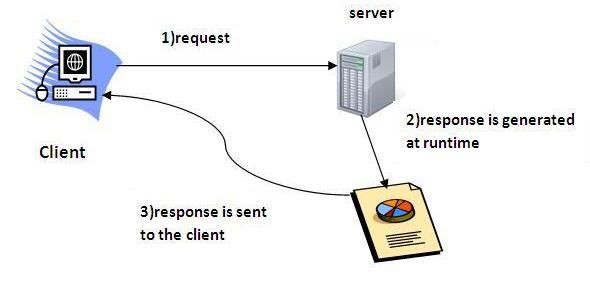
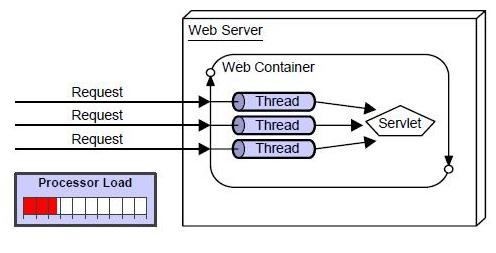
# SERVLET

Servlet can be described in many ways, depending on the context.

* Servlet is a technology i.e. used to create web application.
* Servlet is an API that provides many interfaces and classes including documentations.
* Servlet is an interface that must be implemented for creating any servlet.
* Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
* Servlet is a web component that is deployed on the server to create dynamic web page.



1.1 Advantage of Servlet



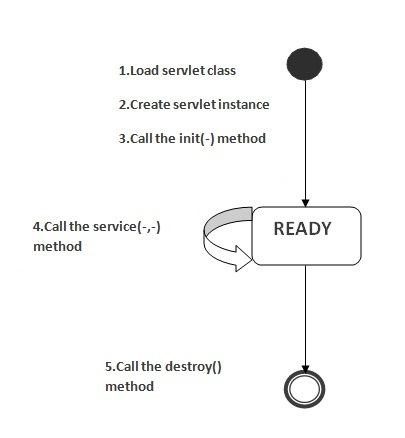
There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the servlet. Threads have a lot of benefits over the Processes such as they share a common memory area, lighweight, cost of communication between the threads are low. The basic benefits of servlet are as follows:

1. **better performance:** because it creates a thread for each request not process.
2. **Portability:** because it uses java language.
3. **Robust:** Servlets are managed by JVM so no need to worry about momory leak, garbage collection etc.
4. **Secure:** because it uses java language.

# 2. LIFE CYCLE OF A SERVLET (SERVLET LIFE CYCLE)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

## 1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

## 2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

## 3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

1. **public** **void** init(ServletConfig config) **throws** ServletException

## 4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

1. **public** **void** service(ServletRequest request, ServletResponse response)
2. **throws** ServletException, IOException

## 5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

|  |  |
| --- | --- |
| 1. | **public** **void** destroy() |

# 3. SERVLET TERMINOLOGY

There are some key points that must be known by the servlet programmer like server, container, get request, post request etc. Let's first discuss these points before starting the servlet technology.

The basic **terminology used in servlet** are given below:

1. HTTP
2. HTTP Request Types
3. Difference between Get and Post method
4. Container
5. Server and Difference between web server and application server
6. Content Type
7. Introduction of XML
8. Deployment

# 4.GET AND POST

There are many differences between the Get and Post request. Let's see these differences:

|  |  |
| --- | --- |
| **GET** | **POST** |
| 1) In case of Get request, only **limited amount of data**can be sent because data is sent in header. | In case of post request, **large amount of data** can be sent because data is sent in body. |
|  |  |
| 2) Get request is **not secured** because data is exposed in URL bar. | Post request is **secured** because data is not exposed in URL bar. |
|  |  |
| 3) Get request **can be bookmarked** | Post request **cannot be** bookmarked |
|  |  |
| 4) Get request is **idempotent**. It means second request will be ignored until response of first request is delivered. | Post request is **non-idempotent** |
|  |  |
| 5) Get request is **more efficient** and used more than Post | Post request is **less efficient** and used less than get. |

## Anatomy of Get Request

As we know that data is sent in request header in case of get request. It is the default request type. Let's see what informations are sent to the server.



## Anatomy of Post Request

As we know, in case of post request original data is sent in message body. Let's see how informations are passed to the server in case of post request.



# 5.SERVLET API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api. The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol. The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

# 6.SERVLET INTERFACE

**Servlet interface** provides common behaviour to all the servlets. Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

## 6.1 METHODS OF SERVLET INTERFACE

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **public void init(ServletConfig config)** | initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once. |
|  |  |
| **public void service(ServletRequest request,ServletResponse response)** | provides response for the incoming request. It is invoked at each request by the web container. |
| **public void destroy()** | is invoked only once and indicates that servlet is being destroyed. |
|  |  |
| **public ServletConfig getServletConfig()** | returns the object of ServletConfig. |
|  |  |
| **public String getServletInfo()** | returns information about servlet such as writer, copyright, version etc. |

**EXAMPLE**

*File: First.java*

|  |
| --- |
| 1. **import** java.io.\*; 2. **import** javax.servlet.\*;  3.  4. **public** **class** First **implements** Servlet{ 5. ServletConfig config=**null**;  6.   1. **public** **void** init(ServletConfig config){ 2. **this**.config=config; |
| 1. System.out.println("servlet is initialized"); 2. } 11. 3. **public** **void** service(ServletRequest req,ServletResponse res) 4. **throws** IOException,ServletException{   14.  15. res.setContentType("text/html");  16.  PrintWriter out=res.getWriter();  out.print("<html><body>");  out.print("<b>hello simple servlet</b>");  out.print("</body></html>");  }  **public** **void** destroy(){System.out.println("servlet is destroyed");}  **public** ServletConfig getServletConfig(){**return** config;}  **public** String getServletInfo(){**return** "copyright 2007-1010";} 25. } |

# 7.GENERIC SERVLET

**GenericServlet** class implements **Servlet**,**ServletConfig** and **Serializable** interfaces. It provides the implementaion of all the methods of these interfaces except the service method. GenericServlet class can handle any type of request so it is protocol-independent. You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

## 7.1.METHODS OF GENERIC SERVLET CLASS

There are many methods in GenericServlet class. They are as follows:

1. **public void init(ServletConfig config)** is used to initialize the servlet.
2. **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
3. **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
4. **public ServletConfig getServletConfig()** returns the object of ServletConfig.
5. **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
6. **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)
7. **public ServletContext getServletContext()** returns the object of ServletContext.
8. **public String getInitParameter(String name)** returns the parameter value for the given parameter name.
9. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
10. **public String getServletName()** returns the name of the servlet object.
11. **public void log(String msg)** writes the given message in the servlet log file.
12. **public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace.

**EXAMPLE**

*File: First.java*

1. **import** java.io.\*; 2. **import** javax.servlet.\*;

3.

1. **public** **class** First **extends** GenericServlet{
2. **public** **void** service(ServletRequest req,ServletResponse res)
3. **throws** IOException,ServletException{
4. res.setContentType("text/html");
5. PrintWriter out=res.getWriter();
6. out.print("<html><body>");
7. out.print("<b>hello generic servlet</b>");
8. out.print("</body></html>");
9. }
10. }

# 8. HTTPSERVLET

The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

## 8.1.METHODS OF HTTPSERVLET CLASS

There are many methods in HttpServlet class. They are as follows:

1. **public void service(ServletRequest req,ServletResponse res)** dispatches the request to the protected service method by converting the request and response object into http type.
2. **protected void service(HttpServletRequest req, HttpServletResponse res)** receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.
3. **protected void doGet(HttpServletRequest req, HttpServletResponse res)** handles the GET request. It is invoked by the web container.
4. **protected void doPost(HttpServletRequest req, HttpServletResponse res)** handles the POST request. It is invoked by the web container.
5. **protected void doHead(HttpServletRequest req, HttpServletResponse res)** handles the HEAD request. It is invoked by the web container.
6. **protected void doOptions(HttpServletRequest req, HttpServletResponse res)** handles the OPTIONS request. It is invoked by the web container.
7. **protected void doPut(HttpServletRequest req, HttpServletResponse res)** handles the PUT request. It is invoked by the web container.
8. **protected void doTrace(HttpServletRequest req, HttpServletResponse res)** handles the TRACE request. It is invoked by the web container.
9. **protected void doDelete(HttpServletRequest req, HttpServletResponse res)** handles the DELETE request. It is invoked by the web container.
10. **protected long getLastModified(HttpServletRequest req)** returns the time when HttpServletRequest was last modified since midnight January 1, 1970 GMT.

# 9.REQUESTDISPATCHER IN SERVLET

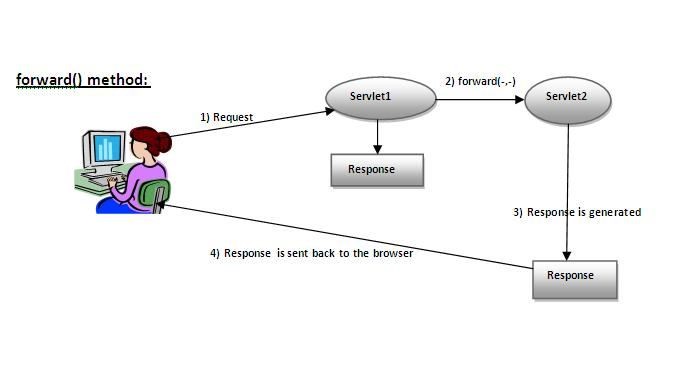
The RequestDispacher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp.This interface can also be used to include the content of antoher resource also. It is one of the way of servlet collaboration.

There are two methods defined in the RequestDispatcher interface.

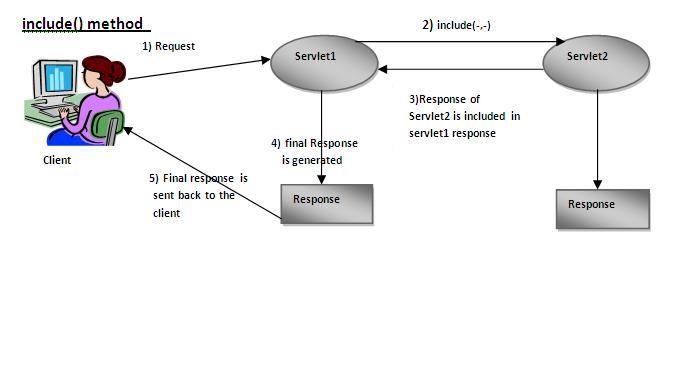
## Methods of RequestDispatcher interface

The RequestDispatcher interface provides two methods. They are:

1. **public void forward(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:**Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. **public void include(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:**Includes the content of a resource (servlet, JSP page, or HTML file) in the response.



As you see in the above figure, response of second servlet is sent to the client. Response of the first servlet is not displayed to the user.



As you can see in the above figure, response of second servlet is included in the response of the first servlet that is being sent to the client.

## How to get the object of RequestDispatcher

The getRequestDispatcher() method of ServletRequest interface returns the object of RequestDispatcher. Syntax:

***Syntax of getRequestDispatcher method***

1. public RequestDispatcher getRequestDispatcher(String resource);

***Example of using getRequestDispatcher method***

1. RequestDispatcher rd=request.getRequestDispatcher("servlet2");
2. //servlet2 is the url-pattern of the second servlet

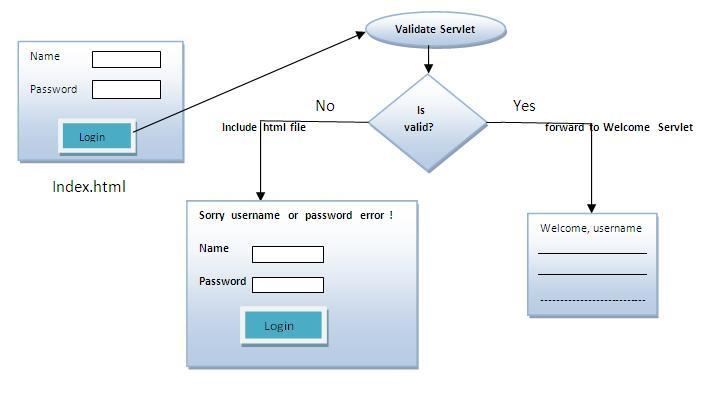
3.

4. rd.forward(request, response);//method may be include or forward

## EXAMPLE

In this example, we are validating the password entered by the user. If password is servlet, it will forward the request to the WelcomeServlet, otherwise will show an error message: sorry username or password error!. In this program, we are cheking for hardcoded information. But you can check it to the database also that we will see in the development chapter. In this example, we have created following files:

* **index.html file:** for getting input from the user.
* **Login.java file:** a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.
* **WelcomeServlet.java file:** a servlet class for displaying the welcome message.
* **web.xml file:** a deployment descriptor file that contains the information about the servlet.



**index.html**

|  |
| --- |
| 1. <form action="servlet1" method="post"> 2. Name:<input type="text" name="userName"/><br/> 3. Password:<input type="password" name="userPass"/><br/> 4. <input type="submit" value="login"/> 5. </form> |

**Login.java**

1. import java.io.\*;
2. import javax.servlet.\*;
3. import javax.servlet.http.\*;
4. public class Login extends HttpServlet {
5. public void doPost(HttpServletRequest request, HttpServletResponse response)
6. throws ServletException, IOException {
7. response.setContentType("text/html"); 8. PrintWriter out = response.getWriter();

9.

1. String n=request.getParameter("userName");
2. String p=request.getParameter("userPass");
3. if(p.equals("servlet"){
4. RequestDispatcher rd=request.getRequestDispatcher("servlet2");
5. rd.forward(request, response);
6. }
7. else{
8. out.print("Sorry UserName or Password Error!");
9. RequestDispatcher rd=request.getRequestDispatcher("/index.html");
10. rd.include(request, response);
11. }
12. }
13. }

**WelcomeServlet.java**

|  |
| --- |
| 1. import java.io.\*; 2. import javax.servlet.\*; 3. import javax.servlet.http.\*;   4.  5. public class WelcomeServlet extends HttpServlet { 6.   1. public void doPost(HttpServletRequest request, HttpServletResponse response) 2. throws ServletException, IOException {   9.   1. response.setContentType("text/html"); 2. PrintWriter out = response.getWriter();   12.   1. String n=request.getParameter("userName"); 2. out.print("Welcome "+n); 3. }   16.  17. } |

**web.xml**

1. <web-app>
2. <servlet>
3. <servlet-name>Login</servlet-name>
4. <servlet-class>Login</servlet-class>
5. </servlet>
6. <servlet>
7. <servlet-name>WelcomeServlet</servlet-name>
8. <servlet-class>WelcomeServlet</servlet-class>
9. </servlet>
10. <servlet-mapping>
11. <servlet-name>Login</servlet-name>
12. <url-pattern>/servlet1</url-pattern>
13. </servlet-mapping>
14. <servlet-mapping>
15. <servlet-name>WelcomeServlet</servlet-name>
16. <url-pattern>/servlet2</url-pattern>
17. </servlet-mapping>
18. <welcome-file-list>
19. <welcome-file>index.html</welcome-file>
20. </welcome-file-list>
21. </web-app>

# 9.SENDREDIRECT IN SERVLET

The **sendRedirect()** method of**HttpServletResponse** interface can be used to redirect response to another resource, it may be servlet, jsp or html file.

It accepts relative as well as absolute URL.

It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

# 10.DIFFERENCE BETWEEN FORWARD() AND SENDREDIRECT() METHOD

There are many differences between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse interface. They are given below:

|  |  |
| --- | --- |
| **forward() method** | **sendRedirect() method** |
| The forward() method works at server side. | The sendRedirect() method works at client side. |
|  |  |
| It sends the same request and response objects to another servlet. | It always sends a new request. |
|  |  |
| It can work within the server only. | It can be used within and outside the server. |
|  |  |
| Example: request.getRequestDispacher("servlet2").f orward(request,response); | Example:  response.sendRedirect("servlet2"); |

**Syntax of sendRedirect() method**

1. public void sendRedirect(String URL)throws IOException;

## EXAMPLE

**response.sendRedirect("http://www.javatpoint.com");**

*DemoServlet.java*

1. import java.io.\*;
2. import javax.servlet.\*;
3. import javax.servlet.http.\*;
4. public class DemoServlet extends HttpServlet{
5. public void doGet(HttpServletRequest req,HttpServletResponse res)
6. throws ServletException,IOException
7. {
8. res.setContentType("text/html");
9. PrintWriter pw=res.getWriter();
10. response.sendRedirect("http://www.google.com");
11. pw.close();
12. }}

Creating custom google search using sendRedirect

In this example, we are using sendRedirect method to send request to google server with the request data.

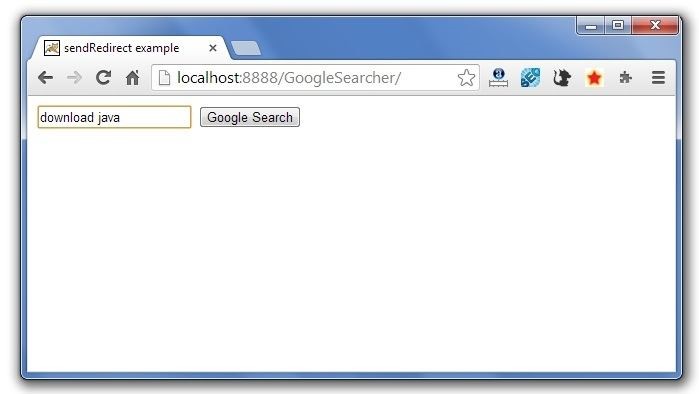
*index.html*

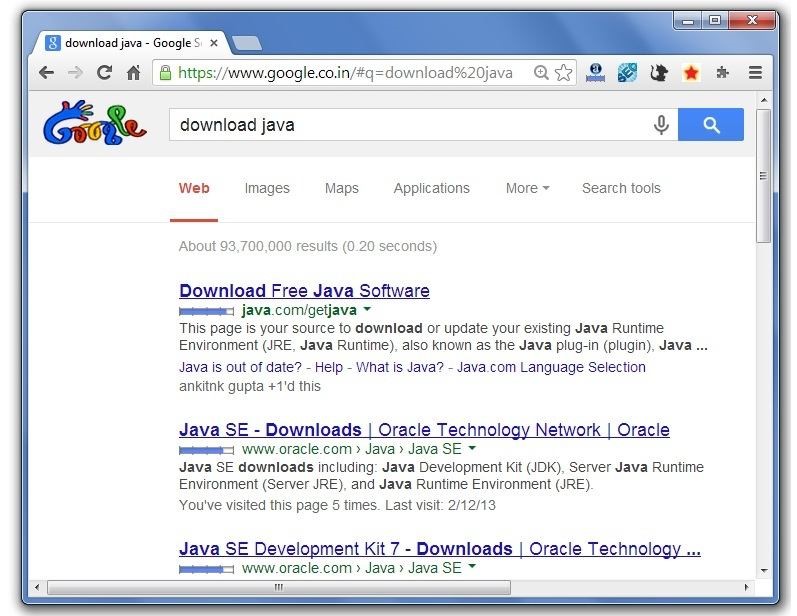
|  |  |
| --- | --- |
| 1. <!DOCTYPE html**>** 2. **<html>** 3. **<head>** 4. **<meta** charset="ISO-8859-1"**>** 5. **<title>**sendRedirect example**</title>** 6. **</head>** 7. **<body>** 8. **<form** action="MySearcher"**>** 9. **<input** type="text" name="name"**>** 10. **<input** type="submit" value="Google Search"**>** 11. **</form>**   12.   1. **</body>** 2. **</html>** | |
|  | *MySearcher.java* |

1. import java.io.IOException;
2. import javax.servlet.ServletException;
3. import javax.servlet.http.HttpServlet;
4. import javax.servlet.http.HttpServletRequest;

|  |  |
| --- | --- |
| 5. import javax.servlet.http.HttpServletResponse; 6.   1. public class MySearcher extends HttpServlet { 2. protected void doGet(HttpServletRequest request, HttpServletResponse response) 3. throws ServletException, IOException { 10. 4. String name=request.getParameter("name"); 5. response.sendRedirect("https://www.google.co.in/#q="+name); 6. } 7. } | |
|  |  |

***Output***





# 11.SERVLETCONFIG

An object of ServletConfig is created by the web container for each servlet. This object can be used to get configuration information from web.xml file. If the configuration information is modified from the web.xml file, we don't need to change the servlet. So it is easier to manage the web application if any specific content is modified from time to time.

## 11.1. ADVANTAGE OF SERVLETCONFIG

The core advantage of ServletConfig is that you don't need to edit the servlet file if information is modified from the web.xml file.

## 11.2. METHODS OF SERVLETCONFIG INTERFACE

1. **public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.
2. **public Enumeration getInitParameterNames():**Returns an enumeration of all the initialization parameter names.
3. **public String getServletName():**Returns the name of the servlet.
4. **public ServletContext getServletContext():**Returns an object of ServletContext.

1. **getServletConfig() method** of Servlet interface returns the object of ServletConfig.

***Syntax of getServletConfig() method***

1. public ServletConfig getServletConfig();

***Example of getServletConfig() method***

1. ServletConfig config=getServletConfig();
2. //Now we can call the methods of ServletConfig interface

## 11.3. SYNTAX TO PROVIDE THE INITIALIZATION PARAMETER FOR A SERVLET

The init-param sub-element of servlet is used to specify the initialization parameter for a servlet.

1. <web-app>
2. <servlet>
3. ......

4.

1. <init-param>
2. <param-name>parametername</param-name>
3. <param-value>parametervalue</param-value>
4. </init-param>
5. ......
6. </servlet>
7. </web-app>

## EXAMPLE

In this example, we are getting the one initialization parameter from the web.xml file and printing this information in the servlet.

**DemoServlet.java**

1. import java.io.\*;
2. import javax.servlet.\*;
3. import javax.servlet.http.\*;
4. public class DemoServlet extends HttpServlet {
5. public void doGet(HttpServletRequest request, HttpServletResponse response)
6. throws ServletException, IOException {
7. response.setContentType("text/html");
8. PrintWriter out = response.getWriter();
9. ServletConfig config=getServletConfig();
10. String driver=config.getInitParameter("driver");
11. out.print("Driver is: "+driver);
12. out.close();
13. }
14. }

**web.xml**

1. <web-app>
2. <servlet>
3. <servlet-name>DemoServlet</servlet-name>
4. <servlet-class>DemoServlet</servlet-class>
5. <init-param>
6. <param-name>driver</param-name>
7. <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
8. </init-param>
9. </servlet>
10. <servlet-mapping>
11. <servlet-name>DemoServlet</servlet-name>
12. <url-pattern>/servlet1</url-pattern>
13. </servlet-mapping>
14. </web-app>

## EXAMPLE

In this example, we are getting all the initialization parameter from the web.xml file and printing this information in the servlet.

**DemoServlet.java**

|  |
| --- |
| 1. import java.io.IOException; 2. import java.io.PrintWriter; 3. import java.util.Enumeration; 4. import javax.servlet.ServletConfig; 5. import javax.servlet.ServletException; 6. import javax.servlet.http.HttpServlet; 7. import javax.servlet.http.HttpServletRequest; 8. import javax.servlet.http.HttpServletResponse; 9. public class DemoServlet extends HttpServlet { 10. public void doGet(HttpServletRequest request, HttpServletResponse response) 11. throws ServletException, IOException { 12. response.setContentType("text/html"); 13. PrintWriter out = response.getWriter(); 14. ServletConfig config=getServletConfig(); 15. Enumeration<String> e=config.getInitParameterNames(); 16. String str=""; 16. while(e.hasMoreElements()){ 17. str=e.nextElement(); 18. out.print("<br>Name: "+str); 19. out.print(" value: "+config.getInitParameter(str)); 20. } 21. out.close(); 22. } 23. } |

**web.xml**

1. <web-app>
2. <servlet>
3. <servlet-name>DemoServlet</servlet-name>

|  |
| --- |
| 1. <servlet-class>DemoServlet</servlet-class> 2. <init-param> 3. <param-name>username</param-name> 4. <param-value>system</param-value> 5. </init-param> 6. <init-param> 7. <param-name>password</param-name> 8. <param-value>oracle</param-value> 9. </init-param>   13.  14. </servlet> 15.   1. <servlet-mapping> 2. <servlet-name>DemoServlet</servlet-name> 3. <url-pattern>/servlet1</url-pattern> 4. </servlet-mapping>   20.  21. </web-app> |

# 12.SERVLETCONTEXT

An object of ServletContext is created by the web container at time of deploying the project. This object can be used to get configuration information from web.xml file. There is only one ServletContext object per web application.

If any information is shared to many servlet, it is better to provide it from the web.xml file using the**<context-param>** element.

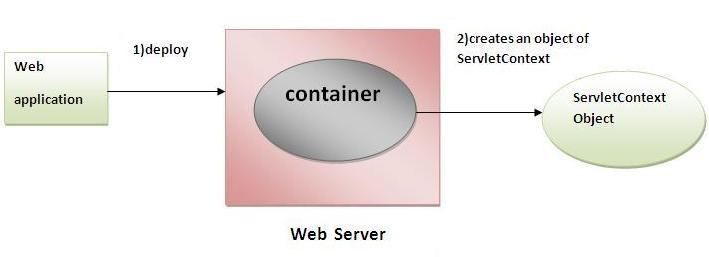
## 12.1.ADVANTAGE OF SERVLETCONTEXT

**Easy to maintain** if any information is shared to all the servlet, it is better to make it available for all the servlet. We provide this information from the web.xml file, so if the information is changed, we don't need to modify the servlet. Thus it removes maintenance problem.

## 12.2.USAGE OF SERVLETCONTEXT INTERFACE

There can be a lot of usage of ServletContext object. Some of them are as follows:

1. The object of ServletContext provides an interface between the container and servlet.
2. The ServletContext object can be used to get configuration information from the web.xml file.
3. The ServletContext object can be used to set, get or remove attribute from the web.xml file.
4. The ServletContext object can be used to provide inter-application communication.



## 12.3.METHODS OF SERVLETCONTEXT

There is given some commonly used methods of ServletContext interface.

1. **public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.
2. **public Enumeration getInitParameterNames():**Returns the names of the context's initialization parameters.
3. **public void setAttribute(String name,Object object):**sets the given object in the application scope.
4. **public Object getAttribute(String name):**Returns the attribute for the specified name.
5. **public Enumeration getInitParameterNames():**Returns the names of the context's initialization parameters as an Enumeration of String objects.
6. **public void removeAttribute(String name):**Removes the attribute with the given name from the servlet context.

### Get the object of ServletContext interface

1. **getServletContext() method** of ServletConfig interface returns the object of ServletContext.
2. **getServletContext() method** of GenericServlet class returns the object of ServletContext.

***Syntax of getServletContext() method***

1. public ServletContext getServletContext()

***Example of getServletContext() method***

1. //We can get the ServletContext object from ServletConfig object 2. ServletContext application=getServletConfig().getServletContext();

3.

1. //Another convenient way to get the ServletContext object
2. ServletContext application=getServletContext();

## 12.4.SYNTAX TO PROVIDE THE INITIALIZATION PARAMETER IN CONTEXT SCOPE

The **context-param** element, subelement of web-app, is used to define the initialization parameter in the application scope. The param-name and param-value are the sub-elements of the context-param. The param-name element defines parameter name and and paramvalue defines its value.

1. <web-app>
2. ......

3.

1. <context-param>
2. <param-name>parametername</param-name>
3. <param-value>parametervalue</param-value>
4. </context-param>
5. ......
6. </web-app>

## EXAMPLE

In this example, we are getting the initialization parameter from the web.xml file and printing the value of the initialization parameter. Notice that the object of ServletContext represents the application scope. So if we change the value of the parameter from the web.xml file, all the servlet classes will get the changed value. So we don't need to modify the servlet. So it is better to have the common information for most of the servlets in the web.xml file by context-param element. Let's see the simple example:

**DemoServlet.java**

|  |  |  |
| --- | --- | --- |
| 1. import java.io.\*; 2. import javax.servlet.\*; 3. import javax.servlet.http.\*; 4. public class DemoServlet extends HttpServlet{ 5. public void doGet(HttpServletRequest req,HttpServletResponse res) 6. throws ServletException,IOException 6. { 7. res.setContentType("text/html"); 8. PrintWriter pw=res.getWriter();   10.   1. //creating ServletContext object 2. ServletContext context=getServletContext();   13.   1. //Getting the value of the initialization parameter and printing it 2. String driverName=context.getInitParameter("dname"); 3. pw.println("driver name is="+driverName);   17.  18. pw.close(); 19.  20. }} | | |
|  | **web.xml** |  |

1. <web-app>

2.

1. <servlet>
2. <servlet-name>sonoojaiswal</servlet-name>
3. <servlet-class>DemoServlet</servlet-class>
4. </servlet>
5. <context-param>
6. <param-name>dname</param-name>
7. <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
8. </context-param>
9. <servlet-mapping>
10. <servlet-name>sonoojaiswal</servlet-name>
11. <url-pattern>/context</url-pattern>
12. </servlet-mapping>
13. </web-app>

## EXAMPLE OF SERVLETCONTEXT TO GET ALL THE INITIALIZATION PARAMETERS

In this example, we are getting all the initialization parameter from the web.xml file. For getting all the parameters, we have used the getInitParameterNames() method in the servlet class.

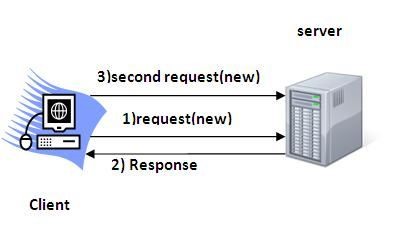
**DemoServlet.java**

|  |  |  |
| --- | --- | --- |
| 1. import java.io.\*; 2. import javax.servlet.\*; 3. import javax.servlet.http.\*;   4.  5.   1. public class DemoServlet extends HttpServlet{ 2. public void doGet(HttpServletRequest req,HttpServletResponse res) 8. throws ServletException,IOException 3. { 4. res.setContentType("text/html"); 5. PrintWriter out=res.getWriter(); 6. ServletContext context=getServletContext(); 7. Enumeration<String> e=context.getInitParameterNames(); 14. String str=""; 8. while(e.hasMoreElements()){ 9. str=e.nextElement(); | | |
| 1. out.print("<br> "+context.getInitParameter(str)); 2. } 3. }} | | |
|  | **web.xml** |  |

1. <web-app>
2. <servlet>
3. <servlet-name>sonoojaiswal</servlet-name>
4. <servlet-class>DemoServlet</servlet-class>
5. </servlet>
6. <context-param>
7. <param-name>dname</param-name>
8. <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
9. </context-param>
10. <context-param>
11. <param-name>username</param-name>
12. <param-value>system</param-value>
13. </context-param>
14. <context-param>
15. <param-name>password</param-name>
16. <param-value>oracle</param-value>
17. </context-param>
18. <servlet-mapping>
19. <servlet-name>sonoojaiswal</servlet-name>
20. <url-pattern>/context</url-pattern>
21. </servlet-mapping>
22. </web-app>

# 13.SESSIONS IN SERVLETS

**Session** simply means a particular interval of time. **Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet. Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user. HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



## 13.1. SESSION TRACKING TECHNIQUES

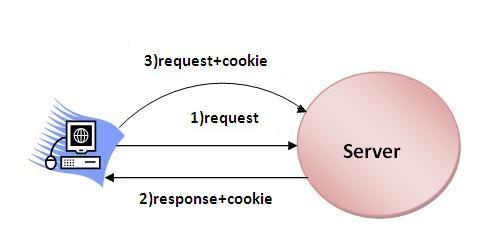
There are four techniques used in Session tracking:

1. **Cookies**
2. **Hidden Form Field**
3. **URL Rewriting**
4. **HttpSession**

## 13.2. COOKIES IN SERVLET

A **cookie** is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number. By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



## 13.3. TYPES OF COOKIE

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie

**13.3.1. NON-PERSISTENT COOKIE**

It is **valid for single session** only. It is removed each time when user closes the browser.

### 13.3.2. PERSISTENT COOKIE

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

## 13.4. ADVANTAGE OF COOKIES

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

## 13.5. DISADVANTAGE OF COOKIES

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

## 13.6. COOKIE CLASS

**javax.servlet.http.Cookie** class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

### Constructor of Cookie class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Cookie() | constructs a cookie. |
| Cookie(String name, String value) | constructs a cookie with a specified name and value. |

## 13.7. METHODS OF COOKIE CLASS

There are given some commonly used methods of the Cookie class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void setMaxAge(int expiry) | Sets the maximum age of the cookie in seconds. |
|  |  |
| public String getName() | Returns the name of the cookie. The name cannot be changed after creation. |
|  |  |
| public String getValue() | Returns the value of the cookie. |
|  |  |
| public void setName(String name) | changes the name of the cookie. |
|  |  |
| public void setValue(String value) | changes the value of the cookie. |

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[] getCookies():**method of HttpServletRequest interface is used to return all the cookies from the browser.

## 13.8. CREATE COOKIE

Let's see the simple code to create cookie.

1. Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object
2. response.addCookie(ck);//adding cookie in the response

## 13.9. DELETE COOKIE

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

1. Cookie ck=new Cookie("user","");//deleting value of cookie
2. ck.setMaxAge(0);//changing the maximum age to 0 seconds
3. response.addCookie(ck);//adding cookie in the response

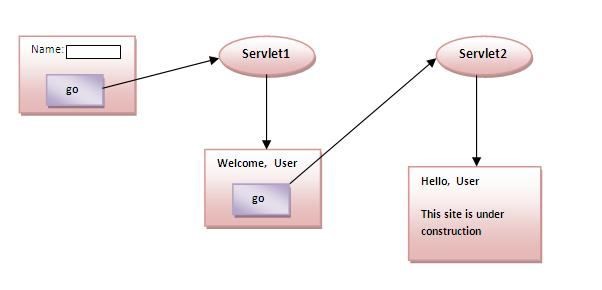
## 13.10. GET COOKIES

Let's see the simple code to get all the cookies.

1. Cookie ck[]=request.getCookies();
2. for(int i=0;i<ck.length;i++){
3. out.print("<br>"+ck[i].getName()+" "+ck[i].getValue());//printing name and value of cookie 4. }

## EXAMPLE

In this example, we are storing the name of the user in the cookie object and accessing it in another servlet. As we know well that session corresponds to the particular user. So if you access it from too many browsers with different values, you will get the different value.



### index.html

|  |
| --- |
| 1. <form action="servlet1" method="post"> 2. Name:<input type="text" name="userName"/><br/> 3. <input type="submit" value="go"/> 4. </form> |

### FirstServlet.java

1. import java.io.\*;
2. import javax.servlet.\*;
3. import javax.servlet.http.\*;
4. public class FirstServlet extends HttpServlet {
5. public void doPost(HttpServletRequest request, HttpServletResponse response){
6. try{
7. response.setContentType("text/html");
8. PrintWriter out = response.getWriter();
9. String n=request.getParameter("userName");
10. out.print("Welcome "+n);
11. Cookie ck=new Cookie("uname",n);//creating cookie object
12. response.addCookie(ck);//adding cookie in the response
13. //creating submit button
14. out.print("<form action='servlet2'>");
15. out.print("<input type='submit' value='go'>");
16. out.print("</form>");
17. out.close();
18. }catch(Exception e){System.out.println(e);}
19. }
20. }

### SecondServlet.java

1. import java.io.\*;
2. import javax.servlet.\*;
3. import javax.servlet.http.\*;
4. public class SecondServlet extends HttpServlet {
5. public void doPost(HttpServletRequest request, HttpServletResponse response){
6. try{
7. response.setContentType("text/html");
8. PrintWriter out = response.getWriter();
9. Cookie ck[]=request.getCookies();
10. out.print("Hello "+ck[0].getValue());
11. out.close();
12. }catch(Exception e){System.out.println(e);}
13. } 14. }

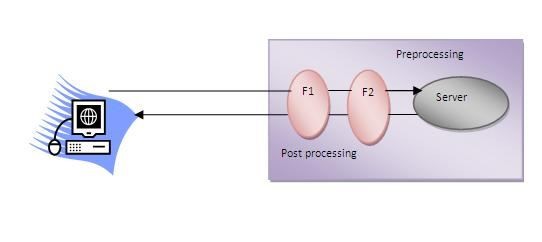
### web.xml

1. <web-app>
2. <servlet>
3. <servlet-name>s1</servlet-name>
4. <servlet-class>FirstServlet</servlet-class>
5. </servlet>
6. <servlet-mapping>
7. <servlet-name>s1</servlet-name>
8. <url-pattern>/servlet1</url-pattern>
9. </servlet-mapping>
10. <servlet>
11. <servlet-name>s2</servlet-name>
12. <servlet-class>SecondServlet</servlet-class>
13. </servlet>
14. <servlet-mapping>
15. <servlet-name>s2</servlet-name>
16. <url-pattern>/servlet2</url-pattern>
17. </servlet-mapping>
18. </web-app>

# 15.SERVLET FILTER

A **filter** is an object that is invoked at the preprocessing and postprocessing of a request. It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc. The **servlet filter is pluggable**, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the servlet.

So maintenance cost will be less.



## 15.1.USAGE OF FILTER

* recording all incoming requests
* logs the IP addresses of the computers from which the requests originate
* conversion
* data compression  encryption and decryption  input validation etc.

## 15.2.ADVANTAGE OF FILTER

1. Filter is pluggable.
2. One filter don't have dependency onto another resource.
3. Less Maintenance

## 15.3.FILTER API

Like servlet filter have its own API. The javax.servlet package contains the three interfaces of Filter API.

1. Filter
2. FilterChain
3. FilterConfig

### 1) Filter interface

For creating any filter, you must implement the Filter interface. Filter interface provides the life cycle methods for a filter.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void init(FilterConfig config) | init() method is invoked only once. It is used to initialize the filter. |
|  |  |
| public void doFilter(HttpServletRequest request,HttpServletResponse response,  FilterChain chain) | doFilter() method is invoked every time when user request to any resource, to which the filter is mapped.It is used to perform filtering tasks. |
|  |  |
| public void destroy() | This is invoked only once when filter is taken out of the service. |

### 2) FilterChain interface

The object of FilterChain is responsible to invoke the next filter or resource in the chain.This object is passed in the doFilter method of Filter interface.The FilterChain interface contains only one method:

1. **public void doFilter(HttpServletRequest request, HttpServletResponse response):** it passes the control to the next filter or resource.

## 15.4.DEFINE FILTER

We can define filter same as servlet. Let's see the elements of filter and filter-mapping.

1. <web-app>
2. <filter>
3. <filter-name>...</filter-name>
4. <filter-class>...</filter-class>
5. </filter>
6. <filter-mapping>
7. <filter-name>...</filter-name>
8. <url-pattern>...</url-pattern>
9. </filter-mapping>
10. </web-app>

For mapping filter we can use, either url-pattern or servlet-name. The url-pattern elements has an advantage over servlet-name element i.e. it can be applied on servlet, JSP or HTML.

## EXAMPLE

In this example, we are simply displaying information that filter is invoked automatically after the post processing of the request. **index.html**

1. <a href="servlet1">click here</a>

### MyFilter.java

1. import java.io.IOException;
2. import java.io.PrintWriter;
3. import javax.servlet.\*;
4. public class MyFilter implements Filter{
5. public void init(FilterConfig arg0) throws ServletException {}
6. public void doFilter(ServletRequest req, ServletResponse resp,
7. FilterChain chain) throws IOException, ServletException {
8. PrintWriter out=resp.getWriter();
9. out.print("filter is invoked before");
10. chain.doFilter(req, resp);//sends request to next resource
11. out.print("filter is invoked after");
12. }
13. public void destroy() {}
14. }

### HelloServlet.java

|  |  |  |
| --- | --- | --- |
| 1. import java.io.IOException; 2. import java.io.PrintWriter; 3. import javax.servlet.ServletException; 4. import javax.servlet.http.\*; 5. public class HelloServlet extends HttpServlet { 6. public void doGet(HttpServletRequest request, HttpServletResponse response) 7. throws ServletException, IOException { 8. response.setContentType("text/html"); 9. PrintWriter out = response.getWriter(); 10. out.print("<br>welcome to servlet<br>"); 11. } 12. } | | |
|  | **web.xml** |  |

For defining the filter, filter element of web-app must be defined just like servlet.

1. <web-app>
2. <servlet>
3. <servlet-name>s1</servlet-name>
4. <servlet-class>HelloServlet</servlet-class>
5. </servlet>
6. <servlet-mapping>
7. <servlet-name>s1</servlet-name>
8. <url-pattern>/servlet1</url-pattern>
9. </servlet-mapping>
10. <filter>
11. <filter-name>f1</filter-name>
12. <filter-class>MyFilter</filter-class>
13. </filter>
14. <filter-mapping>
15. <filter-name>f1</filter-name>
16. <url-pattern>/servlet1</url-pattern>
17. </filter-mapping>
18. </web-app>

Authentication Filter

We can perform authentication in filter. Here, we are going to check to password given by the user in filter class, if given password is admin, it will forward the request to the WelcomeAdmin servlet otherwise it will display error message.

Example of authenticating user using filter

Let's see the simple example of authenticating user using filter.

Here, we have created 4 files:

* + index.html
  + MyFilter.java
  + AdminServlet.java
  + web.xml **index.html**

|  |
| --- |
| 1. <form action="servlet1"> 2. Name:<input type="text" name="name"/><br/> 3. Password:<input type="password" name="password"/><br/> 4. <input type="submit" value="login"> 5. </form> |

**MyFilter.java**

1. **import** java.io.IOException;
2. **import** java.io.PrintWriter;
3. **import** javax.servlet.\*;
4. **public** **class** MyFilter **implements** Filter{
5. **public** **void** init(FilterConfig arg0) **throws** ServletException {}
6. **public** **void** doFilter(ServletRequest req, ServletResponse resp,
7. FilterChain chain) **throws** IOException, ServletException {
8. PrintWriter out=resp.getWriter();
9. String password=req.getParameter("password");
10. **if**(password.equals("admin")){
11. chain.doFilter(req, resp);//sends request to next resource
12. }
13. **else**{
14. out.print("username or password error!");
15. RequestDispatcher rd=req.getRequestDispatcher("index.html");
16. rd.include(req, resp);
17. }

18.

1. }
2. **public** **void** destroy() {} 21.

22. }

**AdminServlet.java**

1. **import** java.io.IOException;
2. **import** java.io.PrintWriter;
3. **import** javax.servlet.ServletException;
4. **import** javax.servlet.http.\*;
5. **public** **class** AdminServlet **extends** HttpServlet {
6. **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)
7. **throws** ServletException, IOException {
8. response.setContentType("text/html");
9. PrintWriter out = response.getWriter();
10. out.print("welcome ADMIN");
11. out.close();
12. }
13. }

**web.xml**

1. <web-app>
2. <servlet>
3. <servlet-name>AdminServlet</servlet-name>
4. <servlet-**class**>AdminServlet</servlet-**class**> 5. </servlet>

6.

1. <servlet-mapping>
2. <servlet-name>AdminServlet</servlet-name>
3. <url-pattern>/servlet1</url-pattern>
4. </servlet-mapping>

11.

1. <filter>
2. <filter-name>f1</filter-name>
3. <filter-**class**>MyFilter</filter-**class**>
4. </filter>
5. <filter-mapping>
6. <filter-name>f1</filter-name>
7. <url-pattern>/servlet1</url-pattern>
8. </filter-mapping>

20.

21. </web-app>

### 15.5. FILTERCONFIG

An object of FilterConfig is created by the web container. This object can be used to get the configuration information from the web.xml file.

Methods of FilterConfig interface

There are following 4 methods in the FilterConfig interface.

1. **public void init(FilterConfig config):** init() method is invoked only once it is used to initialize the filter.
2. **public String getInitParameter(String parameterName):** Returns the parameter value for the specified parameter name.
3. **public java.util.Enumeration getInitParameterNames():** Returns an enumeration containing all the parameter names.
4. **public ServletContext getServletContext():** Returns the ServletContext object.

## EXAMPLE

In this example, if you change the param-value to no, request will be forwarded to the servlet otherwise filter will create the response with the message: this page is underprocessing. Let's see the simple example of FilterConfig. Here, we have created 4 files:

* index.html
* MyFilter.java
* HelloServlet.java
* web.xml

**index.html**

1. <a href="servlet1">click here</a>

**MyFilter.java**

|  |
| --- |
| 1. **import** java.io.IOException; 2. **import** java.io.PrintWriter;  3.  4. **import** javax.servlet.\*;  5.  6. **public** **class** MyFilter **implements** Filter{ |
| 7. FilterConfig config;  8.   1. **public** **void** init(FilterConfig config) **throws** ServletException { 2. **this**.config=config; 11. } 12. 3. **public** **void** doFilter(ServletRequest req, ServletResponse resp, 4. FilterChain chain) **throws** IOException, ServletException {   15.  16. PrintWriter out=resp.getWriter();  17.  18. String s=config.getInitParameter("construction");  19.   1. **if**(s.equals("yes")){ 2. out.print("This page is under construction"); 3. } 4. **else**{ 5. chain.doFilter(req, resp);//sends request to next resource 6. } 26. 7. } 8. **public** **void** destroy() {} 29. } |

**HelloServlet.java**

|  |
| --- |
| 1. **import** java.io.IOException; 2. **import** java.io.PrintWriter;  3.  4. **import** javax.servlet.ServletException; 5. **import** javax.servlet.http.\*;  6.   1. **public** **class** HelloServlet **extends** HttpServlet { 2. **public** **void** doGet(HttpServletRequest request, HttpServletResponse response) 3. **throws** ServletException, IOException { 10. 4. response.setContentType("text/html"); 5. PrintWriter out = response.getWriter();   13.   1. out.print("<br>welcome to servlet<br>"); 2. 16. }   17. |

18. }

**web.xml**

|  |
| --- |
| 1. <web-app> 2.   1. <servlet> 2. <servlet-name>HelloServlet</servlet-name> 3. <servlet-**class**>HelloServlet</servlet-**class**> 6. </servlet>   7.   1. <servlet-mapping> 2. <servlet-name>HelloServlet</servlet-name> 3. <url-pattern>/servlet1</url-pattern> 4. </servlet-mapping>   12.   1. <filter> 2. <filter-name>f1</filter-name> 3. <filter-**class**>MyFilter</filter-**class**> 4. <init-param> 5. <param-name>construction</param-name> 6. <param-value>no</param-value> 7. </init-param> 8. </filter> 9. <filter-mapping> 10. <filter-name>f1</filter-name> 11. <url-pattern>/servlet1</url-pattern> 12. </filter-mapping> 13. </web-app> |

# 16. THREAD SAFE SERVLET

A thread is a single execution process; in other words, an individual, sequential flow of control within a program. When we say that a program is multithreaded, we are not implying that the program runs two separate instances simultaneously (as if you concurrently executed the program twice from the command line). Rather, we are saying that the same instance (executed only once) spawns multiple threads that process this single instance of code. This means that more than one sequential flow of control runs through the same memory block. So what do we mean by *thread-safe,* you ask? When multiple threads execute a single instance of a program and therefore share memory, multiple threads could possibly be attempting to read and write to the same place in memory. Let's look at an example. If we have a multithreaded program, we will have multiple threads processing the same instance (see Figure 1).

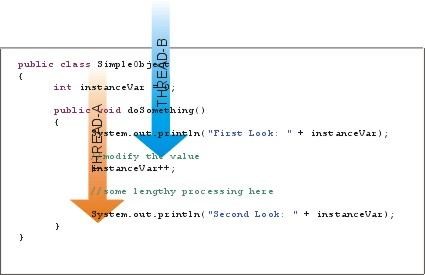
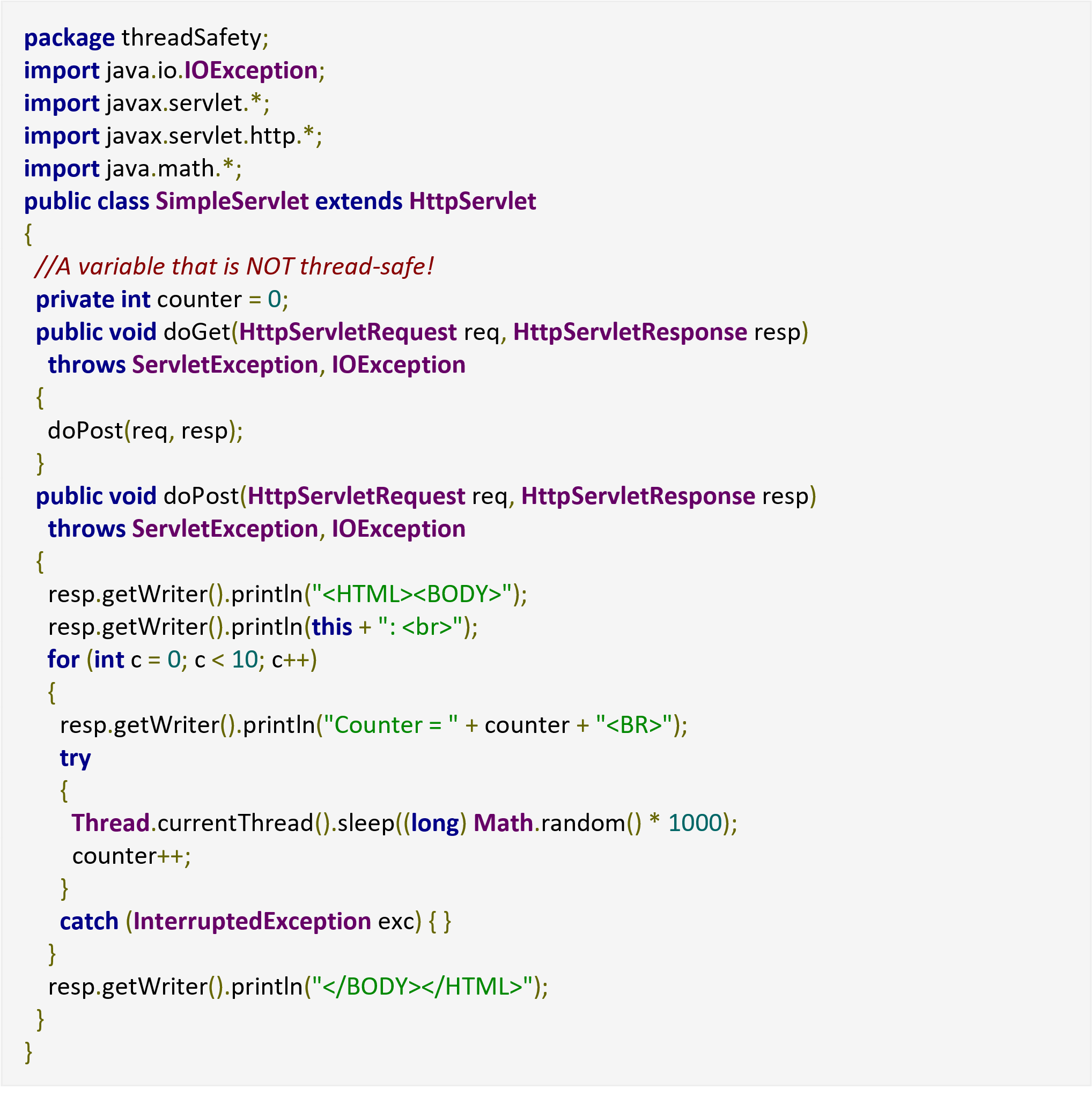


Figure 1. A multithreaded application

What happens when Thread-A examines variable instanceVar? Notice how Thread-B has just incremented instanceVar. The problem here is Thread-A has written to the instanceVar and is not expecting that value to change unless Thread-Aexplicitly does so. Unfortunately Thread-B is thinking the same thing regarding itself; the only problem is they share the same variable. This issue is not unique to servlets. It is a common programming problem only present when multithreading an application. You are probably thinking; "Well I didn't ask for multithreading. I just want a servlet!" And a servlet is what you have. Let me introduce you to our friend the servlet container.

## 16.1. EXAMPLE WITHOUT THREAD SAFE

Below is a simple servlet that is not thread-safe. Look closely, because at first glance, nothing appears wrong with it:

The variable counter is an instance variable, called such because it is tied to the class instance. Because it is defined within the class definition, it belongs within that class instance. It's convenient to place our variables within this scope because it lives outside each of the class's methods and can be accessed at any time. The value is also retained between method calls. The problem here is that our servlet container is multithreaded and shares single instances of servlets for multiple requests. Does defining your variables as instance variables sound like a good idea now? Remember, only one place in memory is allocated for this variable, and it is shared between all threads that intend on executing this same class instance. Let's find out what happens when we execute this servlet simultaneously. We add a delay in processing by using the sleep() method. This method helps simulate more accurate behavior, as most requests differ in the amount of time required for processing. Of course, as is our luck as programmers, this also causes our problem to occur more often. This simple servlet will increment counter such that each servlet should be able to display sequential values. We create simultaneous requests by using HTML frames; each frame's source is the same servlet:

**<HTML>**

**<BODY>**

**<TABLE>**

**<TR>**

**<TD>**

**<IFRAME** src="/theWebapp/SimpleServlet" name="servlet1" height="200%"**>**

**</IFRAME>**

**</TD>**

**</TR>**

**<TR>**

**<TD>**

**<IFRAME** src="/theWebapp/SimpleServlet" name="servlet2" height="200%"**>**

**</IFRAME>**

**</TD>**

**</TR>**

**<TR>**

**<TD>**

**<IFRAME** src="/theWebapp/SimpleServlet" name="servlet3" height="200%"**>**

**</IFRAME>**

**</TD>**

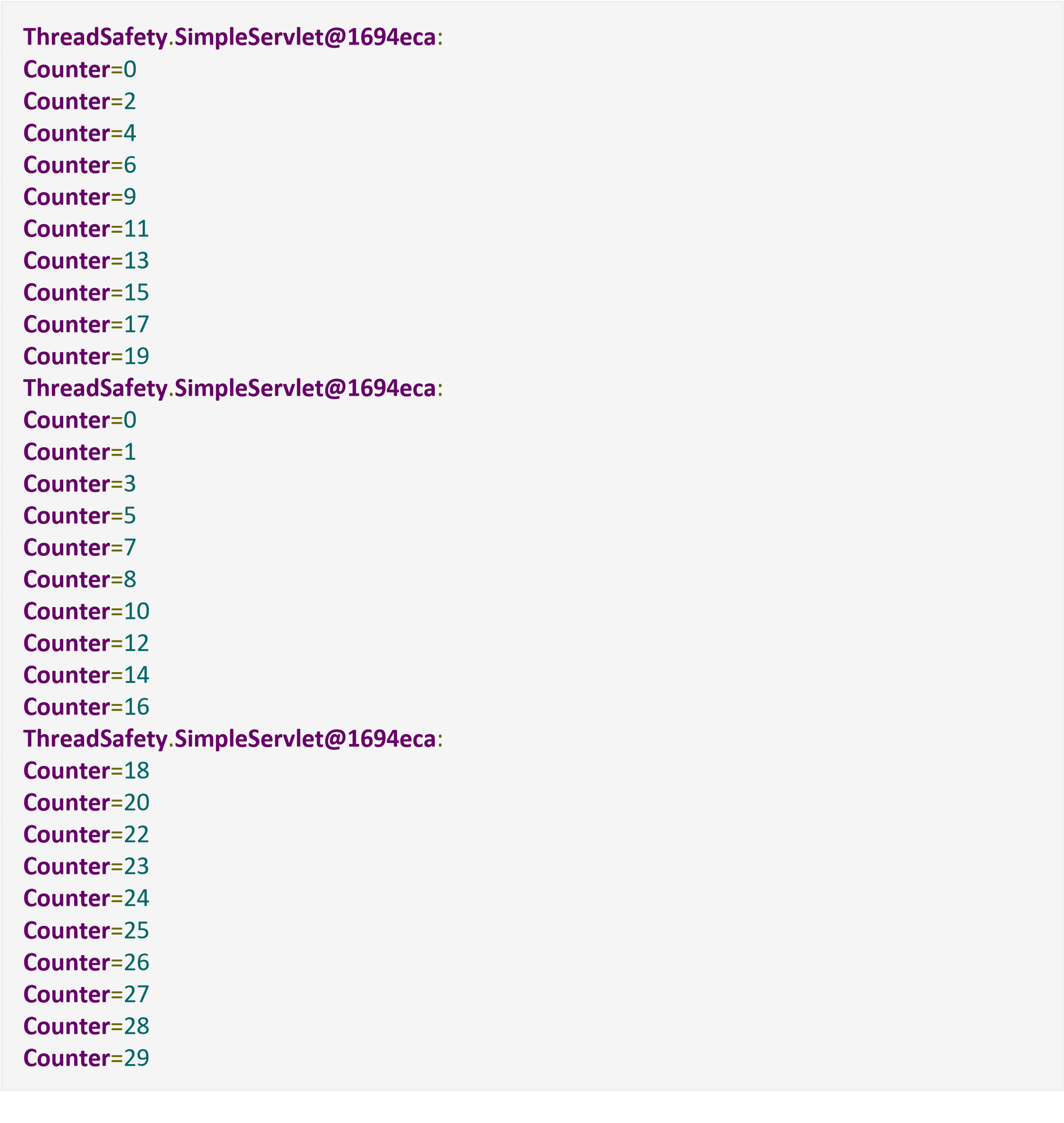
**</TR>**

**</TABLE>**

**</BODY>**

**</HTML>**

Our code, which is a non-thread-safe servlet, generates the following output:

As we can see in our output, we fail to get the results we desire. Notice the value printed from the this reference is duplicated. This is the servlet's memory address. It tells us that only one servlet is instantiated to service all requests. The servlet tried its best to output sequential data, but because all threads share the memory allocated for counter, we managed to step on our own toes. We can see that the values are not always sequential, which is bad! What if that variable is being used to point at a user's private information? What if a user logs into their online banking system and on a particular page, that user sees someone else's banking information? This problem can manifest itself in many ways, most of which are difficult to identify, but the good news is that this problem is easily remedied. So let's take a look at our options.

## 16.2. EXAMPLE USING THREAD SAFE

I have always said that the best way to fix problems is to avoid them all together; in our case, this approach is best. When discussing thread safety, we are interested only in the variables that we both read and write to and that pertain to a particular Web conversation. If the variable is for read-only use or it is application-wide, then no harm results in sharing this memory space across all instances. For all other variable uses, we want to make sure that we either have synchronized access to the variable (more on this in a moment) or that we have a unique variable for each thread. To ensure we have our own unique variable instance for each thread, we simply move the declaration of the variable from within the class to within the method using it. We have now changed our variable from an instance variable to a local variable. The difference is that, for each call to the method, a new variable is created; therefore, each thread has its own variable. Before, when the variable was an instance variable, the variable was shared for all threads processing that class instance. The following thread-safe code has a subtle, yet important, difference. Notice where the counter variable is declared!

|  |
| --- |
| **import** java.io.**IOException**; **import** javax.servlet.\*; **import** javax.servlet.http.\*; **import** java.math.\*;  **public** **class** **SimpleServlet** **extends** **HttpServlet**  {  **public** **void** doGet(**HttpServletRequest** req, **HttpServletResponse** resp) **throws** **ServletException**, **IOException**  {  doPost(req, resp);  }  **public** **void** doPost(**HttpServletRequest** req, **HttpServletResponse** resp) **throws** **ServletException**, **IOException**  {  *//A variable that IS thread-safe!* **private** **int** counter = 0;  resp.getWriter().println("<HTML><BODY>"); resp.getWriter().println(**this** + ": <br>"); **for** (**int** c = 0; c < 10; c++)  { |

resp

.

getWriter

().

println

(

"Counter = "

+

counter

+

"<BR>"

;

)

**try**

{

**Thread**

.

currentThread

().

sleep

((

**long**

)

**Math**

.

random

()

\*

1000

)

;

counter

++

;

}

**catch**

(

**InterruptedException**

exc

)

{

}

}

resp

.

getWriter

().

println

(

"</BODY

><

/HTML>"

;

)

}

}

Our code, which is now a thread-safe servlet, generates the following output:

**ThreadSafety**.**SimpleServlet@1694eca**:

**Counter**=0

**Counter**=1

**Counter**=2

**Counter**=3

**Counter**=4

**Counter**=5

**Counter**=6

**Counter**=7

**Counter**=8

**Counter**=9

**ThreadSafety**.**SimpleServlet@1694eca**:

**Counter**=0

**Counter**=1

**Counter**=2

**Counter**=3

**Counter**=4

**Counter**=5

**Counter**=6

**Counter**=7

**Counter**=8

**Counter**=9

**ThreadSafety**.**SimpleServlet@1694eca**:

**Counter**=0

**Counter**=1

Move the variable declaration to within the doGet() method and test again. Notice a change in behavior? I know, you're thinking; "It can't be that easy," but usually it is. As you scramble to revisit your latest servlet code to check where you declared your variables, you may run into a small snag. As you move your variables from within the class definition to within the method, you may find that you were leveraging the scope of the variable and accessing it from within other methods. If you find yourself in this situation, you have a couple of choices. First, change the method interfaces and pass this variable (and any other shared variables) to each method requiring it. I highly recommend this approach. Explicitly passing your data elements from method to method is always best; it clarifies your intentions, documents each method's requirements, makes your code well structured, and offers many other benefits. If you discover that you must share a variable between servlets and this variable is going to be read from and written to by multiple threads (and you are not storing it in a database), then you will require thread synchronization.

**Counter**

=

2

**Counter**

=

3

**Counter**

=

4

**Counter**

=

5

**Counter**

=

6

**Counter**

=

7

**Counter**

=

8

**Counter**

=

9

# 17. SERVLET LISTENER

We know that using ServletContext, we can create an attribute with application scope that all other servlets can access but we can initialize **ServletContext init parameters as String only** in deployment descriptor (web.xml). What if our application is database oriented and we want to set an attribute in ServletContext for Database Connection. If you application has a single entry point (user login), then you can do it in the first servlet request but if we have multiple entry points then doing it everywhere will result in a lot of code redundancy. Also if database is down or not configured properly, we won’t know until first client request comes to server. To handle these scenario, servlet API provides Listener interfaces that we can implement and configure to listen to an event and do certain operations. **Event** is occurrence of something, in web application world an event can be initialization of application, destroying an application, request from client, creating/destroying a session, attribute modification in session etc. **Servlet API** provides different types of Listener interfaces that we can implement and configure in web.xml to process something when a particular event occurs. For example, in above scenario we can create a Listener for the application startup event to read context init parameters and create a database connection and set it to context attribute for use by other resources. We can perform some important tasks at the occurrence of these exceptions, such as counting total and current logged-in users, creating tables of the database at time of deploying the project, creating database connection object etc.

There are many Event classes and Listener interfaces in the javax.servlet and javax.servlet.http packages.

The event classes are as follows:

1. ServletRequestEvent
2. ServletContextEvent
3. ServletRequestAttributeEvent
4. ServletContextAttributeEvent
5. HttpSessionEvent
6. HttpSessionBindingEvent

The event interfaces are as follows:

1. ServletRequestListener
2. ServletRequestAttributeListener
3. ServletContextListener
4. ServletContextAttributeListener
5. HttpSessionListener
6. HttpSessionAttributeListener
7. HttpSessionBindingListener
8. HttpSessionActivationListener
   1. **SERVLETCONTEXTEVENT AND SERVLETCONTEXTLISTENER**

The ServletContextEvent is notified when web application is deployed on the server. If you want to perform some action at the time of deploying the web application such as creating database

connection, creating all the tables of the project etc, you need to implement ServletContextListener interface and provide the implementation of its methods.

* 1. **CONSTRUCTOR OF SERVLETCONTEXTEVENT CLASS**

There is only one constructor defined in the ServletContextEvent class. The web container creates the instance of ServletContextEvent after the ServletContext instance.

1. ServletContextEvent(ServletContext e)

There is only one method defined in the ServletContextEvent class:

1. **public ServletContext getServletContext()**: returns the instance of ServletContext.

There are two methods declared in the ServletContextListener interface which must be implemented by the servlet programmer to perform some action such as creating database connection etc.

1. **public void contextInitialized(ServletContextEvent e)**: is invoked when application is deployed on the server.
2. **public void contextDestroyed(ServletContextEvent e)**: is invoked when application is undeployed from the server.

**17.3.EXAMPLE OF SERVLETCONTEXTEVENT AND SERVLETCONTEXTLISTENER**

In this example, we are retrieving the data from the emp32 table. To serve this, we have created the connection object in the listener class and used the connection object in the servlet.

**index.html**

1. <a href="servlet1">fetch records</a>

**MyListener.java**

|  |  |  |
| --- | --- | --- |
| 1. **import** javax.servlet.\*; 2. **import** java.sql.\*; 3. **public** **class** MyListener **implements** ServletContextListener{ 4. **public** **void** contextInitialized(ServletContextEvent event) { 5. **try**{ 5. Class.forName("oracle.jdbc.driver.OracleDriver"); 6. Connection con=DriverManager.getConnection( 7. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");   9.   1. //storing connection object as an attribute in ServletContext 2. ServletContext ctx=event.getServletContext(); 3. ctx.setAttribute("mycon", con); 4. }**catch**(Exception e){e.printStackTrace();} 5. } 6. **public** **void** contextDestroyed(ServletContextEvent arg0) {} 7. }   17. | | |
|  | **MyListener.java** |  |

1. **import** java.io.\*;
2. **import** javax.servlet.\*;
3. **import** javax.servlet.http.\*;
4. **import** java.sql.\*;
5. **public** **class** FetchData **extends** HttpServlet {
6. **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)
7. **throws** ServletException, IOException {
8. response.setContentType("text/html");
9. PrintWriter out = response.getWriter();
10. **try**{
11. //Retrieving connection object from ServletContext object
12. ServletContext ctx=getServletContext();
13. Connection con=(Connection)ctx.getAttribute("mycon");
14. //retieving data from emp32 table
15. PreparedStatement ps=con.prepareStatement("select \* from emp32",
16. ResultSet.TYPE\_SCROLL\_SENSITIVE,ResultSet.CONCUR\_UPDATABLE);
17. ResultSet rs=ps.executeQuery();
18. **while**(rs.next()){
19. out.print("<br>"+rs.getString(1)+" "+rs.getString(2));
20. }
21. con.close();
22. }**catch**(Exception e){e.printStackTrace();}
23. out.close();
24. }
25. }

**17.4.EXAMPLE OF SERVLETCONTEXTLISTENER TO CREATE TABLE OF A PROJECT**

In this example, we are creating table of the project. So we don't need to create all the tables manually in the database.

**MyListener.java**

|  |
| --- |
| 1. **import** javax.servlet.\*; 2. **import** java.sql.\*; 3. **public** **class** MyListener **implements** ServletContextListener{ 4. **public** **void** contextInitialized(ServletContextEvent arg0) { 5. **try**{ 5. Class.forName("oracle.jdbc.driver.OracleDriver"); 6. Connection con=DriverManager.getConnection(" 7. jdbc:oracle:thin:@localhost:1521:xe","system","oracle"); 8. String query="create table emp32(id number(10),name varchar2(40))"; 9. PreparedStatement ps=con.prepareStatement(query); 10. ps.executeUpdate(); 11. System.out.println(query); 12. }**catch**(Exception e){e.printStackTrace();} 13. } 14. **public** **void** contextDestroyed(ServletContextEvent arg0) { 15. System.out.println("project undeployed"); 16. } 17. } |



**HITESH KUMAR SHARMA / RAVI TOMAR hksharma@ddn.upes.ac.in/rtomar@ddn.upes.ac.in**

**CENTER FOR INFORMATION TECHNOLOGY**

# UNIVERSITY OF PETROLEUM AND ENERGY STUDIES (UPES)

**“ENERGY ACRES”, P.O. BIDHOLI, VIA PREM NAGAR, DEHRADUN 248 007**