping a Servlet

To configure a servlet in the web.xml file, you write this:

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app

PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app>

<servlet>

<servlet-name>controlServlet</servlet-name>

<servlet-class>com.jenkov.butterfly.ControlServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>controlServlet</servlet-name>

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

</web-app>

First you configure the servlet. This is done using the <servlet> element. Here you give the servlet a name, and writes the class name of the servlet.

Second, you map the servlet to a URL or URL pattern. This is done in the <servlet-mapping> element. In the above example, all URL's ending in .html are sent to the servlet.

Other possible servlet URL mappings are:

/myServlet

/myServlet.do

/myServlet\*

The \* is a wild card, meaning any text. As you can see, you can either map a servlet to a single, specific URL, or to a pattern of URL's, using a wild card (\*). What you will use depends on what the servlet does.

## Servlet Init Parameters

You can pass parameters to a servlet from the web.xml file. The init parameters of a servlet can only be accessed by that servlet. Here is how you configure them in the web.xml file:

<servlet>

<servlet-name>controlServlet</servlet-name>

<servlet-class>com.jenkov.butterfly.ControlServlet</servlet-class>

**<init-param>**

**<param-name>myParam</param-name>**

**<param-value>paramValue</param-value>**

**</init-param>**

</servlet>

Here is how you read the init parameters from inside your servlet - in the servlets init() method:

public class SimpleServlet extends GenericServlet {

protected String myParam = null;

public void init(ServletConfig servletConfig) throws ServletException{

**this.myParam = servletConfig.getInitParameter("myParam");**

}

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

response.getWriter().write("<html><body>myParam = " +

this.myParam + "</body></html>");

}

}

A servlets init() method is called when the servlet container loads the servlet for the first time. No one can access the servlet until the servlet has been loaded, and the init() method has been called successfully.

## Servlet Load-on-Startup

The <servlet> element has a subelement called <load-on-startup> which you can use to control when the servlet container should load the servlet. If you do not specify a <load-on-startup> element, the servlet container will typically load your servlet when the first request arrives for it.

By setting a <load-on-startup> element, you can tell the servlet container to load the servlet as soon as the servlet container starts. Remember, the servlets init() method is called when the servlet is loaded.

Here is an example <load-on-startup> configuration:

<servlet>

<servlet-name>controlServlet</servlet-name>

<servlet-class>com.jenkov.webui.ControlServlet</servlet-class>

<init-param><param-name>container.script.static</param-name>

<param-value>/WEB-INF/container.script</param-value>

</init-param>

**<load-on-startup>1</load-on-startup>**

</servlet>

The number inside the **<load-on-startup>1</load-on-startup>** element tells the servlet container in what sequence the servlets should be loaded. The lower numbers are loaded first. If the value is negative, or unspecified, the servlet container can load the servlet at any time.

## Context Parameters

You can also set some context parameters which can be read from all servlets in your application. Here is how you configure a context parameter:

<context-param>

<param-name>myParam</param-name>

<param-value>the value</param-value>

</context-param>

Here is how you access the parameter from inside an HttpServlet subclass:

String myContextParam =

request.getSession()

.getServletContext()

.getInitParameter("myParam");

**Servlet Concurrency**

A Java servlet container / web server is typically multithreaded. That means, that multiple requests to the same servlet may be executed at the same time. Therefore, you need to take concurrency into consideration when you implement your servlet.

I am not going to describe Java concurrency in great detail here. If you are interested in this topic, read my tutorial on [Java Concurrency](http://tutorials.jenkov.com/java-concurrency/index.html).

To make sure that a servlet is thread safe, there are a few basic rules of thumb you must follow:

1. Your servlet service() method should not access any member variables, unless these member variables are thread safe themselves.
2. Your servlet service() should not reassign member variables, as this may affect other threads executing inside the service() method. If you really, really need to reassign a member variable, make sure this is done inside a synchronized block.
3. Rule 1 and 2 also counts for static variables.
4. Local variables are always thread safe. Keep in mind though, that the object a local variable points to, may not be so. If the object was instantiated inside the method, and never escapes, there will be no problem. On the other hand, a local variable pointing to some shared object, may still cause problems. Just because you assign a shared object to a local reference, does not mean that object automatically becomes thread safe.

The request and response objects are of course thread safe to use. A new instance of these are created for every request into your servlet, and thus for every thread executing in your servlet.

Here is a diagram which illustrates the servlet concurrency rules / issues mentioned above. The red boxes represent state (variables) that your servlet's service() method should be careful about accessing.

|  |
| --- |
| Java Servlet Concurrency Rules |
| **Java Servlet Concurrency Rules.** |

## Other Shared Resources

Of course it is not only the member variables and static variables inside the servlet class itself, that you need to be careful about accessing. Static variables in any other class which are accessed by your servlet, must also be thread safe. The same is true for member variables of any shread objects accessed by your servlet.

## Code Example

Here is a code example that shows you some of the rules I have been talking about in this text.

public class SimpleHttpServlet extends HttpServlet {

// Not thread safe, static.

**protected static List list = new ArrayList();**

// Not thread safe

**protected Map map = new HashMap();**

// Thread safe to access object, not thread safe to reassign variable.

**protected Map map = new ConcurrentHashMap();**

// Thread safe to access object (immutable), not thread safe to reassign variable.

**protected String aString = "a string value";**

protected void doGet( HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

// Not thread safe, unless the singleton is 100% thread safe.

**SomeClass.getSomeStaticSingleton();**

// Thread safe, locally instantiated, and never escapes method.

**Set set = new HashSet();**

}

}

**7.How the servlet is loaded?**

A servlet can be loaded when:

* First request is made.
* Server starts up (auto-load).
* There is only a single instance which answers all requests concurrently. This saves memory and allows a Servlet to easily manage persistent data.
* Administrator manually loads.

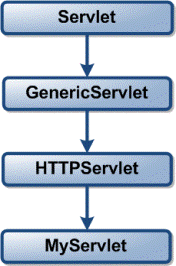
**8.How a Servlet is unloaded?**

A servlet is unloaded when:

* Server shuts down.
* Administrator manually unloads.

**9.What is Servlet interface?**

The central abstraction in the Servlet API is the Servlet interface. All servlets implement this interface, either directly or , more commonly by extending a class that implements it.



*Note: Most Servlets, however, extend one of the standard implementations of that interface, namely* javax.servlet.GenericServlet and javax.servlet.http.HttpServlet.

**10.What is the GenericServlet class?**

GenericServlet is an abstract class that implements the Servlet interface and the ServletConfig interface. In addition to the methods declared in these two interfaces, this class also provides simple versions of the lifecycle methods init and destroy, and implements the log method declared in the ServletContext interface.   
*Note: This class is known as generic servlet, since it is not specific to any protocol.*

**11.What's the difference between GenericServlet and HttpServlet?**

|  |  |
| --- | --- |
| **GenericServlet** | **HttpServlet** |
| The GenericServlet is an abstract class that is extended by HttpServlet to provide HTTP protocol-specific methods. | An abstract class that simplifies writing HTTP servlets. It extends the GenericServlet base class and provides an framework for handling the HTTP protocol. |
| The GenericServlet does not include protocol-specific methods for handling request parameters, cookies, sessions and setting response headers. | The HttpServlet subclass passes generic service method requests to the relevant doGet() or doPost() method. |
| GenericServlet is not specific to any protocol. | HttpServlet only supports HTTP and HTTPS protocol. |

|  |
| --- |
|  |

**12.Why is HttpServlet declared abstract?**

The HttpServlet class is declared abstract because the default implementations of the main service methods do nothing and must be overridden. This is a convenience implementation of the Servlet interface, which means that developers do not need to implement all service methods. If your servlet is required to handle doGet() requests for example, there is no need to write a doPost() method too.

**13.Can servlet have a constructor ?**

One can definitely have constructor in servlet.Even you can use the constrctor in servlet for initialization purpose,but this type of approch is not so common. You can perform common operations with the constructor as you normally do.The only thing is that you cannot call that constructor explicitly by the new keyword as we normally do.In the case of servlet, servlet container is responsible for instantiating the servlet, so the constructor is also called by servlet container only.

**14.What are the types of protocols supported by HttpServlet ?**

It extends the GenericServlet base class and provides a framework for handling the HTTP protocol. So, HttpServlet only supports HTTP and HTTPS protocol.

**15.What is the difference between doGet() and doPost()?**

|  |  |  |
| --- | --- | --- |
| **#** | **doGet()** | **doPost()** |
| 1 | In doGet() the parameters are appended to the URL and sent along with header information. | In doPost(), on the other hand will (typically) send the information through a socket back to the webserver and it won't show up in the URL bar. |
| 2 | The amount of information you can send back using a GET is restricted as URLs can only be 1024 characters. | You can send much more information to the server this way - and it's not restricted to textual data either. It is possible to send files and even binary data such as serialized Java objects! |
| 3 | doGet() is a request for information; it does not (or should not) change anything on the server. (doGet() should be idempotent) | doPost() provides information (such as placing an order for merchandise) that the server is expected to remember |
| 4 | Parameters are not encrypted | Parameters are encrypted |
| 5 | doGet() is faster if we set the response content length since the same connection is used. Thus increasing the performance | doPost() is generally used to update or post some information to the server.doPost is slower compared to doGet since doPost does not write the content length |
| 6 | doGet() should be idempotent. i.e. doget should be able to be repeated safely many times | This method does not need to be idempotent. Operations requested through POST can have side effects for which the user can be held accountable. |
| 7 | doGet() should be safe without any side effects for which user is held responsible | This method does not need to be either safe |
| 8 | It allows bookmarks. | It disallows bookmarks. |

**16.When to use doGet() and when doPost()?**

Always prefer to use GET (As because GET is faster than POST), except mentioned in the following reason:

* If data is sensitive
* Data is greater than 1024 characters
* If your application don't need bookmarks.

**17.How do I support both GET and POST from the same Servlet?**

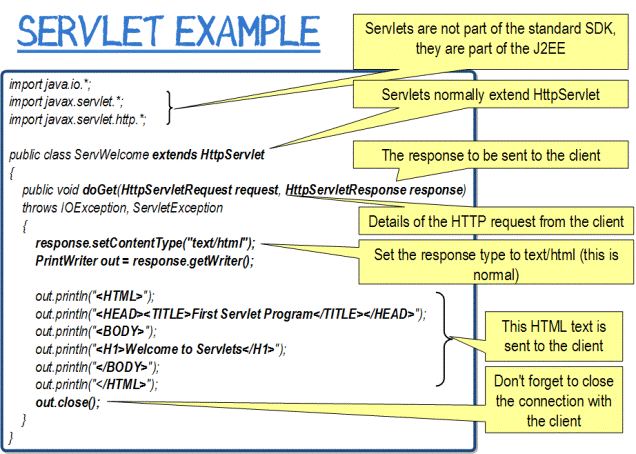
The easy way is, just support POST, then have your doGet method call your doPost method:

**public void doGet(HttpServletRequest request, HttpServletResponse response)  
                        throws ServletException, IOException  
{  
    doPost(request, response);     
}**

**18.Should I override the service() method?**

We never override the service method, since the HTTP Servlets have already taken care of it . The default service function invokes the doXXX() method corresponding to the method of the HTTP request.For example, if the HTTP request method is GET, doGet() method is called by default. A servlet should override the doXXX() method for the HTTP methods that servlet supports. Because HTTP service method check the request method and calls the appropriate handler method, it is not necessary to override the service method itself. Only override the appropriate doXXX() method.

**19.How the typical servlet code look like ?**



**20.What is a servlet context object?**

A servlet context object contains the information about the Web application of which the servlet is a part. It also provides access to the resources common to all the servlets in the application. Each Web application in a container has a single servlet context associated with it.

**21.What are the differences between the ServletConfig interface and the ServletContext interface?**

|  |  |
| --- | --- |
| **ServletConfig** | **ServletContext** |
| The ServletConfig interface is implemented by the servlet container in order to pass configuration information to a servlet. The server passes an object that implements the ServletConfig interface to the servlet's init() method. | A ServletContext defines a set of methods that a servlet uses to communicate with its servlet container. |
| There is one ServletConfig parameter per servlet. | There is one ServletContext for the entire webapp and all the servlets in a webapp share it. |
| The param-value pairs for ServletConfig object are specified in the <init-param> within the <servlet> tags in the web.xml file | The param-value pairs for ServletContext object are specified in the <context-param> tags in the web.xml file. |

**22.What's the difference between forward() and sendRedirect() methods?**

|  |  |
| --- | --- |
| **forward()** | **sendRedirect()** |
| A forward is performed internally by the servlet. | A redirect is a two step process, where the web application instructs the browser to fetch a second URL, which differs from the original. |
| The  browser is completely unaware that it has taken place, so its original URL remains intact. | The browser, in this case, is doing the work and knows that it's making a new request. |
| Any browser reload of the resulting page will simple repeat the original request, with the original URL | A browser reloads of the second URL ,will not repeat the original request, but will rather fetch the second URL. |
| Both resources must be part of the same context (Some containers make provisions for cross-context communication but this tends not to be very portable) | This method can be used to redirect users to resources that are not part of the current context, or even in the same domain. |
| Since both resources are part of same context, the original request context is retained | Because this involves a new request, the previous request scope objects, with all of its parameters and attributes are no longer available after a redirect. (Variables will need to be passed by via the session object). |
| Forward is marginally faster than redirect. | redirect is marginally slower than a forward, since it requires two browser requests, not one. |

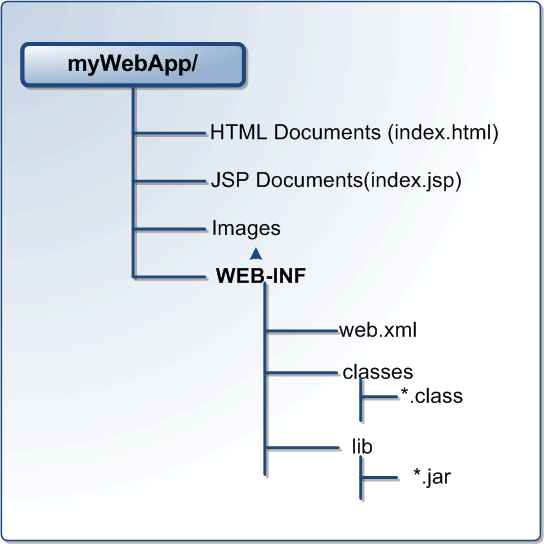
**23.What is the difference between the include() and forward() methods?**

|  |  |
| --- | --- |
| **include()** | **forward()** |
| The RequestDispatcher include() method inserts the the contents of the specified resource directly in the flow of the servlet response, as if it were part of the calling servlet. | The RequestDispatcher forward() method is used to show a different resource in place of the servlet that was originally called. |
| If you include a servlet or JSP document, the included resource must not attempt to change the response status code or HTTP headers, any such request will be ignored. | The forwarded resource may be another servlet, JSP or static HTML document, but the response is issued under the same URL that was originally requested. In other words, it is not the same as a redirection. |
| The include() method is often used to include common "boilerplate" text or template markup that may be included by many servlets. | The forward() method is often used where a servlet is taking a controller role; processing some input and deciding the outcome by returning a particular response page. |

**24.What's the use of the servlet wrapper classes??**

The HttpServletRequestWrapper and HttpServletResponseWrapper classes are designed to make it easy for developers to create custom implementations of the servlet request and response types. The classes are constructed with the standard HttpServletRequest and HttpServletResponse instances respectively and their default behaviour is to pass all method calls directly to the underlying objects.

**25.What is the directory structure of a WAR file?**



**26.What is a deployment descriptor?**

A deployment descriptor is an XML document with an .xml extension. It defines a component's deployment settings. It declares transaction attributes and security authorization for an enterprise bean. The information provided by a deployment descriptor is declarative and therefore it can be modified without changing the source code of a bean.  
The JavaEE server reads the deployment descriptor at run time and acts upon the component accordingly.

**27.What is the difference between the getRequestDispatcher(String path) method of javax.servlet.ServletRequest interface and javax.servlet.ServletContext interface?**

|  |  |
| --- | --- |
| **ServletRequest.getRequestDispatcher(String path)** | **ServletContext.getRequestDispatcher(String path)** |
| The getRequestDispatcher(String path) method of javax.servlet.ServletRequest interface accepts parameter the path to the resource to be included or forwarded to, which can be relative to the request of the calling servlet. If the path begins with a “/” it is interpreted as relative to the current context root. | The getRequestDispatcher(String path) method of javax.servlet.ServletContext interface cannot accept relative paths. All path must start with a “/” and are   interpreted as relative to current context root. |

**28.What is preinitialization of a servlet?**

A container does not initialize the servlets as soon as it starts up, it initializes a servlet when it receives a request for that servlet first time. This is called lazy loading. The servlet specification defines the element, which can be specified in the deployment descriptor to make the servlet container load and initialize the servlet as soon as it starts up. The process of loading a servlet before any request comes in is called preloading or preinitializing a servlet.

**29.What is the <load-on-startup> element?**

The <load-on-startup> element of a deployment descriptor is used to load a servlet file when the server starts instead of waiting for the first request. It is also used to specify the order in which the files are to be loaded. The <load-on-startup> element is written in the deployment descriptor as follows:

<servlet>  
   <servlet-name>ServletName</servlet-name>  
   <servlet-class>ClassName</servlet-class>  
   <load-on-startup>1</load-on-startup>  
</servlet>

*Note: The container loads the servlets in the order specified in the <load-on-startup> element.*

**31.What is Session Tracking?**

Session tracking is a mechanism that servlets use to maintain state about a series of requests from the same user (that is, requests originating from the same browser) across some period of time.

**32.What is the need of Session Tracking in web application?**

HTTP is a stateless protocol i.e., every request is treated as new request. For web applications to be more realistic they have to retain information across multiple requests. Such information which is part of the application is reffered as "state". To keep track of this state we need session tracking.   
  
*Typical example:* Putting things one at a time into a shopping cart, then checking out--each page request must somehow be associated with previous requests.

**33.What are the types of Session Tracking ?**

Sessions need to work with all web browsers and take into account the users security preferences. Therefore there are a variety of ways to send and receive the identifier:

* **URL rewriting :** URL rewriting is a method of session tracking in which some extra data (session ID) is appended at the end of each URL. This extra data identifies the session. The server can associate this session identifier with the data it has stored about that session. This method is used with browsers that do not support cookies or where the user has disabled the cookies.
* **Hidden Form Fields :** Similar to URL rewriting. The server embeds new hidden fields in every dynamically generated form page for the client. When the client submits the form to the server the hidden fields identify the client.
* **Cookies :** Cookie is a small amount of information sent by a servlet to a Web browser. Saved by the browser, and later sent back to the server in subsequent requests. A cookie has a name, a single value, and optional attributes. A cookie's value can uniquely identify a client.
* **Secure Socket Layer (SSL) Sessions :** Web browsers that support Secure Socket Layer communication can use SSL's support via HTTPS for generating a unique session key as part of the encrypted conversation.

**34.How do I use cookies to store session state on the client?**

In a servlet, the HttpServletResponse and HttpServletRequest objects passed to method HttpServlet.service() can be used to create cookies on the client and use cookie information transmitted during client requests. JSPs can also use cookies, in scriptlet code or, preferably, from within custom tag code.

* To set a cookie on the client, use the addCookie() method in class HttpServletResponse. Multiple cookies may be set for the same request, and a single cookie name may have multiple values.
* To get all of the cookies associated with a single HTTP request, use the getCookies() method of class HttpServletRequest

**35.What are some advantages of storing session state in cookies?**

* Cookies are usually persistent, so for low-security sites, user data that needs to be stored long-term (such as a user ID, historical information, etc.) can be maintained easily with no server interaction.
* For small- and medium-sized session data, the entire session data (instead of just the session ID) can be kept in the cookie.

**36.What are some disadvantages of storing session state in cookies?**

* Cookies are controlled by programming a low-level API, which is more difficult to implement than some other approaches.
* All data for a session are kept on the client. Corruption, expiration or purging of cookie files can all result in incomplete, inconsistent, or missing information.
* Cookies may not be available for many reasons: the user may have disabled them, the browser version may not support them, the browser may be behind a firewall that filters cookies, and so on. Servlets and JSP pages that rely exclusively on cookies for client-side session state will not operate properly for all clients. Using cookies, and then switching to an alternate client-side session state strategy in cases where cookies aren't available, complicates development and maintenance.
* Browser instances share cookies, so users cannot have multiple simultaneous sessions.
* Cookie-based solutions work only for HTTP clients. This is because cookies are a feature of the HTTP protocol. Notice that the while package javax.servlet.http supports session management (via class HttpSession), package javax.servlet has no such support.

**37.What is URL rewriting?**

URL rewriting is a method of session tracking in which some extra data is appended at the end of each URL. This extra data identifies the session. The server can associate this session identifier with the data it has stored about that session.

Every URL on the page must be encoded using method HttpServletResponse.encodeURL(). Each time a URL is output, the servlet passes the URL to encodeURL(), which encodes session ID in the URL if the browser isn't accepting cookies, or if the session tracking is turned off.  
E.g., http://abc/path/index.jsp;jsessionid=123465hfhs

**Advantages**

* URL rewriting works just about everywhere, especially when cookies are turned off.
* Multiple simultaneous sessions are possible for a single user. Session information is local to each browser instance, since it's stored in URLs in each page being displayed. This scheme isn't foolproof, though, since users can start a new browser instance using a URL for an active session, and confuse the server by interacting with the same session through two instances.
* Entirely static pages cannot be used with URL rewriting, since every link must be dynamically written with the session state. It is possible to combine static and dynamic content, using (for example) templating or server-side includes. This limitation is also a barrier to integrating legacy web pages with newer, servlet-based pages.

**DisAdvantages**

* Every URL on a page which needs the session information must be rewritten each time a page is served. Not only is this expensive computationally, but it can greatly increase communication overhead.
* URL rewriting limits the client's interaction with the server to HTTP GETs, which can result in awkward restrictions on the page.
* URL rewriting does not work well with JSP technology.
* If a client workstation crashes, all of the URLs (and therefore all of the data for that session) are lost.

**38.How can an existing session be invalidated?**

An existing session can be invalidated in the following two ways:

* Setting timeout in the deployment descriptor: This can be done by specifying timeout between the <session-timeout>tags as follows:

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

This will set the time for session timeout to be ten minutes.

* Setting timeout programmatically: This will set the timeout for a specific session. The syntax for setting the timeout programmatically is as follows:

public void setMaxInactiveInterval(int interval)

The setMaxInactiveInterval() method sets the maximum time in seconds before a session becomes invalid.   
Note :Setting the inactive period a*s negative(-1), makes the container stop tracking session, i.e, session never expires.*

**39.How can the session in Servlet can be destroyed?**

An existing session can be destroyed in the following two ways:

* Programatically : Using session.invalidate() method, which makes the container abonden the session on which the method is called.
* When the server itself is shutdown.

**40.A client sends requests to two different web components. Both of the components access the session. Will they end up using the same session object or different session ?**

Creates only one session i.e., they end up with using same session .

Sessions is specific to the client but not the web components. And there is a 1-1 mapping between client and a session.

**41.What is servlet lazy loading?**

* A container doesnot initialize the servlets ass soon as it starts up, it initializes a servlet when it receives a request for that servlet first time. This is called lazy loading.
* The servlet specification defines the <load-on-startup> element, which can be specified in the deployment descriptor to make the servlet container load and initialize the servlet as soon as it starts up.
* The process of loading a servlet before any request comes in is called preloading or preinitializing a servlet.

**42.What is Servlet Chaining?**

Servlet Chaining is a method where the output of one servlet is piped into a second servlet. The output of the second servlet could be piped into a third servlet, and so on. The last servlet in the chain returns the output to the Web browser.

**43.How are filters?**

Filters are Java components that are used to intercept an incoming request to a Web resource and a response sent back from the resource. It is used to abstract any useful information contained in the request or response. Some of the important functions performed by filters are as follows:

* Security checks
* Modifying the request or response
* Data compression
* Logging and auditing
* Response compression

Filters are configured in the deployment descriptor of a Web application. Hence, a user is not required to recompile anything to change the input or output of the Web application.

**SendRedirect ():**

This method is used to redirect client request to some other location for further processing **,**the new location is available on different server or different context.our web container handle this and transfer the request using  browser ,and this request is **visible in browser as a new request**. Some time this is also called as client side redirect.

**Forward():**

This method is used to pass the request to another resource for further processing **within the same server**, another resource could be any servlet, jsp page any kind of file.This process is taken care by web container when we call forward method request is sent to another resource **without the client being informed,** which resource will handle the request it has been mention on requestDispatcher object which we can get by two ways either using ServletContext or Request. This is also called server side redirect. RequestDispatcher rd = request.getRequestDispatcher("pathToResource");  
  rd.forward(request, response);