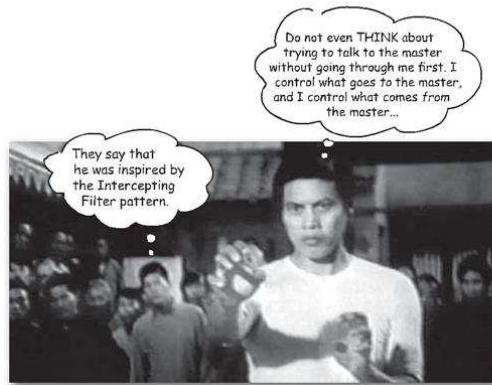


13 filters and wrappers

The Power of Filters



Filters let you intercept the request. And if you can intercept the request, you can also control the response. And best of all, **the servlet remains clueless.** It never knows that someone stepped in between the client request and the Container's invocation of the servlet's `service()` method. What does that mean to you? More vacations. Because the time you would have spent rewriting just one of your servlets can be spent instead writing and configuring a filter that has the ability to affect *all* of your servlets. Want to add user request tracking to *every* servlet in your app? No problem. Want to manipulate the output from *ever* servlet in your app? No problem. And you don't even have to *touch* the servlet code. Filters may be the most powerful web app development tool you have.

this is a new chapter 669

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

official Sun exam objectives

OBJECTIVES

Filters

Coverage Notes:

3.3 Describe the Web Container request processing model; write and configure a filter; create a request or response wrapper; and given a design problem, describe how to apply a filter or wrapper.

This objective is covered completely in this chapter.

11.1 Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: **Intercepting Filter**, **Model-View-Controller**, **Front Controller**, **Service Locator**, **Business Delegate**, and **Transfer Object**.

11.1 Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: **Intercepting Filter**, **Model-View-Controller**, **Service Locator**, **Business Delegate**, and **Transfer Object**.

Filters, which are covered in this chapter, are an example of (imagine this) the Intercepting Filter pattern. We don't cover pattern-specific info until the Patterns chapter, but it's in THIS chapter where you actually see a design that demonstrates the Intercepting Filter pattern.

Enhancing the entire web application

Sometimes you need to enhance your system in ways that span many different use cases or requests. For example, you might want to keep track of your system's response times, across all of its different user interactions.

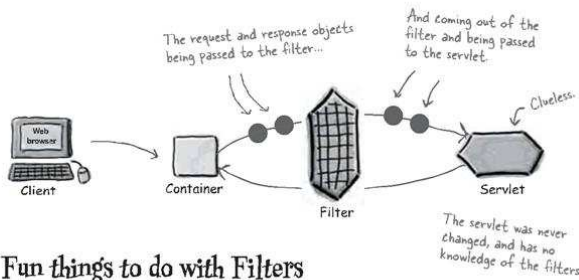


request and response filters

How about some kind of “filter”?

Filters are Java components—very similar to servlets—that you can use to intercept and process requests *before* they are sent to the servlet, or to process responses *after* the servlet has completed, but *before* the response goes back to the client.

The Container decides when to invoke your filters based on declarations in the DD. In the DD, the deployer maps which filters will be called for which request URI patterns. So it's the deployer, not the programmer, who decides which subset of requests or responses should be processed by which filters.



Fun things to do with Filters

Request filters can:

- ▶ perform security checks
- ▶ reformat request headers or bodies
- ▶ audit or log requests

Response filters can:

- ▶ compress the response stream
- ▶ append or alter the response stream
- ▶ create a different response altogether

There is only ONE filter interface, Filter.

There's no such thing as a RequestFilter or ResponseFilter interface—it's just Filter. When we talk about a request filter vs. a response filter, we're talking only about how you USE the filter, not the actual filter interface. As far as the Container is concerned, there is only one kind of filter—anything that implements the Filter interface.

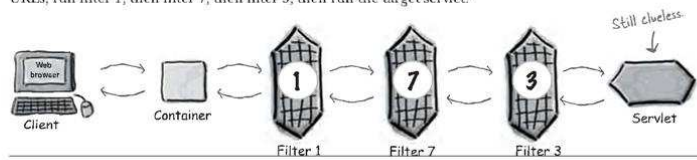
Filters are modular, and configurable in the DD

Filters can be chained together, to run one after the other. Filters are designed to be totally self-contained. A filter doesn't care which (if any) filters ran before *it* did, and it doesn't care which one will run next.*

The DD controls the order in which filters run; we'll talk about filter DD configuration a little later in the chapter.

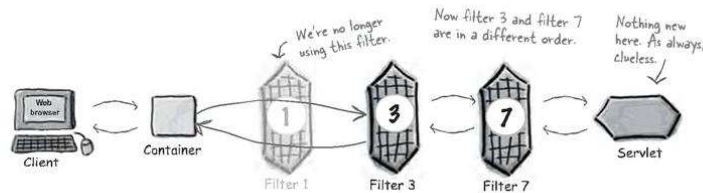
DD configuration 1:

Using the DD, you can link them together by telling the Container: "For these URLs, run filter 1, then filter 7, then filter 3, then run the target servlet."



DD configuration 2:

Then, with a quick change to the DD, you can delete and swap them with: "For these URLs, run filter 3, then filter 7, and then the target servlet."



* We're fudging a little. The deployer often *does* need to configure the order based on the consequences of the transformations performed by the filters. You wouldn't, for example, add a watermark to an image after you applied a compression filter. In that example, the watermark filter would have to do its thing before the data hits the compression filter. The point is, you as the *programmer* will not build dependencies into your code.

filters are like servlets

If filters are like servlets, then I'm guessing they must be invoked by the Container, just like servlets. They probably have their own lifecycle...



Three ways filters are like servlets

Kim's right, filters live in the Container. In many ways they're similar to their co-residents, servlets. Here are a few ways in which filters are like servlets:

The Container knows their API

Filters have their own API. When a Java class implements the **Filter interface**, it's striking a deal with the Container, and it goes from being a plain old class to being an official J2EE Filter. Other members of the filter API allow filters to get access to the `ServletContext`, and to be linked to other filters.

The Container manages their lifecycle

Just like servlets, filters have a lifecycle. Like servlets, they have `init()` and `destroy()` methods. Similar to a servlet's `doGet()/doPost()` method, filters have a `doFilter()` method.

They're declared in the DD

A web app can have **lots of filters**, and a given request can cause more than one filter to execute. The DD is the place where you declare which filters will run in response to which requests, and in which *order*.

Building the request tracking filter

Our task is to enhance the Beer application so that whenever someone requests any of the resources associated with updating recipes, we'll be able to keep track of who made the request. Here's one version of what such a filter might look like.

Filters have no idea who's going to call them or who's next in line!

```

package com.example.web;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;

public class BeerRequestFilter implements Filter {

    private FilterConfig fc;

    public void init(FilterConfig config) throws ServletException {
        this.fc = config;
    }

    public void doFilter(ServletRequest req,
        ServletResponse resp,
        FilterChain chain)
        throws ServletException, IOException {

        HttpServletRequest httpReq = (HttpServletRequest) req;

        String name = httpReq.getRemoteUser();

        if (name != null) {
            fc.getServletContext().log("User " + name + " is updating");
        }

        chain.doFilter(req, resp);

    }

    public void destroy() {
        // do cleanup stuff
    }
}

```

Filter and FilterChain are in javax.servlet

Every filter MUST implement the Filter interface

You must implement init(), usually you just save the config object

doFilter() is where you do the real work. Notice that the method doesn't take HTTP request and response objects... just regular ServletRequest and ServletResponse objects.

But we're pretty sure that we can cast the request and response to their HTTP subtypes.

This is how the next filter or servlet in line gets called - lots more on this in the next couple of pages.

You must implement destroy() but usually it's empty.

you are here ▶ 675

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

filter lifecycle

A filter's life cycle

Every filter must implement the three methods in the Filter interface: `init()`, `doFilter()`, and `destroy()`.

First there's `init()`

When the Container decides to instantiate a filter, the `init()` method is your chance to do any set-up tasks before the filter is called. The most common implementation was shown on the previous page; saving a reference to the `FilterConfig` object for later use in the filter.

`doFilter()` does the heavy lifting

The `doFilter()` method is called every time the Container determines that the filter should be applied to the current request. The `doFilter()` method takes three arguments:

- ▶ A `ServletRequest` (not an `HttpServletRequest`!)
- ▶ A `ServletResponse` (not an `HttpServletResponse`!)
- ▶ A `FilterChain`

The `doFilter()` method is your chance to implement your filter's function. If your filter is supposed to log user names to a file, do it in `doFilter()`. Want to compress the response output? Do it in `doFilter()`.

In the end there's `destroy()`

When the Container decides to remove a filter instance, it calls the `destroy()` method, giving you a chance to do any cleanup you need to do before the instance is destroyed.

^{there are no} Dumb Questions

Q: What is a `FilterChain`?

A: A `FilterChain` is the coolest thing in all of Filter-land. Filters are designed to be modular building blocks you can mix together in a variety of ways to make a combination of things happen, and the `FilterChain` is a big part of what makes this possible. *It's the thing that knows what comes next.* We already mentioned that the filters (not to mention the servlet) shouldn't know anything about the other filters involved in the request... but someone needs to know the order, and that someone is the `FilterChain`, driven by the filter elements you specify in the DD.

By the way, `FilterChain` is in the same package as `Filter`, `javax.servlet`.

Q: I noticed that in your `doFilter()` method you made this call: `chain.doFilter()`... What's a `doFilter()` doing inside a `doFilter()`? You're not gonna get all recursive on us, are you?

A: The `FilterChain` interface's `doFilter()` is a little bit different than the `Filter` interface's `doFilter()`. Here's the main difference:

The `doFilter()` method of the `FilterChain` takes care of figuring out whose `doFilter()` method to invoke next (or, if it's the end of the chain, which servlet's service() method), but the `doFilter()` method in a `Filter` actually *does* the filtering—the thing the filter was created to do.

This means a `FilterChain` can invoke EITHER a filter or a servlet, depending on whether it's the end of the chain. The end of the chain is *always* either a servlet or a JSP (which means a JSP's generated servlet, of course), assuming the Container is able to map the request URL to a servlet or JSP. (If the Container can't locate the right resource for the request, the filter is never invoked.)

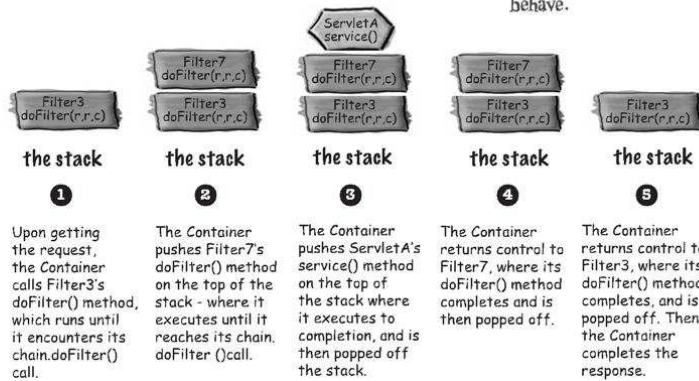
Think of filters as being “stackable”

The servlet spec doesn't dictate how the `chain.doFilter(req, resp)` method is handled inside the container. In practice, though, you can think of the process of filters chaining to each other as if they were simply method calls on a single **stack**. We know there's more going on behind the scenes in the Container, but we don't care, as long as we can predict how our filters will run, and a *conceptual* (if not physical) stack lets us do that.

This “conceptual stack” is just a way to think about filter chain invocations. We don't know (or care) how the Container actually implements this—but thinking of it this way lets you predict how your filter chain will behave.

A conceptual call stack example

In this example, a request for ServletA will be filtered by two filters, Filter3, then Filter7.



configuring filters

Declaring and ordering filters

When you configure filters in the DD, you'll usually do three things:

- ▶ Declare your filter
- ▶ Map your filter to the web resources you want to filter
- ▶ Arrange these mappings to create filter invocation sequences

Declaring a filter

```
<filter>
  <filter-name>BeerRequest</filter-name>
  <filter-class>com.example.web.BeerRequestFilter
  </filter-class>
  <init-param>
    <param-name>LogFileFileName</param-name>
    <param-value>UserLog.txt</param-value>
  </init-param>
</filter>
```

Rules for <filter>

- ▶ The <filter-name> is mandatory.
- ▶ The <filter-class> is mandatory.
- ▶ The <init-param> is optional, and you can have many.

Declaring a filter mapping to a URL pattern

```
<filter-mapping>
  <filter-name>BeerRequest</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
```

Rules for <filter-mapping>

- ▶ The <filter-name> is mandatory and it is used to link to the correct <filter> element.
- ▶ Either the <url-pattern> or the <servlet-name> element is mandatory.
- ▶ The <url-pattern> element defines which web app resources will use this filter.

Declaring a filter mapping to a servlet name

```
<filter-mapping>
  <filter-name>BeerRequest</filter-name>
  <servlet-name>AdviceServlet</servlet-name>
</filter-mapping>
```

- ▶ The <servlet-name> element defines which single web app resource will use this filter.

IMPORTANT: The Container's rules for ordering filters:

When more than one filter is mapped to a given resource, the Container uses the following rules:

- 1) ALL filters with matching URL patterns are located first. This is NOT the same as the URL mapping rules the Container uses to choose the "winner" when a client makes a request for a resource, because ALL filters that match will be placed in the chain! Filters with matching URL patterns are placed in the chain in the order in which they are declared in the DD.
- 2) Once all filters with matching URLs are placed in the chain, the Container does the same thing with filters that have a matching <servlet-name> in the DD.

678 chapter 13

Isn't THAT typical... they give us a way to filter requests coming from a *client*, and they just forget all about requests that *WE* generate through *forwards* and *request dispatches*. Geez... they treat request dispatching like it's a second-class invocation technique?!



News Flash: As of version 2.4, filters can be applied to request dispatchers

Think about it. It's great that filters can be applied to requests that come directly from the *client*. But what about resources requested from a **forward** or **include**, **request dispatch**, and/or the **error** handler? Servlet spec 2.4 to the rescue.

Declaring a filter mapping for request-dispatched web resources

```
<filter-mapping>
  <filter-name>MonitorFilter</filter-name>
  <url-pattern>*.do</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  - and / or -
  <dispatcher>INCLUDE</dispatcher>
  - and / or -
  <dispatcher>FORWARD</dispatcher>
  - and / or -
  <dispatcher>ERROR</dispatcher>
</filter>
```

Declaration Rules

- ▶ The <filter-name> is mandatory.
- ▶ Either the <url-pattern> or <servlet-name> element is mandatory.
- ▶ You can have from 0 to 4 <dispatcher> elements.
- ▶ A REQUEST value activates the filter for client requests. If no <dispatcher> element is present, REQUEST is the default.
- ▶ An INCLUDE value activates the filter for request dispatching from an include() call.
- ▶ A FORWARD value activates the filter for request dispatching from a forward() call.
- ▶ An ERROR value activates the filter for resources called by the error handler.

filter configuration exercise



Sharpen your pencil

Based on the following DD fragment, write down the sequence in which the filters will be executed for each request path. Assume Filter1 through Filter5 have been properly declared. (Answers are at the end of this chapter.)

```
<filter-mapping>
  <filter-name>Filter1</filter-name>
  <url-pattern>/Recipes/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter2</filter-name>
  <servlet-name>/Recipes/HopsList.do</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter3</filter-name>
  <url-pattern>/Recipes/Add/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter4</filter-name>
  <servlet-name>/Recipes/Modify/ModRecipes.do</servlet-name>
</filter-mapping>

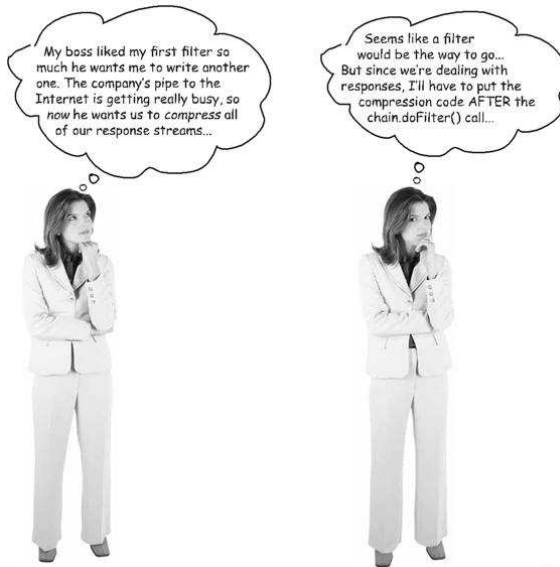
<filter-mapping>
  <filter-name>Filter5</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Request path	Filter Sequence
/Recipes/HopsReport.do	Filters:
/Recipes/HopsList.do	Filters:
/Recipes/Modify/ModRecipes.do	Filters:
/HopsList.do	Filters:
/Recipes/Add/AddRecipes.do	Filters:

Compressing output with a response-side filter

Earlier we showed a very simple *request* filter. But now we'll look at a *response* filter. Response filters are a bit trickier, but they can be incredibly useful. They let us do something to the response output after the servlet does its thing, but before the response is sent to the client. So instead of stepping in at the beginning—*before* the servlet gets the request—we step in at the end—*after* the servlet gets the request and generates a response.

Well, *sort of*... think about it. Filters are *always* invoked in the chain *before* the servlet. There's no such thing as a filter that is invoked only after the servlet. But... remember that stack picture. **The filter gets another shot at this *after* the servlet completes its work and is popped off the (virtual) stack!**



a response filter

Architecture of a response filter

Rachel is talking about the basic structure of what you put in a `doFilter()` method—first you do work related to the request, then you call `chain.doFilter()`, then finally, when the servlet (and any other filter in the chain after your filter) completes and control is returned to your original `doFilter()` method, you can do something to the response.

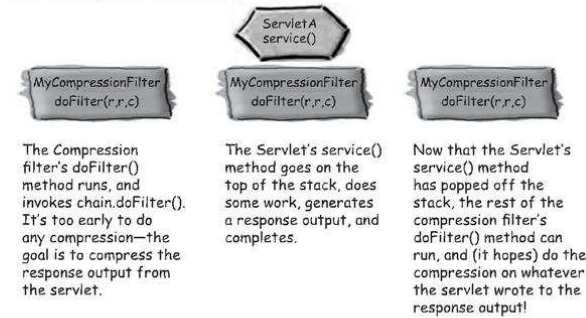
Rachel's pseudo-code for the compression filter

```
class MyCompressionFilter implements Filter {
    init();
    public void doFilter(request, response, chain) {
        // this is where request handling would go
        chain.doFilter(request, response);
        // do compression logic here
    }
    destroy();
}
```

The servlet does its work at this point

Now that the servlet is done, we can get to work on compressing the response the servlet generated...

The conceptual call stack



682 chapter 13

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

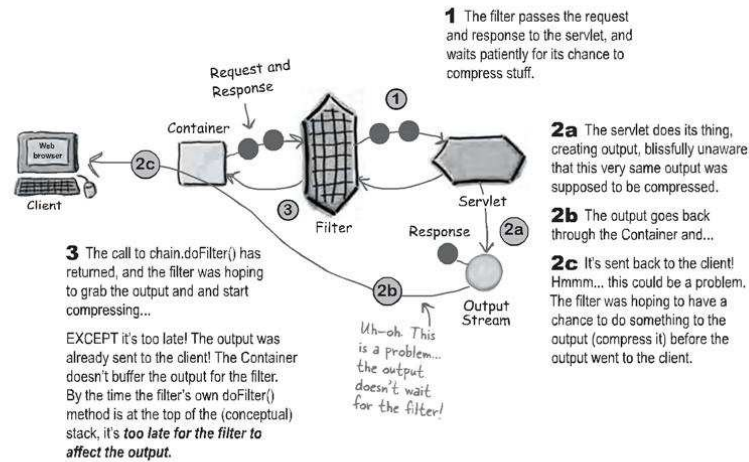
User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

But is it really that simple?

Does compressing the response really involve nothing more than waiting for the servlet to finish, then compressing the servlet's response output? After all, the filter's `doFilter()` method has a reference to the same response object that went to the servlet, so in theory, the filter should have access to the response output...

```
public void doFilter(request, response, chain) {
    // this is where request handling would go
    chain.doFilter(request, response); ① ②
    // do compression logic here ③
}
```



Chapter 13. The Power of Filters

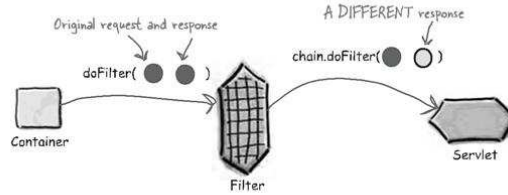
filtering the output

The output has left the building

This won't work! I can't compress something on the way out of the servlet, because it's too late. The output goes straight from the servlet back to the client. But the whole point is to compress the output, so how can I get control of the output BEFORE it goes to the client?



Think about this for a minute... the servlet actually gets the output stream or writer from the response object. What if instead of passing the REAL response object to the servlet, your filter swapped in a custom response object with an output stream that you control? Nobody said the filter has to pass the REAL response when it calls chain.doFilter().



684 chapter 13

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com

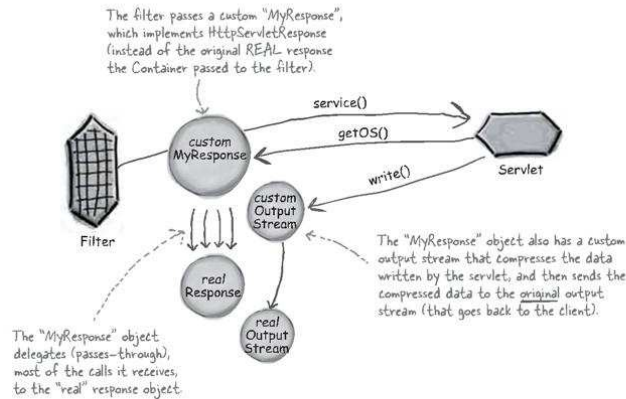
O'Reilly
Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

We can implement our OWN response

The Container already implements the `HttpServletResponse` interface; that's what you get in the `doFilter()` and `service()` methods. But to get this compression filter working, we have to make our own custom implementation of the `HttpServletResponse` interface and pass that to the servlet via the `chain.doFilter()` call. And that custom implementation has to also include a custom output stream as well, since that's the goal—to capture the output *after* the servlet writes to it but *before* it goes back to the client.



Q: Filters pass `ServletRequest` and `ServletResponse` objects to the next thing in the chain, NOT `HttpServletResponse`! So why are you talking about implementing `HttpServletResponse`?

A: Filters were designed to be generic, and so officially, you're right. If we thought one of our filters might be used in a non-web app, we'd be implementing the non-HTTP interface (`ServletResponse`), but today, the chances of someone developing non-HTTP servlets is close to zero, so we're not worried. And since `ServletResponse` is the supertype of `HttpServletResponse`, there's no problem passing an `HttpServletResponse` where a `ServletResponse` is expected.

implementing `HttpServletResponse`

HttpServletResponse is such a complicated interface... if only there were a way to avoid implementing all those methods and delegating calls to the real response...



She doesn't know about the servlet Wrapper classes

Creating your own custom `HttpServletResponse` implementation *would* be a pain. Especially when all you want to implement are just a *few* of the methods. And since `HttpServletResponse` is an interface that extends another interface, to implement your own custom response, you'd have to implement *everything* in both `HttpServletResponse` and its superinterface, `ServletResponse`.

But fortunately, someone at Sun did that for you, by creating a support convenience class that implements the `HttpServletResponse` interface. All of the methods in that class delegate the calls to the underlying real response created by the Container.

ServletResponse interface
(`javax.servlet.ServletResponse`)

```
<<interface>>
ServletResponse
getBufferSize()
setContentType()
getOutputStream()
getWriter()
// MANY more methods...
```

HttpServletResponse interface
(`javax.servlet.http.HttpServletResponse`)

```
<<interface>>
HttpServletResponse
addCookie()
addDateHeader()
addHeader()
encodeRedirectURL()
encodeURL()
sendError()
sendRedirect()
setDateHeader()
setHeader()
setStatus()
// more methods
```

Remember, to implement `HttpServletResponse` you have to implement **EVERYTHING** from both it and its superinterface `ServletResponse`.

filters and wrappers


Wrappers rock

The wrapper classes in the servlet API are awesome—they implement all the methods needed for the thing you’re trying to wrap, delegating all calls to the underlying request or response object. All you need to do is extend one of the wrappers, and override just the methods you need to do your custom work.

You’ve seen support classes in the J2SF API, of course, with things like the Listener adapter classes for GUIs. And you’ve seen them in the JSP API with the custom tag support classes. But while those support classes and these request and response wrappers are all convenience classes, the wrappers are a little different because they, well, *wrap* an object of the type they implement. In other words, they don’t just provide an *interface implementation*, they actually hold a reference to an object of the same interface type to which they delegate method calls. (By the way, this has nothing whatsoever to do with the J2SF “primitive wrapper” classes like Integer, Boolean, Double, etc.)

Creating a specialized version of a request or response is such a common approach when creating filters, that Sun has created four “convenience” classes to make the job easier:

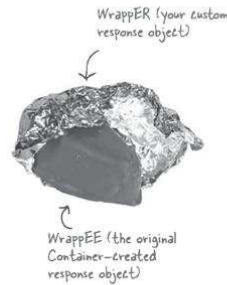
- ▶ ServletRequestWrapper
- ▶ HttpServletRequestWrapper
- ▶ ServletResponseWrapper
- ▶ HttpServletResponseWrapper



Although not explicitly listed in the official objectives, you MIGHT see “Decorator” on the exam.

If you’re familiar with regular old (non-J2EE) design patterns, then you probably recognize this wrapper classes as an example of using a Decorator pattern (although it is also sometimes called Wrapper) pattern. The Decorator/Wrapper decorates/wraps one kind of an object with an “enhanced” implementation. And by “enhanced”, we mean “adds new capabilities” while still doing everything the original wrapped thing did.

It’s like saying, “I’m just a BETTER version of the thing I’m wrapping—I do everything it does, and more.” One characteristic of a Decorator/Wrapper is that it delegates method invocations to the thing it wraps, rather than being a complete replacement.



Whenever you want to create a custom request or response object, just subclass one of the convenience request or response “wrapper” classes.

A wrapper wraps the REAL request or response object, and delegates (passes through) calls to the real thing, while still letting you do the extra things you need for your custom request or response.

a response filter

Adding a simple Wrapper to the design

Let's enhance Rachel's first pseudo-code by adding a wrapper.

Compression filter design, version 2 (pseudocode)

```

class CompressionResponseWrapper extends HttpServletResponseWrapper {
    // override any methods you want to customize
}

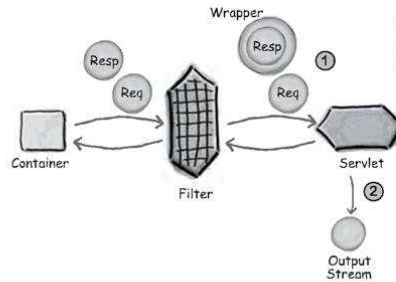
class MyCompressionFilter implements Filter {
    public void init(FilterConfig cfg) { }
    public void doFilter(request, response, chain) {
        CompressionResponseWrapper wrappedResp
        = new CompressionResponseWrapper(response);
        chain.doFilter(request, wrappedResp);
        // do compression logic here
    }
    public void destroy() { }
}
    
```

Let's subclass this wrapper class for our own evil purposes...

We'll be doing some real overriding in a few pages!

The act of "wrapping" the response with our custom Wrapper class.

Now we send this along down the filter chain. None of the components down the chain will ever know that the response object they got was a custom job.



1 The filter passes the request object and a custom response object to the servlet.

2 Since we didn't override any methods in the Wrapper, the output stream isn't affected... yet.

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com

O'Reilly

User number: 729515 Copyright 2007, Safari Books Online, LLC.

Print Publication Date: 8/1/2004
 This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

Add an output stream Wrapper

Let's add a second Wrapper...

Compression filter design, version 3 (pseudocode)

```

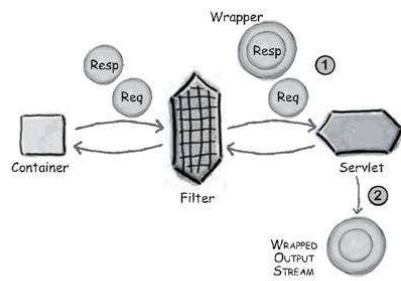
class CompressionResponseWrapper extends HttpServletResponseWrapper {
    public ServletOutputStream getOutputStream() throws... {
        ...
        servletGzipOS = new GzipGOS(resp.getOutputStream());
        return servletGzipOS;
    }
    // maybe override other methods
}

class MyCompressionFilter implements Filter {
    public void init(FilterConfig cfg) { }
    public void doFilter(request, response, chain) {
        CompressionResponseWrapper wrappedResp
        = new CompressionResponseWrapper(response);
        chain.doFilter(request, wrappedResp);
        // do compression logic here
    }
    public void destroy() { }
}
    
```

Override this method to return a custom output stream.

"Wrapping" the ServletOutputStream with our custom ServletOutputStream Wrapper class. For now let's assume Gzip ServletOutputStream extends ServletOutputStream.

Return a "special" ServletOutputStream to whoever asks for one.



1 The filter passes the request object and a custom response object to the servlet. The custom response has a special getOutputStream method.

2 When the servlet asks for an it doesn't KNOW that it will get a "special" output stream.

response compression filter

The real compression filter code

Time to code. We end this chapter by looking at the code for both the compression filter and the wrapper it uses. We're expanding from the previous discussion, and while there is some new stuff here, it's mostly just plain Java code.

This filter provides a mechanism to compress the response body content. This type of filter would commonly be applied to any text content such as HTML, but not to most media formats such as PNG or MPEG, because they are already compressed.

```
package com.example.web;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.zip.GZIPOutputStream;

public class CompressionFilter implements Filter {

    private ServletContext ctx;
    private FilterConfig cfg;

    public void init(FilterConfig cfg)
        throws ServletException {
        this.cfg = cfg;
        ctx = cfg.getServletContext();
        ctx.log(cfg.getFilterName() + " initialized.");
    }

    public void doFilter(ServletRequest req,
        ServletResponse resp,
        FilterChain fc)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;

        String valid_encodings = request.getHeader("Accept-Encoding");
        if ( valid_encodings.indexOf("gzip") > -1 ) {

            CompressionResponseWrapper wrappedResp
                = new CompressionResponseWrapper(response);
        }
    }
}
```

The init method saves the config object and a quick reference to the servlet context object (for logging purposes).

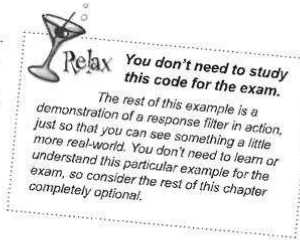
The heart of this filter wraps the response object with a Decorator that wraps the output stream with a compression I/O stream.

Compression of the output stream is performed if and only if the client includes an Accept-Encoding header (specifically, for gzip).

Does the client accept GZIP compression?

If so, wrap the response object with a compression wrapper.

690 chapter 13



Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

Compression filter code, cont.

Debugging Tip!

To test this filter, comment out this line of code. You should see illegible, compressed data in your browser.

```

wrappedResp.setHeader("Content-Encoding", "gzip");
fc.doFilter(request, wrappedResp);

GZIPOutputStream gzos = wrappedResp.getGZIPOutputStream();
gzos.finish();
ctx.log(cfg.getFilterName() + ": finished the request.");
} else {
    ctx.log(cfg.getFilterName() + ": no encoding performed.");
}
}

public void destroy() {
    // nulling out my instance variables
    cfg = null;
    ctx = null;
}
}
    
```

Declare that the response content is being GZIP encoded.

Chain to the next component

A GZIP compression stream must be "finished", which also flushes the GZIP stream buffer, and sends all of its data to the original response stream.

The container handles the rest of the work.

"Off the path"

Compression meets HTTP

How does the server know it can send compressed data? How does the browser know when it's getting compressed data? It turns out that HTTP is "compression-aware"; here's how it works:

- ▶ One of the headers that the browser sends ("Accept-Encoding: gzip"), tells the server about the browser's capabilities for dealing with different types of content.
- ▶ If the server sees that the browser can deal with compressed data, it will perform the compression, and add a header ("Content-Encoding: gzip"), to the response.
- ▶ When the browser receives the response, the "Content-Encoding: gzip" header tells the browser to de-compress the data before it is displayed.

So far so good. How hard can a little thing like a wrapper be?
(Famous last words...)



response compression wrapper

Compression wrapper code

We looked at the Compression filter; now let's take a look at the wrapper it uses. This is one of the most complicated topics in all of servlet-dom, so don't panic if you don't grok it the first time.

This response wrapper decorates the original response object by adding a compression decorator on the original servlet output stream.

```

package com.example.web;

// Servlet imports
import javax.servlet.http.*;
import javax.servlet.*;
// I/O imports
import java.io.*;
import java.util.zip.GZIPOutputStream;

class CompressionResponseWrapper extends HttpServletResponseWrapper {

    private GZIPOutputStream servletGzipOS = null;
    private PrintWriter pw = null;

    CompressionResponseWrapper(HttpServletResponse resp) {
        super(resp);
    }

    public void setContentLength(int len) {}

    public GZIPOutputStream getGZIPOutputStream() {
        return this.servletGzipOS.internalGzipOS;
    }

```

← The compressed output stream for the servlet response.
 ← The PrintWriter object to the compressed output stream.
 ← The super constructor performs the Decorator responsibility of storing a reference to the object being decorated, in this case the HTTP response object.
 ← Ignore this method—the output will be compressed.
 ← This decorator method, used by the filter, gives the compression filter a handle on the GZIP output stream so that the filter can “finish” and flush the GZIP stream.

692 chapter 13

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

Compression wrapper code, cont.

```

private Object streamUsed = null;
public ServletOutputStream getOutputStream() throws IOException {
    if ((streamUsed != null) && (streamUsed != pw)) {
        throw new IllegalStateException();
    }
    if ( servletGzipOS == null ) {
        servletGzipOS
            = new GZIPServletOutputStream(getResponse()
                .getOutputStream());
        streamUsed = servletGzipOS;
    }
    return servletGzipOS;
}

public PrintWriter getWriter() throws IOException {
    if ((streamUsed != null) && (streamUsed != servletGzipOS)) {
        throw new IllegalStateException();
    }
    if ( pw == null ) {
        servletGzipOS
            = new GZIPServletOutputStream(getResponse()
                .getOutputStream());
        OutputStreamWriter osw
            = new OutputStreamWriter(servletGzipOS,
                getResponse().getCharacterEncoding());
        pw = new PrintWriter(osw);
        streamUsed = pw;
    }
    return pw;
}

```

Provide access to a decorated servlet output stream.

Allow the servlet to access a servlet output stream, only if the servlet has not already accessed the print writer.

Wrap the original servlet output stream with our compression servlet output stream.

Provide access to a decorated print writer.

Allow the servlet to access a print writer, only if the servlet has not already accessed the servlet output stream.

To make a print writer, we have to first wrap the servlet output stream and then wrap the compression servlet output stream in two additional output stream decorators: OutputStreamWriter which converts characters into bytes, and then a PrintWriter on top of the OutputStreamWriter object.

response output decorator

Compression wrapper, helper class code

This helper class is a Decorator on the `ServletOutputStream` abstract class which delegates the real work of compressing the generated content using a standard GZIP output stream.

There is only one abstract method in the `ServletOutputStream` that this Decorator must implement: `write(int)`. This is where all of the delegation magic occurs!

```
class GZIPServletOutputStream extends ServletOutputStream {

    GZIPOutputStream internalGzipOS; ← Keep a reference to the raw GZIP stream. This
    instance variable is package-private to allow the
    compression response wrapper access to this variable.

    /** Decorator constructor */
    GZIPServletOutputStream(ServletOutputStream sos) throws IOException {
        this.internalGzipOS = new GZIPOutputStream(sos);
    }

    public void write(int param) throws java.io.IOException {
        internalGzipOS.write(param);
    }
}
```

← This method implements the compression decoration by delegating the `write()` call to the GZIP compression stream, which is wrapping the original `ServletOutputStream`, (which in turn is ultimately wrapping the TCP network output stream to the client).

694 chapter 13

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

**ANSWERS**

Write down the sequence in which the filters will be executed for each request path. Assume Filter1 - Filter5 have been properly declared.

```
<filter-mapping>
  <filter-name>Filter1</filter-name>
  <url-pattern>/Recipes/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter2</filter-name>
  <servlet-name>/Recipes/HopsList.do</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter3</filter-name>
  <url-pattern>/Recipes/Add/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter4</filter-name>
  <servlet-name>/Recipes/Modify/ModRecipes.do</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>Filter5</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Request path	Filter Sequence
/Recipes/HopsReport.do	Filters: 1, 5
/Recipes/HopsList.do	Filters: 1, 5, 2
/Recipes/Modify/ModRecipes.do	Filters: 1, 5, 4
/HopsList.do	Filters: 5
/Recipes/Add/AddRecipes.do	Filters: 1, 3, 5

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

mock exam



- 1 Which are true about filters? (Choose all that apply.)
- A. A filter can act on only the request or response object, not both.
 - B. The **destroy** method is always a container callback method.
 - C. The **doFilter** method is always a container callback method.
 - D. The only way a filter can be invoked is through a declaration in the DD.
 - E. The next filter in a filter chain can be specified either by the previous filter or in the DD.

- 2 Which are true about declaring filters in the DD? (Choose all that apply.)
- A. Unlike servlets, filters CANNOT declare initialization parameters.
 - B. Filter chain order is always determined by the order the elements appear in the DD.
 - C. A class that extends an API request or response wrapper class must be declared in the DD.
 - D. A class that extends an API request or response wrapper class is using the Intercepting Filter pattern.
 - E. Filter chain order is affected by whether filter mappings are declared via `<url-pattern>` or via `<servlet-name>`.

696 chapter 13

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

3

Given this method in an otherwise properly defined `Filter` implementation:

```
20. public void doFilter (ServletRequest req,
21.                     ServletResponse response,
22.                     FilterChain chain)
23.     throws IOException, ServletException {
24.     HttpServletRequest request = (HttpServletRequest) req;
25.     HttpSession session = request.getSession();
26.     Object user = session.getAttribute("user");
27.     if (user != null) {
28.         UserRequest ureq = new UserRequest(request, user);
29.         chain.doFilter(ureq, response);
30.     } else {
31.         RequestDispatcher rd = request.getRequestDispatcher("/login.jsp");
32.         rd.forward(request, response);
33.     }
```

Which is true?

- A. An exception will always be thrown if line 31 executes.
- B. Line 28 is invalid because `request` must be passed as the first argument.
- C. This line: `chain.doFilter(request, response)` must be inserted somewhere in the `else` block.
- D. This method does not properly implement `Filter.doFilter()` because the method signature is incorrect.
- E. None of the above.

mock exam

4

Given a partial deployment descriptor:

```
11. <filter>
12.   <filter-name>My Filter</filter-name>
13.   <filter-class>com.example.MyFilter</filter-class>
14. </filter>
15. <filter-mapping>
16.   <filter-name>My Filter</filter-name>
17.   <url-pattern>/my</url-pattern>
18. </filter-mapping>
19. <servlet>
20.   <servlet-name>My Servlet</servlet-name>
21.   <servlet-class>com.example.MyServlet</servlet-class>
22. </servlet>
23. <servlet-mapping>
24.   <servlet-name>My Servlet</servlet-name>
25.   <url-pattern>/my</url-pattern>
26. </servlet-mapping>
```

Which is true? (Choose all that apply.)

- A. The file is invalid because the URL pattern `/my` is mapped to both a servlet and a filter.
- B. The file is invalid because neither the servlet name nor the filter name is allowed to contain spaces.
- C. The filter `MyFilter` will be invoked after the `MyServlet` servlet for each request that matches the pattern `/my`.
- D. The filter `MyFilter` will be invoked before the `MyServlet` servlet for each request that matches the pattern `/my`.
- E. The file is invalid because the `<filter>` element must contain a `<servlet-name>` element that defines which servlet the filter should be applied to.

698 *chapter 13*

- 5 Which about filters are true? (Choose all that apply.)
- A. Filters may be used to create request or response wrappers.
 - B. Wrappers may be used to create request or response filters.
 - C. Unlike servlets, all filter initialization code should be placed in the constructor since there is no `init()` method.
 - D. Filters support an initialization mechanism that includes an `init()` method that is guaranteed to be called before the filter is used to handle requests.
 - E. A filter's `doFilter()` method must call `doFilter()` on the input `FilterChain` object in order to ensure that all filters have a chance to execute.
 - F. When calling `doFilter()` on the input `FilterChain`, a filter's `doFilter()` method must pass in the same `ServletRequest` and `ServletResponse` objects that were passed into it.
 - G. A filter's `doFilter()` may block further request processing.
-
- 6 Which are true about the servlet Wrapper classes? (Choose all that apply.)
- A. They provide the only mechanism for wrapping `ServletResponse` objects.
 - B. They can be used to decorate classes that implement `Filter`.
 - C. They can be used even when the application does NOT support HTTP.
 - D. The API provides wrappers for `ServletRequest`, `ServletResponse`, and `FilterChain` objects.
 - E. They implement the Intercepting Filter pattern.
 - F. When you subclass a wrapper class, you must override at least one of the wrapper class's methods.

mock answers



- 1 Which are true about filters? (Choose all that apply.) (Servlet v2.4 section 6)
- A. A filter can act on only the request or response object, not both.
 - B. The **destroy** method is always a container callback method.
 - C. The **doFilter** method is always a container callback method. -Option C is incorrect, doFilter is both a callback and an inline method.
 - D. The only way a filter can be invoked is through a declaration in the DD.
 - E. The next filter in a filter chain can be specified either by the previous filter or in the DD. -Option E is incorrect, the order of filter execution is always determined in the DD.
-
- 2 Which are true about declaring filters in the DD? (Choose all that apply.) (Servlet v2.4 section 6)
- A. Unlike servlets, filters CANNOT declare initialization parameters.
 - B. Filter chain order is always determined by the order the elements appear in the DD. -Option B is incorrect, because <url-pattern> mappings will be chained before <servlet-name> mappings.
 - C. A class that extends an API request or response wrapper class must be declared in the DD.
 - D. A class that extends an API request or response wrapper class is using the Intercepting Filter pattern. -Option D is incorrect, wrappers are examples of the Decorator pattern.
 - E. Filter chain order is affected by whether filter mappings are declared via **<url-pattern>** or via **<servlet-name>**.

700 chapter 13

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

3 Given this method in an otherwise properly defined `Filter` implementation:

```

20. public void doFilter (ServletRequest req,
21.                     ServletResponse response,
22.                     FilterChain chain)
23.     throws IOException, ServletException {
24.     HttpServletRequest request = (HttpServletRequest) req;
25.     HttpSession session = request.getSession();
26.     Object user = session.getAttribute("user");
27.     if (user != null) {
28.         UserRequest ureq = new UserRequest(request, user);
29.         chain.doFilter(ureq, response);
30.     } else {
31.         RequestDispatcher rd = request.getRequestDispatcher("/login.jsp");
32.         rd.forward(request, response);
33.     }

```

Which is true?

- A. An exception will always be thrown if line 31 executes. *-Option A is incorrect as it is valid for a filter to forward a request*
- B. Line 28 is invalid because `request` must be passed as the first argument. *-Option B is incorrect because it is valid for a filter to wrap a request (note that UserRequest must implement ServletRequest).*
- C. This line: `chain.doFilter(request, response)` must be inserted somewhere in the `else` block. *-Option C is incorrect because the doFilter method is NOT required to call chain.doFilter().*
- D. This method does not properly implement `Filter.doFilter()` because the method signature is incorrect. *-Option D is incorrect because the method signature is correct.*
- E. None of the above.

mock answers

- 4 Given a partial deployment descriptor: (Servlet v2.4 pg 53)
- ```

11. <filter>
12. <filter-name>My Filter</filter-name>
13. <filter-class>com.example.MyFilter</filter-class>
14. </filter>
15. <filter-mapping>
16. <filter-name>My Filter</filter-name>
17. <url-pattern>/my</url-pattern>
18. </filter-mapping>
19. <servlet>
20. <servlet-name>My Servlet</servlet-name>
21. <servlet-class>com.example.MyServlet</servlet-class>
22. </servlet>
23. <servlet-mapping>
24. <servlet-name>My Servlet</servlet-name>
25. <url-pattern>/my</url-pattern>
26. </servlet-mapping>

```

Which is true? (Choose all that apply.)

- A. The file is invalid because the URI pattern `/my` is mapped to both a servlet and a filter. -Option A is incorrect because this is proper syntax used to map a filter to the same pattern as a servlet.
- B. The file is invalid because neither the servlet name nor the filter name is allowed to contain spaces. -Option B is incorrect because there is no such restriction.
- C. The filter `MyFilter` will be invoked after the `MyServlet` servlet for each request that matches the pattern `/my`. -Option C is incorrect because filters are executed before servlets, not after.
- D. The filter `MyFilter` will be invoked before the `MyServlet` servlet for each request that matches the pattern `/my`.
- E. The file is invalid because the `<filter>` element must contain a `<servlet-name>` element that defines which servlet the filter should be applied to. -Option E is incorrect because either a `<servlet-name>` element or a `<url-pattern>` may be used within a `<filter-mapping>` element.

702 chapter 13

## Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly

Print Publication Date: 8/1/2004

User number: 729515 Copyright 2007, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

*filters and wrappers*

- 5 Which about filters are true? (Choose all that apply.) (Servlet v2.4 pg 51)
- A. Filters may be used to create request or response wrappers.
  - B. Wrappers may be used to create request or response filters. -Option B is incorrect because the terminology is reversed.
  - C. Unlike servlets, all filter initialization code should be placed in the constructor since there is no `init()` method. -Option C is incorrect because there is an `init()` method that should be used for filter initialization.
  - D. Filters support an initialization mechanism that includes an `init()` method that is guaranteed to be called before the filter is used to handle requests.
  - E. A filter's `doFilter()` method must call `doFilter()` on the input `FilterChain` object in order to ensure that all filters have a chance to execute. -Option E is incorrect because calling `doFilter()` is not necessary if a filter wishes to block further request processing.
  - F. When calling `doFilter()` on the input `FilterChain`, a filter's `doFilter()` method must pass in the same `ServletRequest` and `ServletResponse` objects that were passed into it. -Option F is incorrect because the filter may choose to "wrap" the request or the response object and pass those instead.
  - G. A filter's `doFilter()` may block further request processing.

- 6 Which are true about the servlet Wrapper classes? (Choose all that apply.) (API)
- A. They provide the only mechanism for wrapping `ServletResponse` objects. -Option A is incorrect because you can create your own wrapper class.
  - B. They can be used to decorate classes that implement `Filter`. -Option B is incorrect because these classes are used to wrap requests and responses.
  - C. They can be used even when the application does NOT support HTTP.
  - D. The API provides wrappers for `ServletRequest`, `ServletResponse`, and `FilterChain` objects. -Option D is incorrect because the API does NOT provide a `FilterChain` wrapper.
  - E. They implement the Intercepting Filter pattern. -Option E is incorrect because these wrappers implement the Decorator pattern.
  - F. When you subclass a wrapper class, you must override at least one of the wrapper class's methods.

Chapter 13. The Power of Filters

Head First Servlets and JSP By Bert Bates, Kathy Sierra, Bryan Basham ISBN: 0596005407 Publisher: Prepared for Augusto Jaramillo Forcada, Safari ID: augustojf.cv@gmail.com O'Reilly  
 Print Publication Date: 8/1/2004 User number: 729515 Copyright 2007, Safari Books Online, LLC.  
 This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.