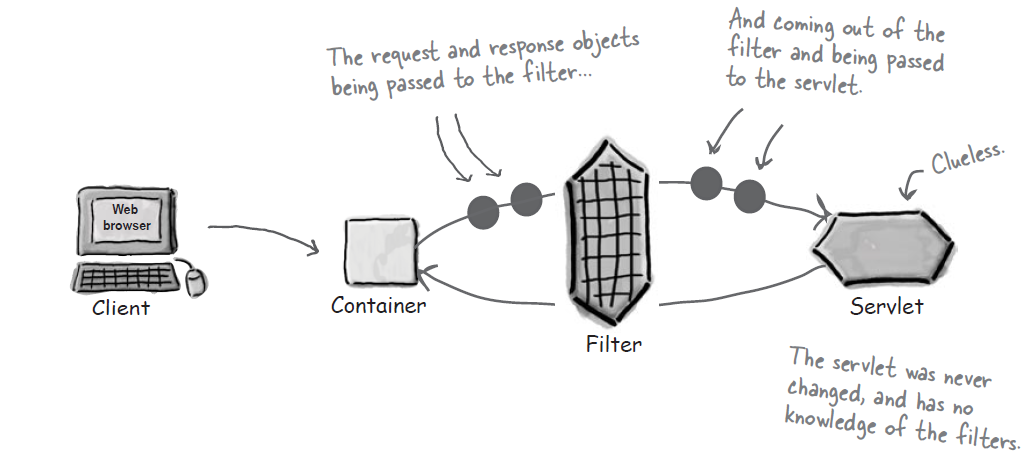
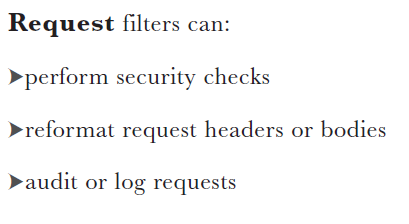
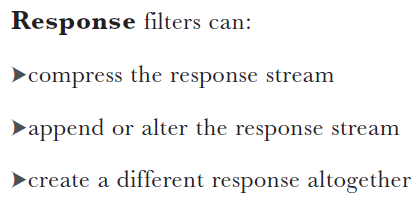
**­­­Filters**

* **Filters let you intercept the request.** And if you can intercept the *request*, you can also control the *response*.
* **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.
* 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 URL patterns. So it’s the deployer, not the programmer, who decides which subset of requests or responses should be processed by which filters.



Things that a Filter Can do:



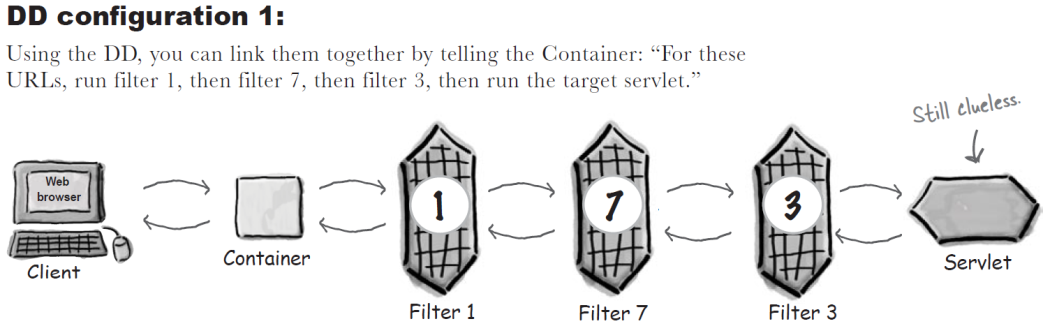


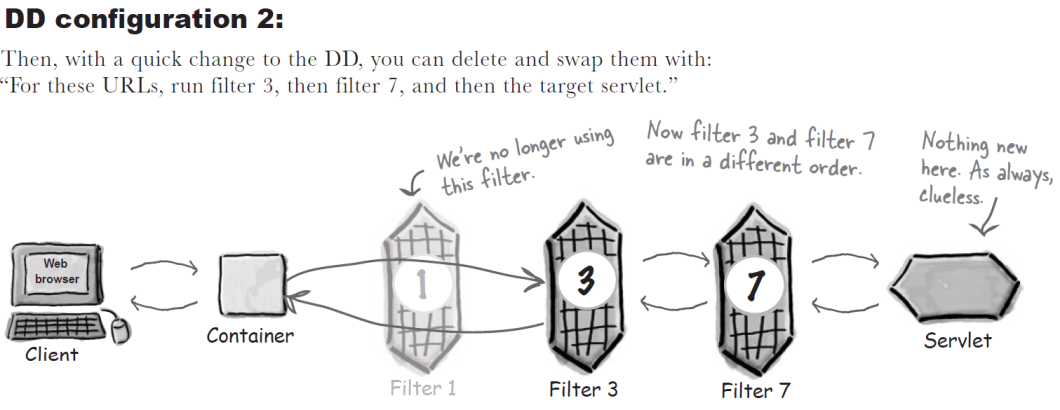
**Note:**

*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 inter- face. 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;

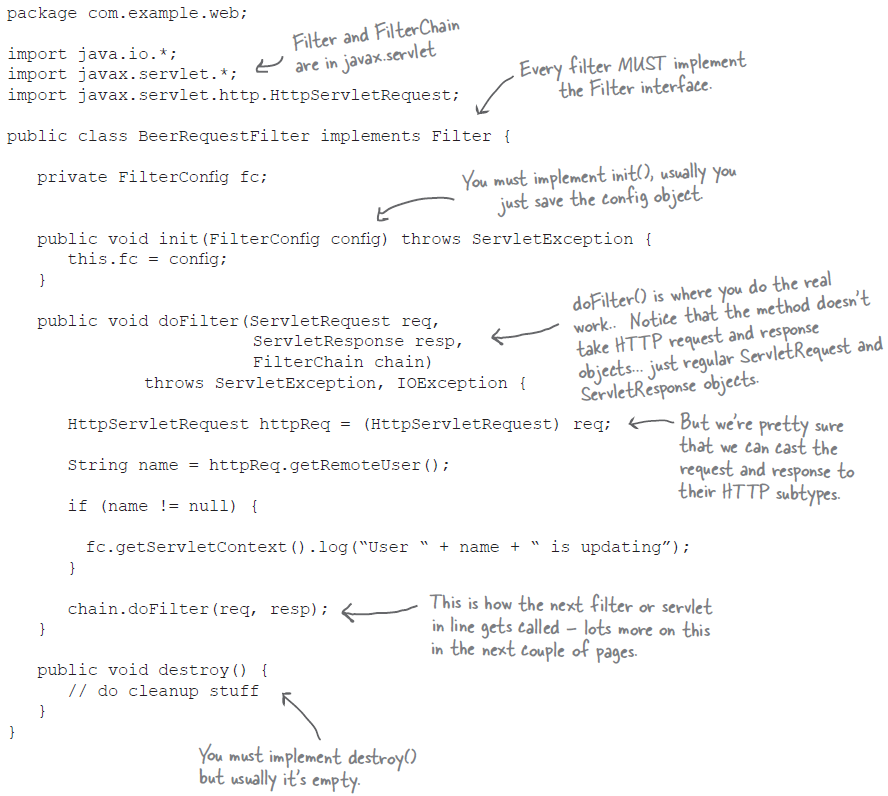
****

****

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 have their own API. When a Java class implements the **Filter interface**, it’s a deal with the Container, and it goes from being a plain old class to being an official J2EE Filter.
* 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.
* 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**

****

****

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

****

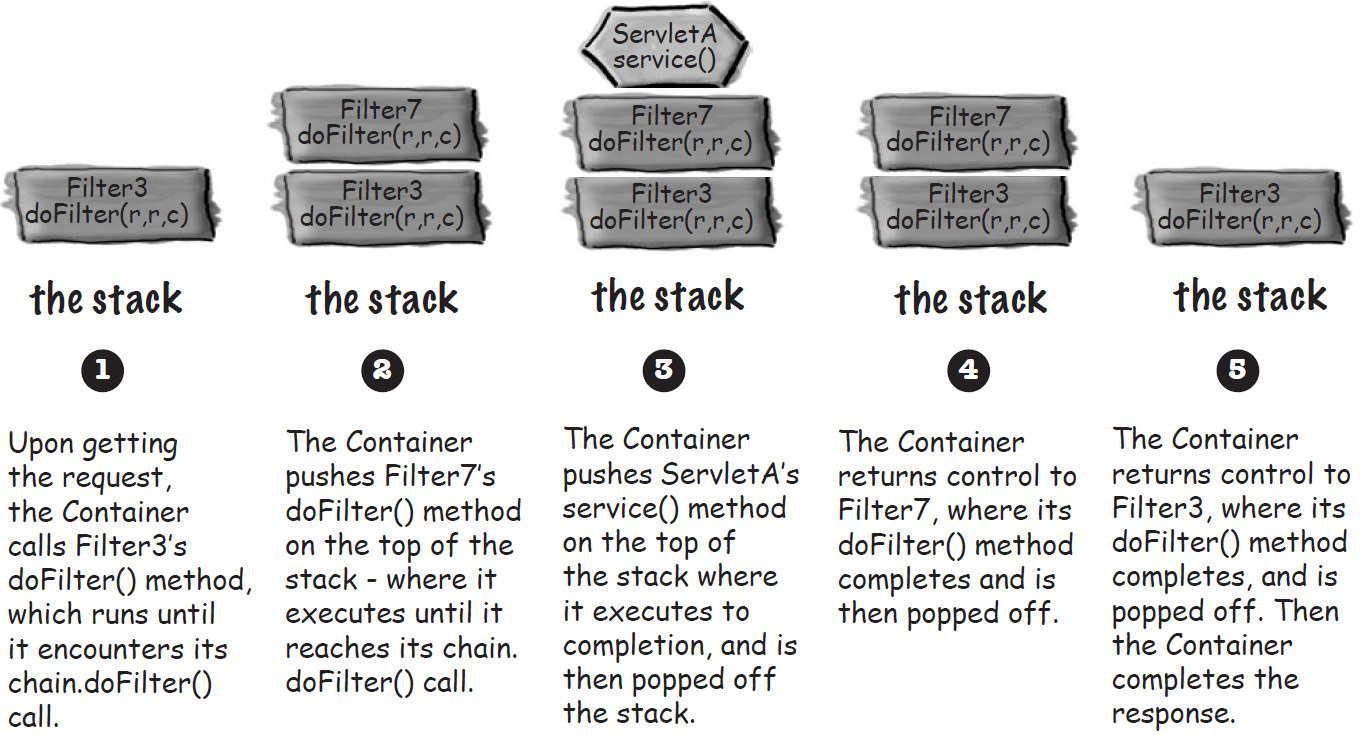
* When the Container decides to remove a filter instance, it calls the **destroy()** method, giving you a chance to do any cleanup you need to do before the instance is destroyed.

**What is a *FilterChain?*** 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.* 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. FilterChain is in the same package as Filter, *javax.servlet*.

**c*hain.doFilter()*... What’s a doFilter() doing inside a doFilter()?** The ***FilterChain*** interface’s doFilter() is a little bit different than the **Filter** interface’s doFilter().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.)

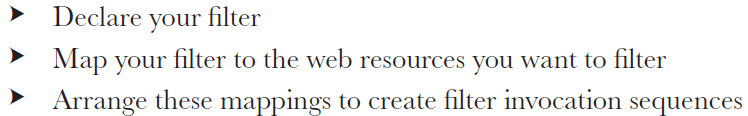
Think of filters as being “stackable”

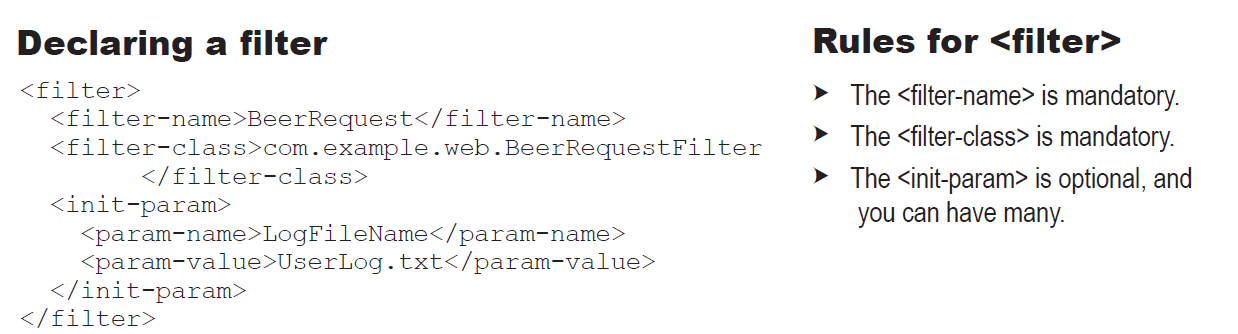
This “conceptual stack” is just a way to think about filter chain invocations. We don’t know (or care) how the Container actually implements this—but thinking of it this way lets you predict how your filter chain will behave.



Declaring and ordering filters

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







**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.