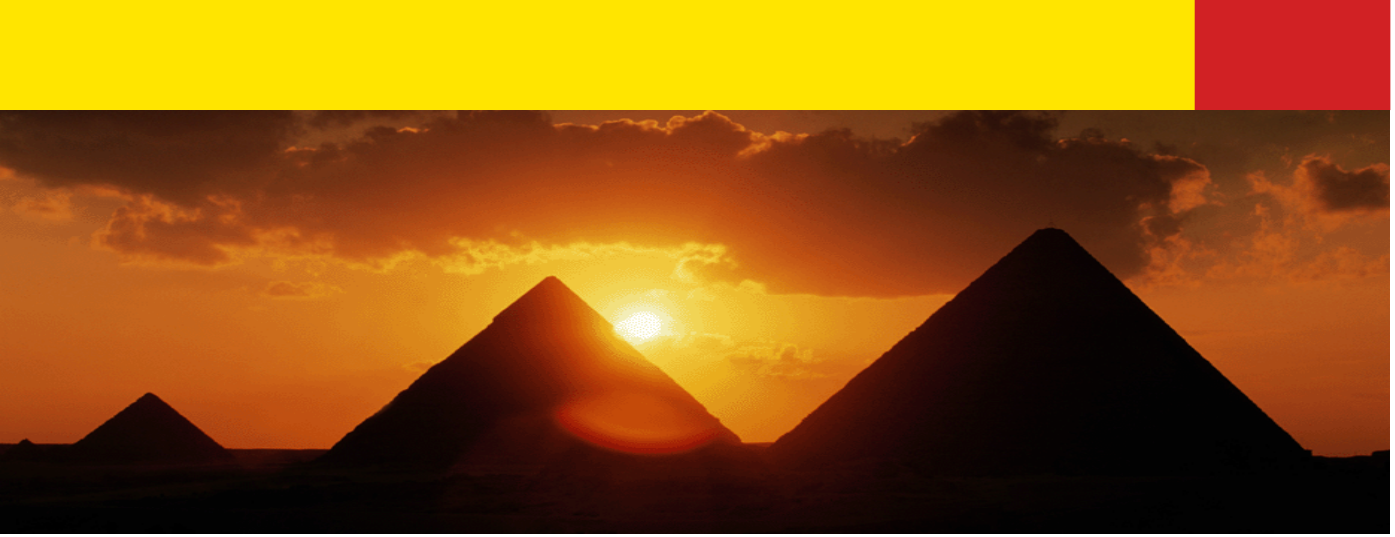
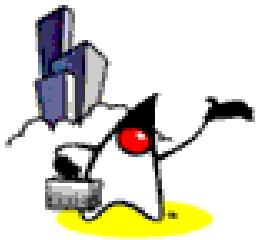
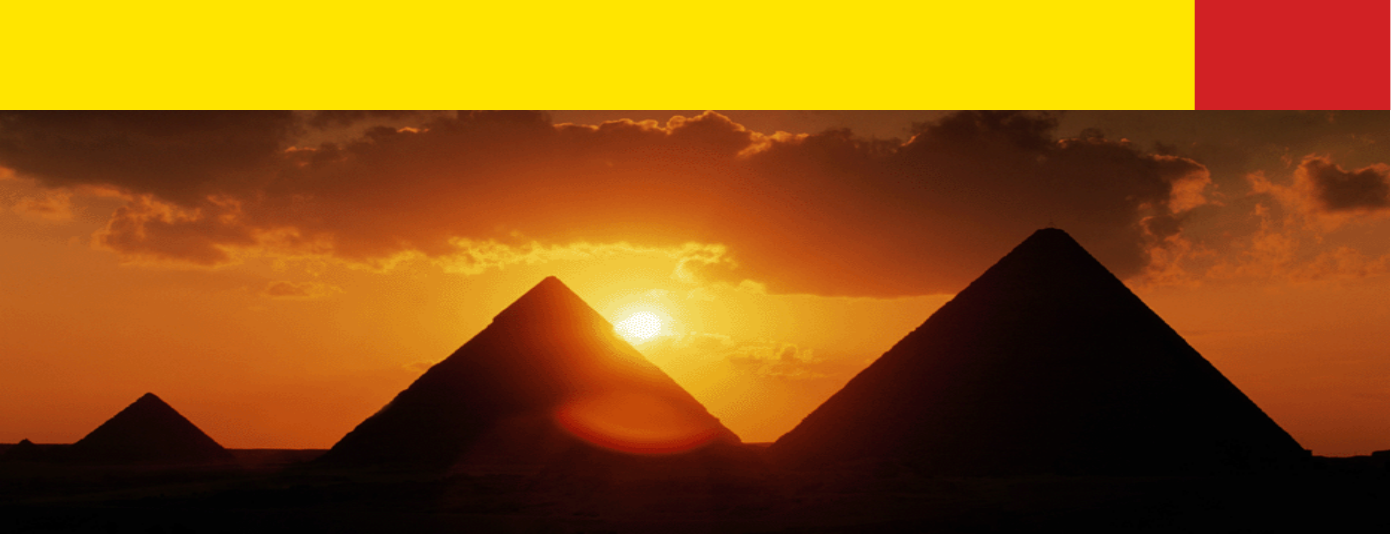


Servlet (Advanced)

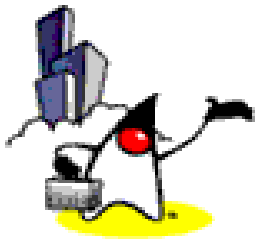


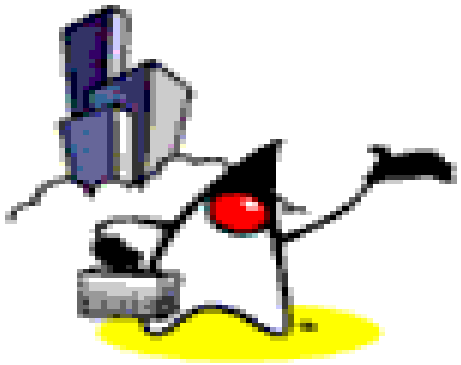
Agenda

- Including, forwarding to, and redirecting to other web resources
- Servlet life-cycle events
- Concurrency Issues with SingleThreadModel
- Invoker Servlet



Including, Forwarding, Redirecting to other Web resources





Including another Web Resource

When to Include another Web resource?

- When it is useful to add static or dynamic contents already created by another web resource
  - Adding a banner content or copyright information in the response returned from a Web component

Types of Included Web Resource

- Static resource
  - It is like “programmatic” way of adding the static contents in the response of the “including” servlet
- Dynamic web component (Servlet or JSP page)
  - Send the request to the “included” Web component
  - Execute the “included” Web component
  - Include the result of the execution from the “included” Web component in the response of the “including” servlet

Things that Included Web Resource can and cannot do

- Included Web resource has access to the request object, but it is limited in what it can do with the response
  - It can write to the body of the response and commit a response
  - It cannot set headers or call any method (for example, setCookie) that affects the headers of the response

How to Include another Web resource?

- Get RequestDispatcher object from ServletConext object

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/banner");
```

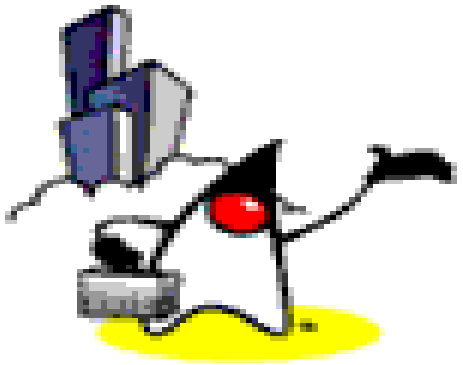
- Then, invoke the include() method of the RequestDispatcher object passing request and response objects
  - dispatcher.include(request, response);

Example: BannerServlet as “Included” Web component

```
public class BannerServlet extends HttpServlet {
  public void doGet (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    out.println("<body bgcolor=\"#ffffff\">" +
    "<center>" + "<hr> <br>  " + "<h1>" +
    "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
    <img src=\"" + request.getContextPath() +
    "/duke.books.gif\">" +
    "<font size=\"+3\" color=\"black\">Bookstore</font>" +
    "</h1>" + "</center>" + "<br>   <hr> <br> ");
  }
  public void doPost (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    out.println("<body bgcolor=\"#ffffff\">" +
    "<center>" + "<hr> <br>  " + "<h1>" +
    "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
    <img src=\"" + request.getContextPath() +
    "/duke.books.gif\">" +
    "<font size=\"+3\" color=\"black\">Bookstore</font>" +
    "</h1>" + "</center>" + "<br>   <hr> <br> ");
  }
}
```

Example: Including “BannerServlet”

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/banner");
if (dispatcher != null)
  dispatcher.include(request, response);
}
```



Forwarding to another Web Resource

When to use “Forwarding” to another Web resource?

- When you want to have one Web component do preliminary processing of a request and have another component generate the response
- You might want to partially process a request and then transfer to another component depending on the nature of the request

Rules of “Forwarding” to another Web resource?

- Should be used to give another resource responsibility for replying to the user
  - If you have already accessed a ServletOutputStream or PrintWriter object within the servlet, you cannot use this method; it throws an IllegalStateException

How to do “Forwarding” to another Web resource?

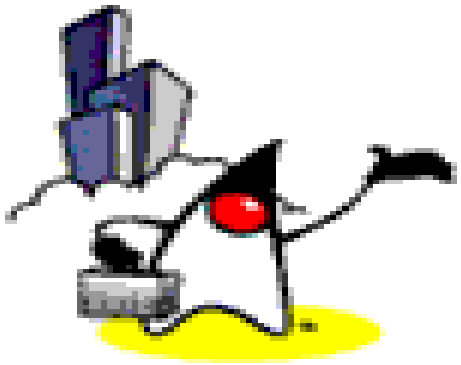
- Get RequestDispatcher object from HttpServletRequest object
  - Set “request URL” to the path of the forwarded page

```
RequestDispatcher dispatcher
= request.getRequestDispatcher("/template.jsp");
```

- If the original URL is required for any processing, you can save it as a request attribute
- Invoke the forward() method of the RequestDispatcher object
  - dispatcher.forward(request, response);

Example: Dispatcher Servlet

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



Instructing Browser to Redirecting to another Web Resource

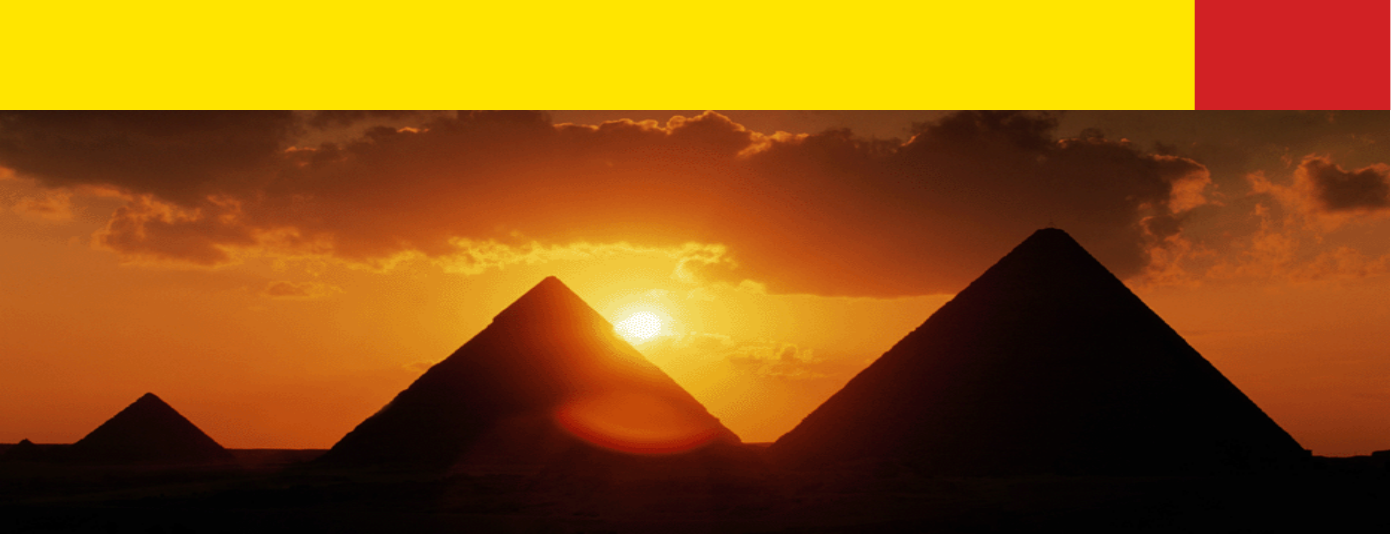
Redirecting a Request

- Two programming models for directing a request
- Method 1:

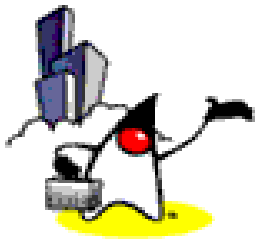
```
res.setStatus(res.SC_MOVED_PERMANTLY);
res.setHeader("Location", "http://...");
```

- Method 2:

```
public void sendRedirect(String url)
```

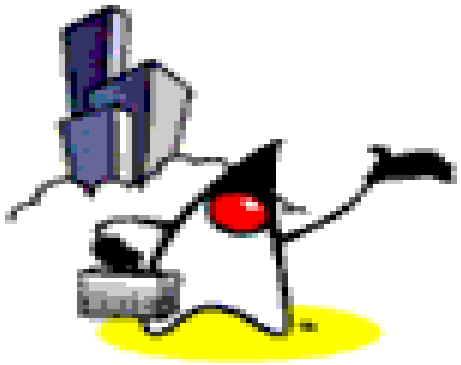


Servlet Filters



Sub-Agenda: Servlet Filters

- What is & Why servlet filters (with use case scenarios)?
- How are servlet filters chained?
- Servlet filter programming APIs
- Servlet filter configuration in web.xml
- Steps for building and deploying servlet filters
- Example code



What is and Why Servlet Filter?

What are Java Servlet Filters?

- New component framework for intercepting and modifying requests and responses
  - Filters can be chained and plugged in to the system during deployment time
- Allows range of custom activities:
  - Marking access, blocking access
  - Caching, compression, logging
  - Authentication, access control, encryption
  - Content transformations
- Introduced in Servlet 2.3 (Tomcat 4.0)

What Can a Filter Do?

- Examine the request headers
- Customize the request object if it wishes to modify request headers or data
- Customize the response object if it wishes to modify response headers or data
- Invoke the next entity in the filter chain
- Examine response headers after it has invoked the next filter in the chain
- Throw an exception to indicate an error in processing

Use Case Scenario 1 of Filters[4]

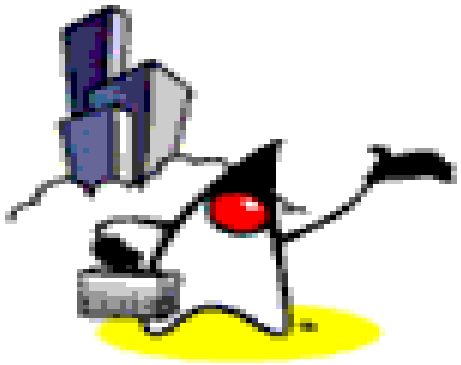
- You have many servlets and JSP pages that need to perform common functions such as logging or XSLT transformation
  - You want to avoid changing all these servlets and JSP pages
  - You want to build these common functions in a modular and reusable fashion
- Solution
  - build a single logging filter and compression filter
  - plug them at the time of deployment

Use Case Scenario 2 of Filters[4]

- How do you separate access control decisions from presentation code (JSP pages)
  - You do not want to change individual JSP pages since it will mix “access-control” logic with presentation logic
- Solution
  - build and deploy a “access-control” servlet

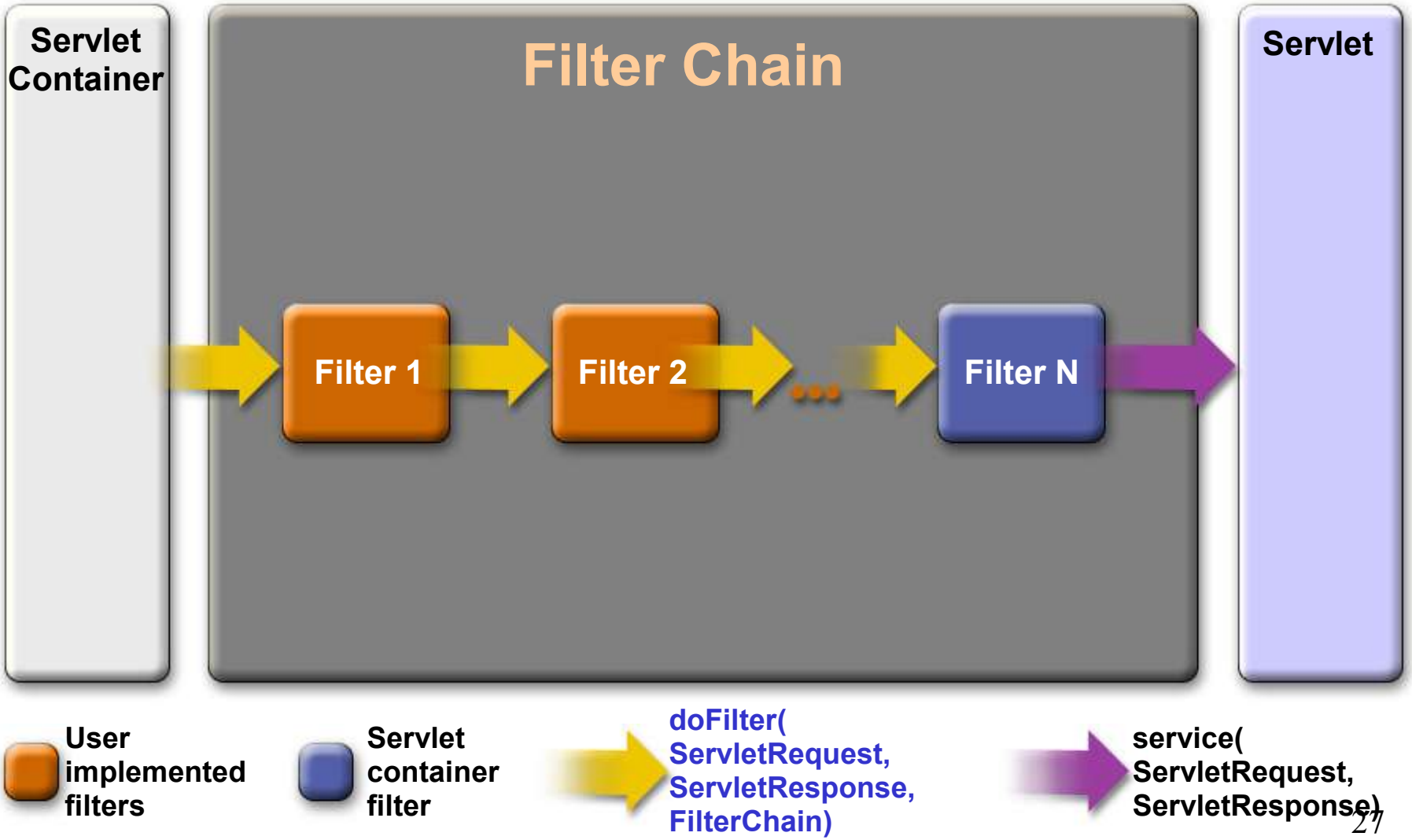
Use Case Scenario 3 of Filters[4]

- You have many existing Web resources that need to remain unchanged except a few values (such as banners or company name)
  - You cannot make changes to these Web resources every time company name gets changed
- Solution
  - Build and deploy “banner replacement filter” or “company name replacement filter”



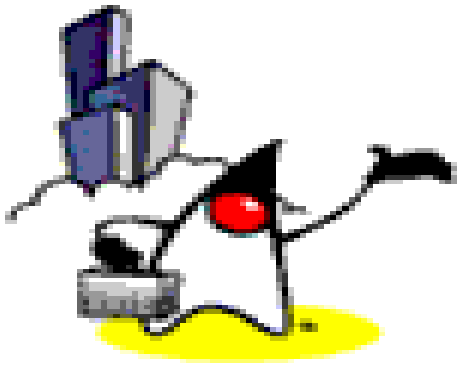
How are Servlet Filters chained?

How Servlet Filter Work?



How Filter Chain Works

- Multiple filters can be chained
  - order is dictated by the order of <filter> elements in the web.xml deployment descriptor
- The first filter of the filter chain is invoked by the container
  - via doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
  - the filter then perform whatever filter logic and then call the next filter in the chain by calling chain.doFilter(..) method
- The last filter's call to chain.doFilter() ends up calling service() method of the Servlet



Servlet Filter Programming APIs

javax.servlet.Filter Interface

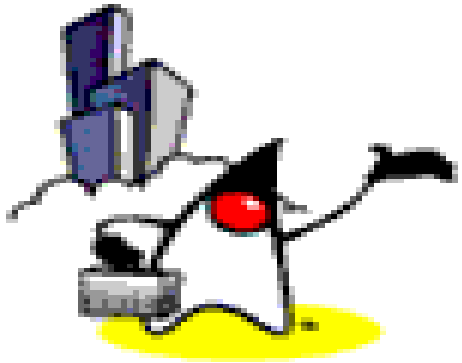
- init(FilterConfig)
  - called only once when the filter is first initialized
  - get ServletContext object from FilterConfig object and save it somewhere so that doFilter() method can access it
  - read filter initialization parameters from FilterConfig object through getInitParameter() method
- destroy()
  - called only once when container removes filter object
  - close files or database connections

javax.servlet.Filter Interface

- doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
  - gets called each time a filter is invoked
  - contains most of filtering logic
  - ServletRequest object is casted to HttpServletRequest if the request is HTTP request type
  - may wrap request/response objects
  - invoke next filter by calling chain.doFilter(..)
  - or block request processing
    - by omitting calling chain.doFilter(..)
    - filter has to provide output the client
  - set headers on the response for next entity

Other Sevlet Filter Related Classes

- javax.servlet.FilterChain
  - passed as a parameter in doFilter() method
- javax.servlet.FilterConfig
  - passed as a parameter in init() method
- javax.servlet.HttpServletResponseWrapper
  - convenient implementation of the HttpServletResponse interface



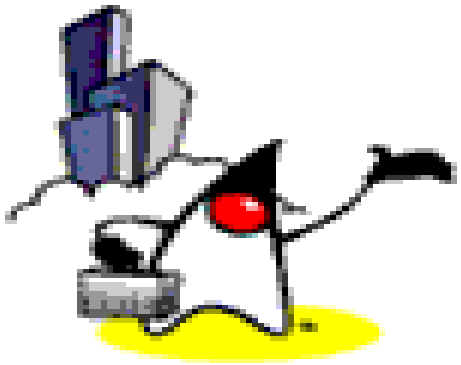
Servlet Filter Configuration in the web.xml file

Configuration in web.xml

- <filter>
  - <filter-name>: assigns a name of your choosing to the filter
  - <filter-class>: used by the container to identify the filter class
- </filter>
- <filter-mapping>
  - <filter-name>: assigns a name of your choosing to the filter
  - <url-pattern>: declares a pattern URLs (Web resources) to which the filter applies
- </filter-mapping>

Example: web.xml of BookStore

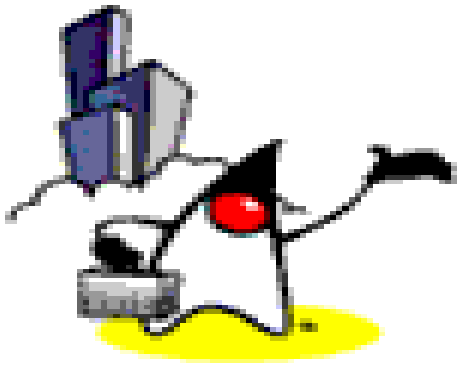
```
<web-app>
 <display-name>Bookstore1</display-name>
 <description>no description</description>
 <filter>
  <filter-name>OrderFilter</filter-name>
  <filter-class>filters.OrderFilter</filter-class>
 </filter>
 <filter>
  <filter-name>HitCounterFilter</filter-name>
  <filter-class>filters.HitCounterFilter</filter-class>
 </filter>
 <filter-mapping>
  <filter-name>OrderFilter</filter-name>
  <url-pattern>/receipt</url-pattern>
 </filter-mapping>
 <filter-mapping>
  <filter-name>HitCounterFilter</filter-name>
  <url-pattern>/enter</url-pattern>
 </filter-mapping>
 <listener>
  ...
 </listener>
 <servlet>
  ...
```



Steps for Building and Deploying Servlet Filters

Steps for Building a Servlet Filter

- Decide what custom filtering behavior you want to implement for a web resource
- Create a class that implements Filter interface
  - Implement filtering logic in the doFilter() method
  - Call the doFilter() method of FilterChain object
- Configure the filter with Target servlets and JSP pages
  - use <filter> and <filter-mapping> elements



Servlet Filter Example Code

Example: HitCounterFilter

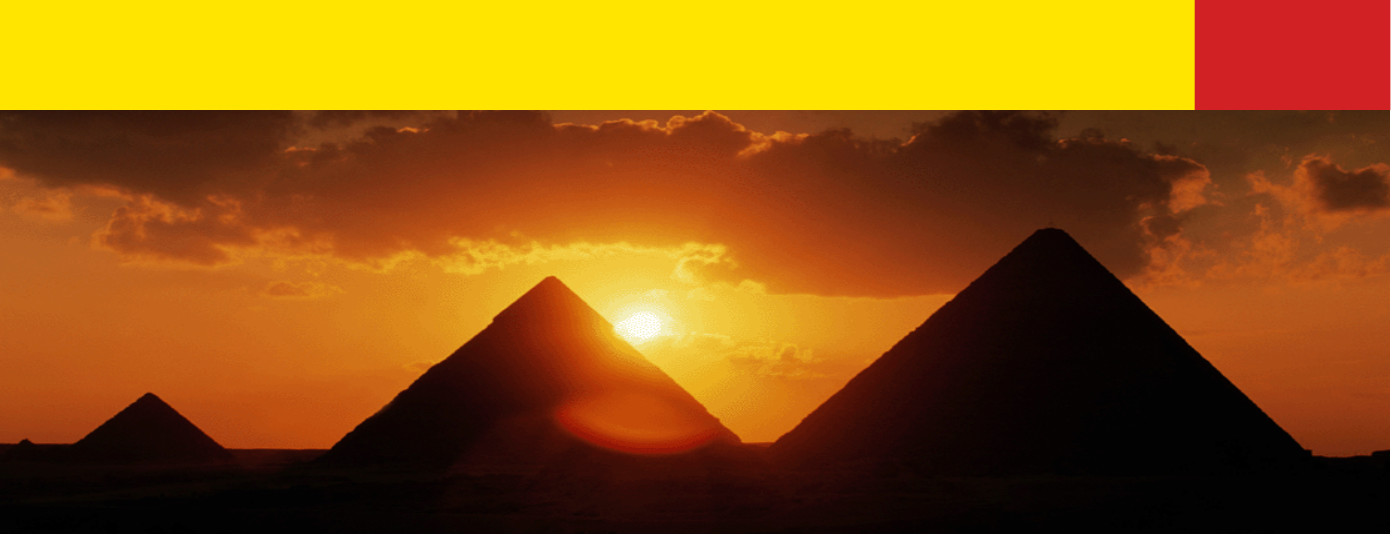
```
public final class HitCounterFilter implements Filter {
  private FilterConfig filterConfig = null;

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

  // Continued in the next page...
```

Example: HitCounterFilter

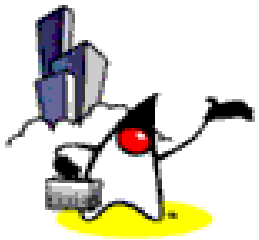
```
public void doFilter(ServletRequest request,
            ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

  if (filterConfig == null) return;
  StringWriter sw = new StringWriter();
  PrintWriter writer = new PrintWriter(sw);
  Counter counter =
   (Counter) filterConfig.getServletContext().getAttribute("hitCounter");
  writer.println("The number of hits is: " +
    counter.incCounter());

  // Log the resulting string
  writer.flush();
  filterConfig.getServletContext().log(sw.getBuffer().toString());
  ...
  chain.doFilter(request, wrapper);
  ...
  }
}
```

HitCounterFilter Configuration

```
<?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>
  <display-name>Bookstore1</display-name>
  <description>no description</description>

  <filter>
    <filter-name>HitCounterFilter</filter-name>
    <filter-class>filters.HitCounterFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>HitCounterFilter</filter-name>
    <url-pattern>/enter</url-pattern>
  </filter-mapping>
  ...
```



Servlet LifeCycle Events



Servlet Lifecycle Events

- Support event notifications for state changes in
  - ServletContext
    - Startup/shutdown
    - Attribute changes
  - HttpSession
    - Creation and invalidation
    - Changes in attributes

Steps for Implementing Servlet Lifecycle Event

1. Decide which scope object you want to receive an event notification
2. Implement appropriate interface
3. Override methods that need to respond to the events of interest
4. Obtain access to important Web application objects and use them
5. Configure web.xml accordingly
6. Provide any needed initialization parameters

Listener Registration

- Web container
  - creates an instance of each listener class
  - registers it for event notifications before processing first request by the application
  - Registers the listener instances according to
    - the interfaces they implement
    - the order in which they appear in the deployment descriptor web.xml
- Listeners are invoked in the order of their registration during execution

Listener Interfaces

- ServletContextListener
  - contextInitialized/Destroyed(ServletContextEvent)
- ServletContextAttributeListener
  - attributeAdded/Removed/Replaced(ServletContextAttributeEvent)
- HttpSessionListener
  - sessionCreated/Destroyed(HttpSessionEvent)
- HttpSessionAttributeListener
  - attributedAdded/Removed/Replaced(HttpSessionBindingEvent)
- HttpSessionActivationListener
  - Handles sessions migrate from one server to another
  - sessionWillPassivate(HttpSessionEvent)
  - sessionDidActivate(HttpSessionEvent)

Example: Context Listener

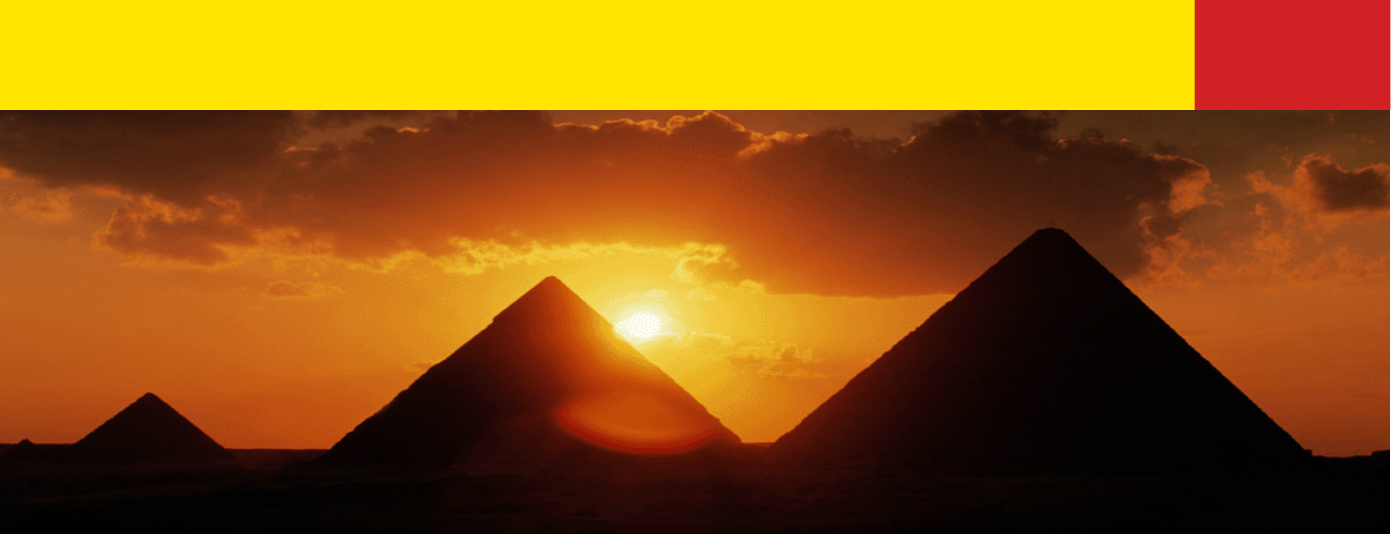
```
public final class ContextListener
     implements ServletContextListener {
     private ServletContext context = null;

     public void contextInitialized(ServletContextEvent event)
          context = event.getServletContext();

          try {
             BookDB bookDB = new BookDB();
             context.setAttribute("bookDB", bookDB);
          } catch (Exception ex) {
             context.log("Couldn't create bookstore
                          database bean: " + ex.getMessage());
          }

          Counter counter = new Counter();
          context.setAttribute("hitCounter", counter);
          counter = new Counter();
          context.setAttribute("orderCounter", counter);
     }
```

Example: Context Listener

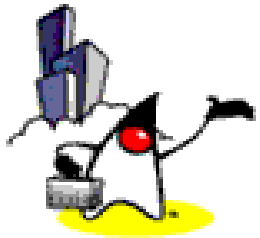
```
  public void contextDestroyed(ServletContextEvent event) {
        context = event.getServletContext();
        BookDB bookDB = (BookDB)context.getAttribute
("bookDB");
        bookDB.remove();
        context.removeAttribute("bookDB");
        context.removeAttribute("hitCounter");
        context.removeAttribute("orderCounter");
  }
}
```

Listener Configuration

```
<web-app>
  <display-name>Bookstore1</display-name>
  <description>no description</description>

  <filter>..</filter>
  <filter-mapping>..</filter-mapping>
  <listener>
    <listener-class>listeners.ContextListener</listener-class>
  </listener>
  <servlet>..</servlet>
  <servlet-mapping>..</servlet-mapping>
  <session-config>..</session-config>
  <error-page>..</error-page>
  ...
</web-app>
```



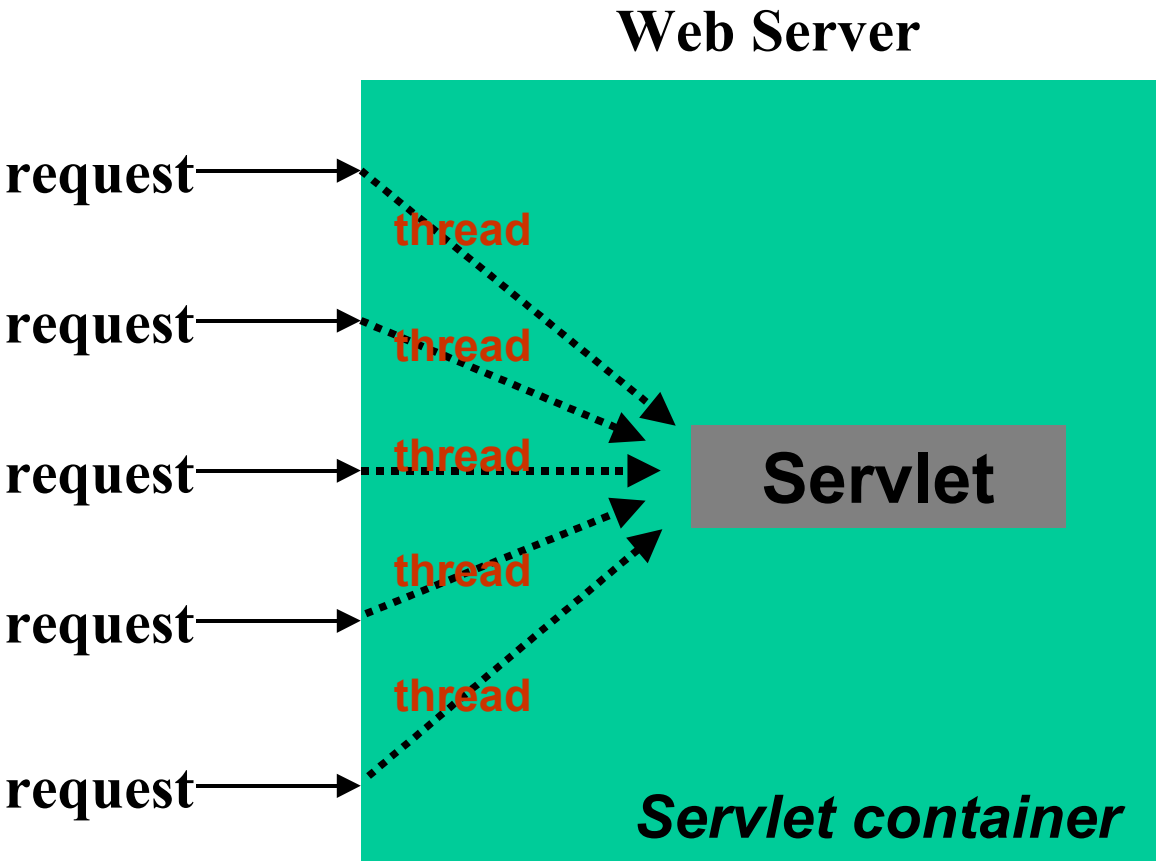
Servlet Synchronization & Thread Model



Concurrency Issues on a Servlet

- The service() method of a servlet instance can be invoked by multiple clients (multiple threads)
- Servlet programmer has to deal with concurrency issue
  - shared data needs to be protected
  - this is called "servlet synchronization"
- 2 options for servlet synchronization
  - use of synchronized block
  - use of SingleThreadModel

Many Threads, One Servlet Instance



Use of synchronized block

- Synchronized blocks are used to guarantee only one thread at a time can execute within a section of code

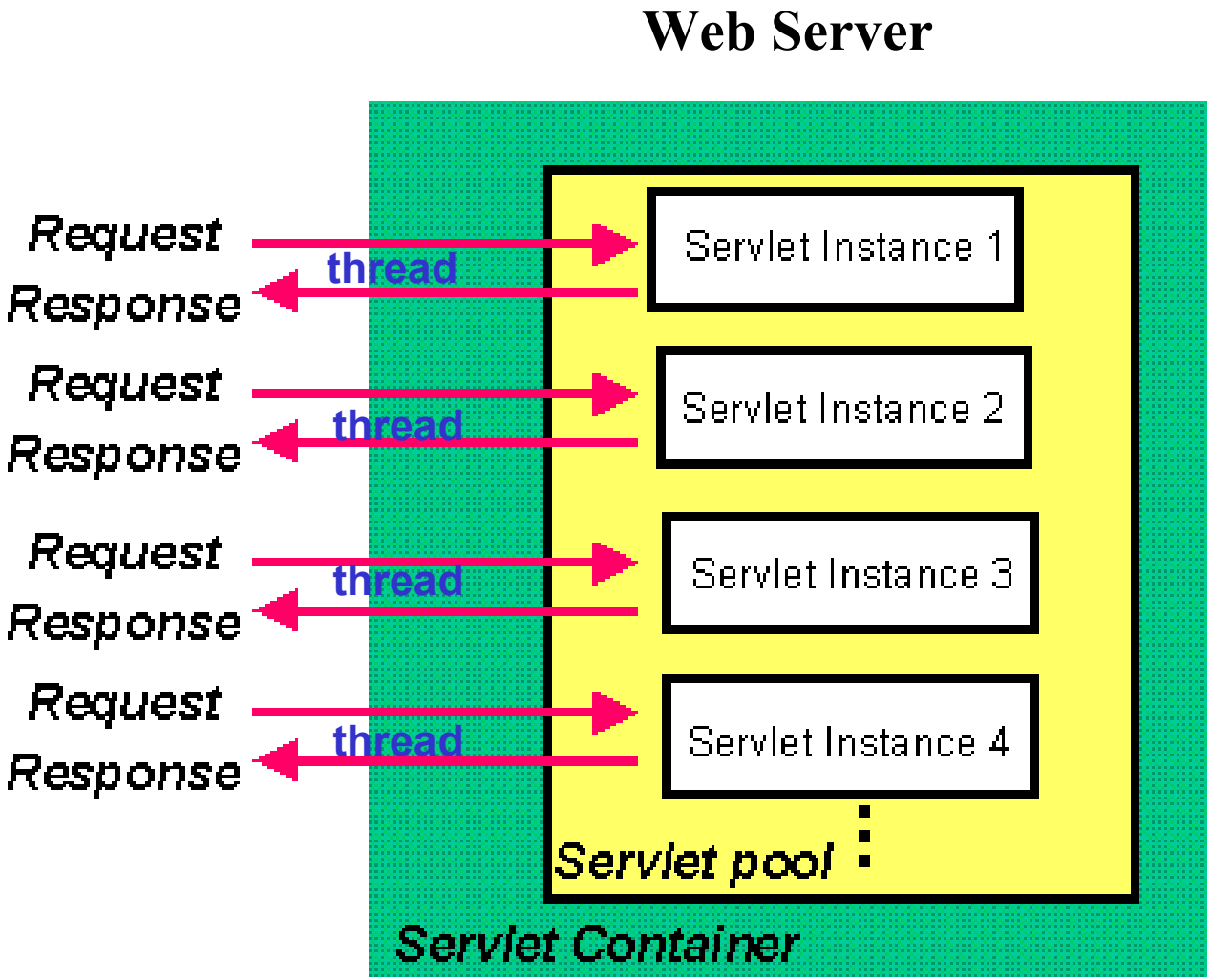
```
synchronized(this) {
      myNumber = counter + 1;
      counter = myNumber;
 }
...
synchronized(this) {
      counter = counter - 1 ;
 }
```

SingleThreadModel Interface

- Servlets can also implement javax.servlet.SingleThreadModel
- The server will manage a pool of servlet instances
- Guaranteed there will only be one thread per instance
- This could be overkill in many instances

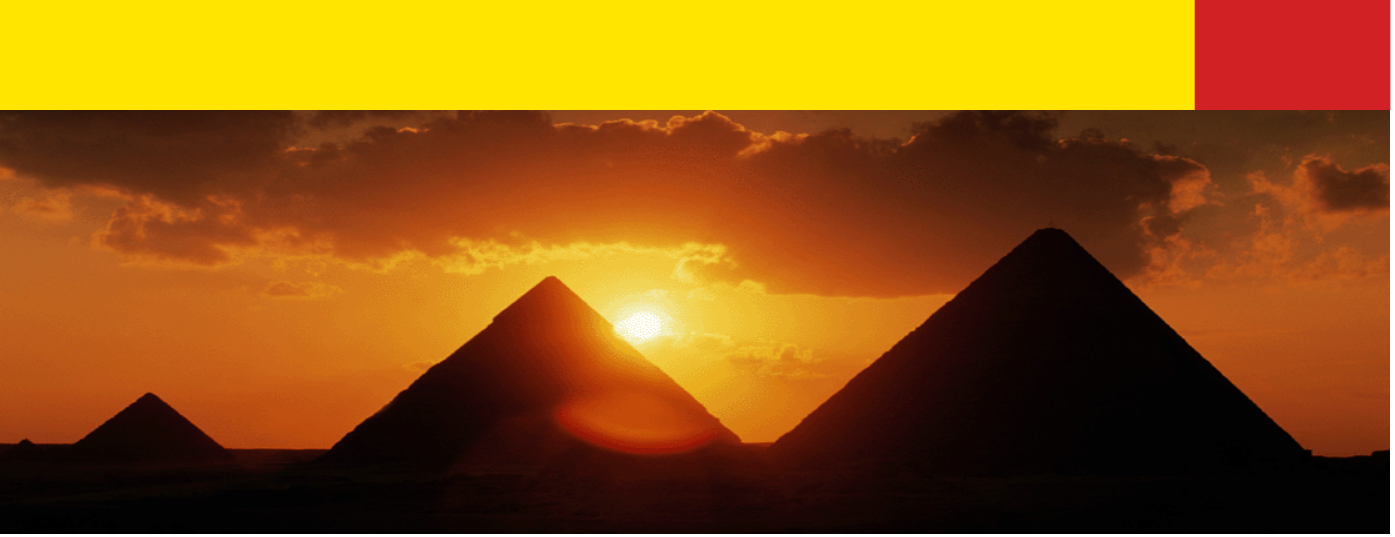
```
Public class SingleThreadModelServlet extends
   HttpServlet implements SingleThreadModel {
...
}
```

SingleThreadModel

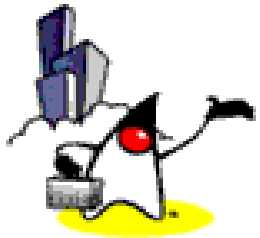


Best Practice Recommendation

- Do use synchronized block whenever possible
  - SingleThreadModel is expensive (performance wise)



Invoker Servlet



What is Invoke Servlet?

- Executes anonymous servlet classes that have not been defined in a web.xml file
- Mostly used for debugging and testing purpose

How to Use Invoke Servlet?

- Uncomment the following from <Tomcat>/conf/web.xml and restart Tomcat

```
<!--
   <servlet>
      <servlet-name>invoker</servlet-name>
      <servlet-class>
       org.apache.catalina.servlets.InvokerServlet
      </servlet-class>
      <init-param>
         <param-name>debug</param-name>
         <param-value>0</param-value>
      </init-param>
      <load-on-startup>2</load-on-startup>
   </servlet>
-->
```

How to Use Invoke Servlet?

- Add the following to the web.xml of the application

```
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/myservlets/*</url-pattern>
</servlet-mapping>
```

- From your browser, access a servlet via
  - http://localhost:8080/myservlets/myServlet

Passion!