3

The Web Container Model

CERTIFICATION OBJECTIVES

In this chapter we examine more closely that largely overlooked piece of software that looks after your requests, responses, and servlets: the web container. As you'll have gathered from looking at the J2EE API documentation for javax.servlet and related packages, there are a large number of classes and interfaces implicated in web applications. Some you have to write yourself as a developer: implementing interfaces in the API or extending useful base classes (such as GenericServlet). However, there are many more interfaces that you never have to implement—because your web container provider has done it for you. That's the deal if you are creating a J2EE web container: You have to understand the servlet specifications in detail and provide suitably compliant classes. That's not to mention orchestrating the runtime environment: making sure that instances of servlets are created in the right circumstances and that life cycle methods are called at the right time.

Fortunately, you are a web *component* developer, not a web *container* developer. So you can take for granted much of what is built into the web container (both for the exam and for real development work). You are a consumer of what the web container provides—and have to understand the infrastructure only insofar as it affects your own business applications.

That's the focus of the exam objectives we explore in this chapter. We start with the ServletContext interface. You never build a class implementing this interface, nor do you instantiate such a class at runtime: The web container does that for you. All you have to do is to understand when and why a ServletContext is available—and what you can do with it (attaching initialization parameters for your web application, for example). We go on to dissect three different scopes maintained by the web container—request, session, and context—and examine how you can attach information to these scopes. We'll also see how this information is affected by the multithreaded nature of web containers. The mechanism of request dispatching will be laid bare: how one servlet can take advantage of other servlets and other web resources. And we'll finish the chapter with a look at filters, a way of trapping and processing requests and responses going to and from a target web resource.

CERTIFICATION OBJECTIVE 3.01

ServletContext (Exam Objective 3.1)

For the ServletContext initialization parameters: write servlet code to access initialization parameters; and create the deployment descriptor elements for declaring initialization parameters.

We have already met the ServletContext, and in this chapter—describing the web container model—we explore its remaining secrets. The ServletContext most closely represents the web application itself—or, more correctly, provides a set of services for the web application to work with the web container.

We'll start this chapter by looking at one of the fundamentally useful aspects of the ServletContext: the ability to set up initialization parameters that are then available to every servlet and JSP in your web application.

ServletContext Initialization Parameters

What if you want some fundamental information available to all the dynamic resources (servlets, JSPs) within your web application? We've already seen how to provide initialization information for servlets by using servlet initialization parameters in the deployment descriptor and by using the `getInitParameter(String parmName)` method. But a servlet initialization parameter is accessible only from its containing servlet. For web application level, we have ServletContext parameters.

Setting Up the Deployment Descriptor

You can have as many ServletContext initialization parameters as you wish—none, one, fifty, or more. The listing below shows two ServletContext initialization parameters as the only things in the deployment descriptor:

```
<web-app>
  <context-param>
    <param-name>machineName</param-name>
    <param-value>GERALDINE</param-value>
  </context-param>
  <context-param>
    <param-name>secretParameterFile</param-name>
    <param-value>/WEB-INF/xml/secretParms.xml</param-value>
  </context-param>
</web-app>
```

How does this compare with servlet initialization parameters (covered in Chapter 1)? You can see that the `<param-name>` and `<param-value>` tag pairings are identical between Servlet and ServletContext. If you are good at deciphering the XSD used to validate the deployment descriptor—not that you need to be for the exam!—this will come as no surprise, because this "block" of `<param-name>` and `<param-value>` has a unifying element description that occurs wherever a parameter name/value pairing is required. But note that the parent element is not the same.

Whereas a servlet has <init-param> to encase the parameter name/value pairing, ServletContext has <context-param>. And whereas—for servlets—<init-param> has <servlet> as its parent, a ServletContext's <context-param> elements sit directly in the root element, <web-app>. This makes perfect sense, because servlet initialization parameters belong to a servlet, whereas context initialization parameters belong to a web application.

Writing Code to Retrieve ServletContext Initialization Parameters

Furthermore, the coding approach for getting hold of initialization parameters is nearly identical. In fact, the method signatures involved—`getInitParameter (String parmName)` and `getInitParameterNames()`—are identical. Here's servlet code to retrieve the ServletContext parameters we set up in the deployment descriptor in the previous section. (It's a fragment: You'll have to imagine this is part of a `doGet()` method and that the response's PrintWriter has already been retrieved to a variable called `out`.)

```
ServletContext sc = getServletContext();
String database = sc.getInitParameter("machineName");
String secret = sc.getInitParameter("secretParameterFile");
out.write("<BR />The machine name is: " + database);
out.write("<BR />The secret parameter file is: " + secret);
```

This might yield output in the web page looking like the following:

```
The machine name is: GERALDINE
The secret parameter file is: /WEB-INF/xml/secretParms.xml
```

If you wish to recover the names of all the parameters set up for the Servlet Context, then look up the values for those names, the snippet of servlet code below will do the trick:

```
ServletContext sc = getServletContext();
Enumeration e = sc.getInitParameterNames();
while (e.hasMoreElements()) {
  String paramName = (String) e.nextElement();
  out.write("<BR />Parameter name <B>" + paramName
    + "</B> has the value <I>" + sc.getInitParameter(paramName)
    + "</I>");
}
```

When I run this code in Tomcat, I get the following web page output:

```
Parameter name secretParameterFile has the value /WEB-INF/xml/secretParms.xml
Parameter name machineName has the value GERALDINE
```

You might have noticed that this lists the parameters in the opposite order from their setup in the deployment descriptor, where the *machineName* parameter came first. Let this be a warning to you: In this case and many more, you can't rely on deployment descriptor order.

exam Watch

ServletContext initialization parameters are not tricky—it might be hard to imagine any difficult questions arising! Watch out, though, for questions that focus on what you get back from ServletContext methods when the initialization parameters are wrong or missing. `getInitParameter(String parmName)` hands back a* null *String reference if the parameter is not recognized. If there are no context parameters at all, `getInitParameterNames()` hands back an empty Enumeration (i.e., a non-*null*, bona fide Enumeration reference—but the Enumeration has no elements). What you* don't *get is any kind of exception being thrown from these methods—at least, not under these normal circumstances, where parameters are unrecognized or missing.

on the Job

I don't put all my system parameters and other meta-information in ServletContext initialization parameters. I put most data of that sort in files on the file system—maybe in simple properties files, or XML files for more sophisticated data. I would locate the files under a directory located within the WEB-INF directory, then use a single ServletContext initialization parameter to hold this location, relative to the web context. For properties files, I would use methods such as `getResourceAsStream()` to return an InputStream I could `load()` into a Properties object. (XML files might need a more handcrafted approach, dependent on the Java classes that will receive the XML-described information.) The properties object itself could be set (and subsequently accessed) as a ServletContext attribute—something we'll learn about in the next section of this chapter. Why do I do this? Because while ServletContext initialization parameters are a great idea, they're not as flexible as data held in files. Files can be altered in situ—and, if necessary, you can

arrange a call to a servlet that will reload the file values into Java variables in your web application dynamically, without recourse to closing the application down. A change to a ServletContext parameter necessitates redeploying the deployment descriptor file and restarting the web application to cause the parameters to be re-read.

EXERCISE 3-1

Using the ServletContext to Discover and Read a Properties File

In this exercise, you'll set up a ServletContext initialization parameter that holds the location and name of a file. You'll write servlet code to use this parameter to locate and read the file, holding the individual properties within it as ServletContext attributes (this gives you a taste of what is to come in the next section!). Finally, you'll write more servlet code to access those properties at a later point in the servlet life cycle.

Each exercise from now on will involve a web application of its own. I suggest you work in the following way, as described more fully in Appendix B:

- Create a directory that follows the naming structure of the chapter and exercise number—ex0301 for this exercise.
- Create a web application directory structure underneath this. This structure should contain the /WEB-INF directory, the /WEB-INF/classes directory, and the deployment descriptor file web.xml in the /WEB-INF directory.
- Create a package structure under /WEB-INF/classes of your own (you don't have to put your servlets into packages).
- Create your source files directly in your package directories (or directly in /WEB-INF/classes if there are no package directories).
- Compile the source in place so that the class files appear in the same directories as the source directories.

For this exercise, there's a solution in the CD in the file sourcecode/ch03/ex0301.war—check there if you get stuck.

Set Up the Deployment Descriptor

1. In web.xml, set up two context initialization parameters, one named "propsFileName," with a value of /WEB-INF, and the other "propsFile

Location," with a value of ex0301.properties. Refer to the section above to see how to set up context initialization parameters.

2. Declare a servlet named ContextInitParms, with a suitable servlet mapping. Ensure that it loads on start up of the server. Refer to Chapter 2 to refresh yourself on `<servlet>` element setup if you need to.

Set Up the Properties File

3. Create a file called "ex0301.properties" in the /WEB-INF directory.
4. Use a text editor to put the following contents into the properties file:

```
application_name=EX0301 ContextInitParms
```

Write the ContextInitParms Servlet

5. Create a Java source file ContextInitParms.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
6. In your ContextInitParms servlet, override the `init()` method (inherited from GenericServlet)—that's the `init()` without any parameters.
7. In the `init()` method, return the context initialization parameters you set up in step 1 of the exercise into local String variables.
8. Still in the `init()` method, concatenate the Strings retrieved in step 7, with a forward slash in between. Use this concatenated string as the parameter to `ServletContext.getRealPath()`, which will return the true path on the file system to the properties file.
9. Still in the `init()` method, create a FileInputStream from the true path you calculated in step 8. Create a new Properties object, and use the `Properties.load()` method to load information from the FileInput Stream into your Properties object.
10. Use the `ServletContext.setAttribute()` method to set up the Properties object as an attribute of the ServletContext. You don't meet servlet context attributes until the next section of this chapter, but don't let that worry you—they're simple. They allow you to associate a String name with any object. Make sure the name of the attribute is *properties*. The code you use to do it will probably look very like this:

```
context.setAttribute("properties", properties);
```

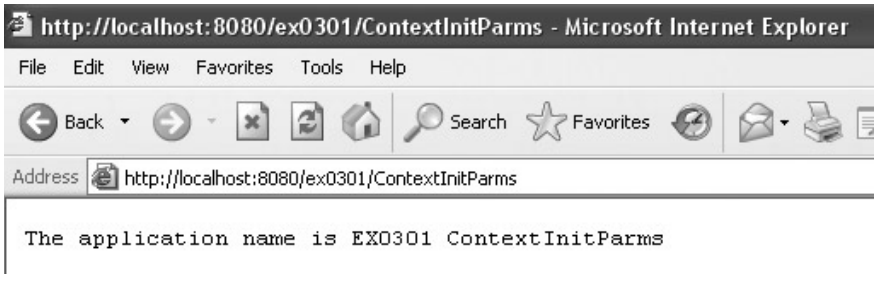
11. Now override the `doGet()` method in ContextInitParms.
12. Retrieve the context attribute you set up in the `init()` method. You'll use the `ServletContext.getAttribute()` method with a parameter of *properties*. Since this returns a plain Object, ensure that you cast the result to a Properties type.
13. Obtain the "application_name" property from your properties object with the `getProperty()` method, and display this (it's a String) in the servlet (write it to the response's PrintWriter).

Run the Application

14. Having compiled your code, copy the entire ex0301 directory to your server's web applications directory.
15. Stop and start your server, checking the application log produced. Since you specified that ContextInitParms should load on start up, the `init()` method should run—make sure that there are no problems.
16. If step 15 was successful, point your browser (using the appropriate servlet mapping that you set up in step 2) to the ContextInitParms servlet with a URL such as this one:

    ```
    http://localhost:8080/ex0301/ContextInitParms
    ```

17. Check that the application name "EX0301 ContextInitParms" appears on the resulting web page. Here's how the solution page looks:



CERTIFICATION OBJECTIVES

Attributes, Scope, and Multithreading (Exam Objectives 3.2 and 4.1)

For the fundamental servlet attribute scopes (request, session, and context): write servlet code to add, retrieve, and remove attributes; given a usage scenario, identify the proper scope for an attribute; and identify multithreading issues associated with each scope.

Write servlet code to store objects into a session object and retrieve objects from a session object.

In this section, we're going to spend some time on attributes. These have a little in common with parameters, but we will explore the difference. Parameters allow information to flow into a web application. We started (in Chapter 1) with request parameters—attaching identifiable data to an HTTP request. In Chapter 2 we met initialization parameters for servlets, and at the beginning of this chapter, we met initialization parameters for the servlet context. Both get information from the web deployment descriptor to web application code. Attributes are more of a means of handling information *within* the web application. They are more of a two-way street than parameters—because you can update attributes in your code, not just read them.

We're also going to consider scope in this section. Web application context (as represented by the ServletContext object) is one of three "scopes" you can use. The other two are session scope (represented by HttpSession objects) and request scope (represented by objects implementing the ServletRequest or HttpServletRequest interfaces). You can attach your attributes to any of these three scopes, and you need to be very familiar with their characteristics for the exam. (Before the end of the book you'll encounter a fourth—page scope—but that needs to wait until we discuss JavaServer Pages more fully in Chapter 6.)

Once you know about scopes, you'll be ready to understand about multithreading issues: questions about the thread safety of different types of attribute—those attaching to request, session, and context. When should you use synchronized blocks for attribute access? Is it ever appropriate to use servlet instance variables instead of attributes? We'll cover these questions and more before the end of the chapter.

Attributes

Attributes are a dumping ground for information. The web container uses attributes (alongside APIs) as a place to

- *Provide information to interested code*. There are a number of standard attributes that a web container should provide as part of the servlet specification, and chances are it will provide a few optional extras of its own. You can regard attributes used this way as a means of supplementing the standard APIs that yield information about the web container.
- *Hang on to information that your application, session, or even request requires later*. These "user-defined" attributes are likely to account for the bulk of attribute usage in your web application.

Attributes are easy to learn for this reason: Whatever scope you are dealing with (request, session, or context), the retrieval and update mechanisms are pretty much identical. The essentials are that you can pick the appropriate object for the scope, then use the appropriate get/set/removeAttribute method.

The thing that takes a little more learning is the idea of "scope." The idea of scope will already be familiar from general Java coding—for example, the idea that a local variable goes out of scope when a method comes to an end. Java coding scope defines how long a variable is available. Web application scope is the same idea on a grander scale—an attribute's scope can span an entire request, web application, session, and even—occasionally—multiple JVMs.

exam Watch

Make sure you don't mix up your attributes and parameters! Parameters—whatever scope you find them in—are read-only. They travel one way: perhaps from an HTML form through an HTTP request, or from initialization parameter settings in the deployment descriptor file to the servlet context or individual servlets. You'll find only "getters" for them. Attributes are read/write—you can create, update, and delete them from all the scopes—request, session, or context.

Mechanisms for Manipulating Attributes

For all three scopes—request, session, and context—there are four methods for the manipulation of attributes. If you take any one of those four methods—say, `getAttributeNames()`—you'll find that it has an identical signature (well, nearly identical) across request, session, and context. Input parameters and return types don't vary, nor do the rules governing what type or constant is returned under what circumstances. The only thing that spoils the perfect symmetry is session, whose methods can throw IllegalStateExceptions. No need to worry about why those might

occur (it's a topic we cover in Chapter 4)—it's just worth noting that the session methods are the "exception" that proves the rule when it comes to parameter and attribute methods (which in general don't throw exceptions on their signature).

The four methods and the three scopes are shown in Table 3-1, for ease of comparison.

TABLE 3-1 Comparison of Attribute Methods for Different Scopes

Scope	Request Scope	Session Scope	Context (Application) Scope
on Interface	javax.servlet .ServletRequest	javax.servlet.http .HttpSession	javax.servlet .ServletContext
`public void setAttribute (String name,Object value)`	Binds an object to the request, keyed by the String name. If the object passed as a value is **null**, has the same effect as `removeAttribute()`.	Binds an object to the session, keyed by the String name. If the object passed as a value is **null**, has the same effect as `removeAttribute()`. Throws IllegalStateException if invoked when the session is invalid.	Binds an object to the context, keyed by the String name. If the object passed as a value is **null**, has the same effect as `removeAttribute()`.
`public Object getAttribute (String name)`	Returns the value of the named attribute as an Object, or **null** if no attribute of the given name exists.	Returns the value of the named attribute as an Object, or **null** if no attribute of the given name exists. Throws IllegalState Exception if invoked when the session is invalid.	Returns the value of the named attribute as an Object, or **null** if no attribute of the given name exists.
`public Enumeration getAttribute Names()`	Returns an Enumeration containing the names of available attributes. Returns an empty Enumeration if no attributes exist.	Returns an Enumeration containing the names of available attributes. Returns an empty Enumeration if no attributes exist. Throws IllegalStateException if invoked when the session is invalid.	Returns an Enumeration containing the names of available attributes. Because there are some attributes that the web container must supply to the context, this Enumeration should never be empty.
`public void remove Attribute (String name)`	Removes the named attribute.	Removes the named attribute. Throws IllegalState Exception if invoked when the session is invalid.	Removes the named attribute.

exam Watch

You have to memorize the material in Table 3-1 for the exam—no way around it! Focus on the similarities and (few) differences between the method calls. There are numerous tricky questions on this content. Be aware that when JavaServer Pages are introduced in Chapter 6, you'll be adding a fourth scope into the mix—page scope—and that exam questions mostly ask you to differentiate among all four scopes, not just the three you have learned so far.

So as you can see, the mechanisms are pretty much identical. There are a few points worth making about the use of these methods:

- *You can choose any String you want for your attribute name.* That said, the servlet specification suggests you follow the "reverse domain name" standard, using names such as "com.mycompany.myattributename."
- *It's impossible to have two or more attributes with the same name.* If you make a call to `setAttribute()` using a name of an attribute that already exists, the existing attribute is replaced with the new value.
- *Session methods are unique among the three scopes in being able to throw exceptions.* This has to do with attempting to use the methods on an "invalid session" and is discussed in detail in Chapter 4.
- *But despite this caveat about session method exceptions, note that* none *of the methods (whatever the scope) throw exceptions just because attributes don't exist.* In this, they are just like parameter methods.
- *Although there is a* `removeAttribute(String name)` *method, you don't necessarily need it.* A call such as this to `setAttribute()` has the same effect:

```
scopeinstance.setAttribute("com.myco.attrname", null);
```

There are some families of attribute names that are reserved for use by Sun Microsystems and the servlet specifiers. Proscribed names begin with

- java.
- javax.
- sun.
- com.sun

This is so that web containers can provide some standard attributes—just like standard properties inside a JVM, retrieved with the `System.getProperty(String propertyName)` static method. A case in point is the ServletContext attribute "javax..servlet.context.tempdir." This is a mandatory attribute that web containers must provide, and it specifies a temporary storage directory unique to a particular web application. This is why—as noted in Table 3-1—you should never have an empty Enumeration returned from `ServletContext.getAttributeNames()`.

So let's look at some code that exercises all these methods for ServletContext (which serves as a guide for all three scopes). The code listing is a fragment from a longer `doGet()` method in a servlet:

```
10 // "out" is the response's PrintWriter
11 out.write("<H2>Context (Application) Scope</H2>");
12 ServletContext context = getServletContext();
13 String myAttributeName = "com.osborne.conductor";
14 context.setAttribute(myAttributeName, "Andre Previn");
15
16 enum = context.getAttributeNames();
17 while (enum.hasMoreElements()) {
18   attrName = (String) enum.nextElement();
19   attrValue = context.getAttribute(attrName);
20   out.write("<BR />Attribute name: <B>" + attrName + "</B>, value: <B>"
21             + attrValue + "</B>");
22 }
23 String conductor = (String) context.getAttribute(myAttributeName);
24 out.write("<BR /> Just used getAttribute() to obtain "
25             + myAttributeName + " whose value is " + conductor);
26
27 context.removeAttribute(myAttributeName);
28 context.setAttribute(myAttributeName, null);
29
30 out.write("<BR />Value of attribute " + myAttributeName + " is now "
31             + context.getAttribute(myAttributeName));
```

The code can be explained as follows:

- Line 12: get hold of a handle to the ServletContext.
- Lines 13–14: add our own attribute (name—"com.osborne.conductor," value—a String object holding the conductor "Andre Previn").
- Lines 16–22: obtain the Enumeration of all context attribute names. Loop around this, using the `getAttribute()` method, to obtain the context value for each name in turn, and output the name and value to the web page.

- Lines 23–24: use `getAttribute()` to get hold of the attribute we added above at lines 13–14, and output the name and value to the web page.
- Line 27 removes this attribute using `removeAttribute()`.
- Line 28 removes this attribute again, using a different technique with `setAttribute()`. This action is completely redundant, for the attribute has already been removed in the previous line, but makes the point that this doesn't matter (no exceptions thrown).
- Line 30: use `getAttribute()` again to prove that the attribute has really gone (**null** is output to the web page, as you can see in the output listing below).

The output from the code (when I run it under Tomcat) is this:

```
Context (Application) Scope
Attribute name: com.osborne.conductor, value: Andre Previn
Attribute name: org.apache.catalina.jsp_classpath, value: /C:/Java/jakarta-
tomcat-5.0.27/webapps/scratchpad/WEB-INF/classes/;C:/Java/jakarta-tomcat-
5.0.27/shared/classes/;<...>
Attribute name: javax.servlet.context.tempdir, value: C:\Java\jakarta-tomcat-
5.0.27\work\Catalina\localhost\scratchpad
Attribute name: org.apache.catalina.resources, value: org.apache.naming
.resources.ProxyDirContext@aa0877
Attribute name: org.apache.catalina.WELCOME_FILES, value: [Ljava.lang
.String;@111ded2
Just used getAttribute() to obtain com.osborne.conductor whose value is Andre
Previn
Value of attribute com.osborne.conductor is now null
```

You can see the context attribute added by our code at the top of the web page—com.osborne.conductor (but do note that there's no guarantee of the order of attributes within the Enumeration returned by the `getAttributeNames()` method). Tucked in the middle of the listing is the compulsory context attribute, javax.servlet.context.tempdir. Sprinkled in between are a few "org.apache.<whatever>" context attributes—these are particular to the Tomcat web container. Some of the outlandish values (e.g., org.apache.naming resources.ProxyDirContext@aa0877) underline the fact that any object of any type can be held as an attribute value—not merely Strings.

Scope

So far in this chapter, I have happily bandied around the term "scope" without attempting to pin down exactly what it means for request, session, and context. I've given you the eagle-level definition: Scope describes the lifetime and availability of

web application artifacts (by which I mean attributes, mostly). What we'll do in this section is to go through each of the three scopes in turn, mapping out their extent.

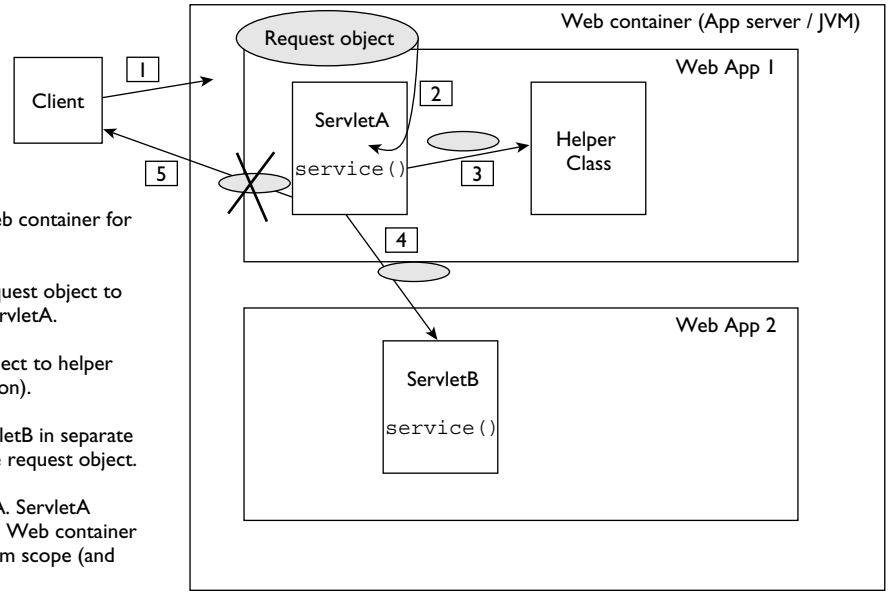
Request Scope

Request scope lasts from the moment an HTTP request hits a servlet in your web container to the moment the servlet is done with delivering the HTTP response. More accurately, the request scope is available from entry into a servlet's `service()` method up to the point of exit from that method. Of course, the default behavior of a servlet's `service()` method—unless you've overridden it to do something unique and strange—is to call a method you're more likely to override—such as `doGet()` or `doPost()`. And here the request is still very much alive and well, and available as a parameter. Now, your `doGet()` or `doPost()` method may in turn make use of other classes, and pass on their HttpServletRequest parameter to those classes' methods. In other words, the whole tree of method calls initiated by the `service()` method—however deep—counts as a single request scope.

The scope is represented by an instance of a request object: most often, one of type HttpServletRequest (it could be of type ServletRequest if we're considering a non-HTTP servlet container—most developers won't meet such a thing, ever). Figure 3-1 shows the progress of a request—possibly not a typical one, for its lifetime is arguably longer than most. You see at (1) in Figure 3-1 a client making a request for a servlet called ServletA. This triggers the web container (at (2) in Figure 3-1) to provide a request object to the servlet's service method. This will—in all likelihood—be passed on as a parameter to the `doGet()` or `doPost()` method (assuming that ServletA is an HttpServlet). ServletA makes use of a helper class (at (3)): We're imagining that the helper class contains a method receiving a Servlet Request as a parameter, so the request object is passed into the helper class. Something more extravagant happens at (4) in Figure 3-1: ServletA decides it needs to make use of ServletB in a separate web application. It can do this by using the RequestDispatcher mechanism—something we learn all about later in this chapter. The reason for its premature inclusion here is to show that a request object's scope can transcend web application boundaries. When you use a RequestDispatcher, you have to pass on the request (and response) objects to the resource to which you are dispatching.

At the end of the request ((5) in Figure 3-1), the request object is placed out of scope. This doesn't necessarily mean that the web container sets its value to **null** and that the JVM garbage collects the object. It's more efficient for a web container to refrain from destroying and recreating request objects—better to throw them back

FIGURE 3-1 Request Scope



1. Client makes request to web container for ServletA.
2. Web container supplies request object to `service()` method of ServletA.
3. ServletA passes request object to helper class (in same web application).
4. ServletA dispatches to ServletB in separate web application, passing the request object.
5. Control returns to ServletA. ServletA `service()` method ends. Web container removes request object from scope (and returns response to client).

into a pool where they can be recycled. You don't have to concern yourself (either for the exam or—usually—in real life) with the mechanism for this: It's up to the web container implementers. What you need to take away is that a request object is no longer any use beyond the scope of a single request. You may wonder how you could ever think otherwise. Well, a misguided move would be to save a reference to the request object—say—as an attribute of your session, then try to recover the request object on the next request to that session. As the servlet specification says, the results could be—well—"unpredictable." As far as the container is concerned, once a request object has passed beyond its rightful scope, the object can be recycled for another request.

Of course, we've accessed the request object for many different reasons already—such as for obtaining header information and retrieving parameters set up on an HTML form. This chapter asks you to consider a new purpose for the request object: as a repository for objects, held as attributes. You might think it hardly worth bothering going to the trouble of attaching and retrieving request attributes. If your request scope comprises a short `doGet()` method in a single servlet, you would be quite right. Why would you set an attribute only to retrieve it a few lines of code later? You might just as well use a local variable in your `doGet()` method instead. How-

ever, the moment you involve a couple of resources—a servlet forwarding to a JSP or, indeed, the fuller life cycle shown in Figure 3-1—request attributes can prove a useful way of passing on information.

Note in passing (though it's very unlikely to come up in the exam) that the web container might attach some attributes of its own to the request. Sometimes it's obliged to when the request is transmitted through a secure protocol such as HTTPs. For example, there is an attribute named javax.servlet.request.key_size, which holds an integer object indicating the bit size of the algorithm used to encrypt the transmission (and you might legitimately wonder why this information isn't enshrined in a full-blown HttpServletRequest API—a method such as `getKeySize()`, perhaps).

Attributes aren't the only things of interest in connection with request objects. Request parameters—which we looked at in Chapter 1—are bound by the same scope. Just as for attributes, if you don't strip out parameter values while the request is in scope—and put them somewhere a bit more permanent—they'll be lost. The "more permanent" aspect is something we now go on to consider as we examine session and context scopes.

Session Scope

Session scope is something we are going to skimp on in this chapter. Don't feel cheated, though—Chapter 4 thoroughly revisits sessions. However, it's worth saying a little about session scope here so you can see it alongside the other two scopes.

You are likely to use session scope a great deal in your applications. Loosely speaking, session scope comes into play from the point where a browser window establishes contact with your web application up to the point where that browser window is closed.

On the application server side, the session is represented by an object implementing the HttpSession interface and can be retrieved from the request object using code such as the following:

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

The idea is that successive requests from the same browser window will each obtain the same session object each time. And that's the whole idea—to provide a scope that allows you as a developer to give a user the feeling of continuity throughout a series of interactions with your web application. So any parameters from web page forms that need to survive beyond single requests—items you add to your shopping cart (the classic, clichéd, and clinching example)—are best served by converting their information to session attributes.

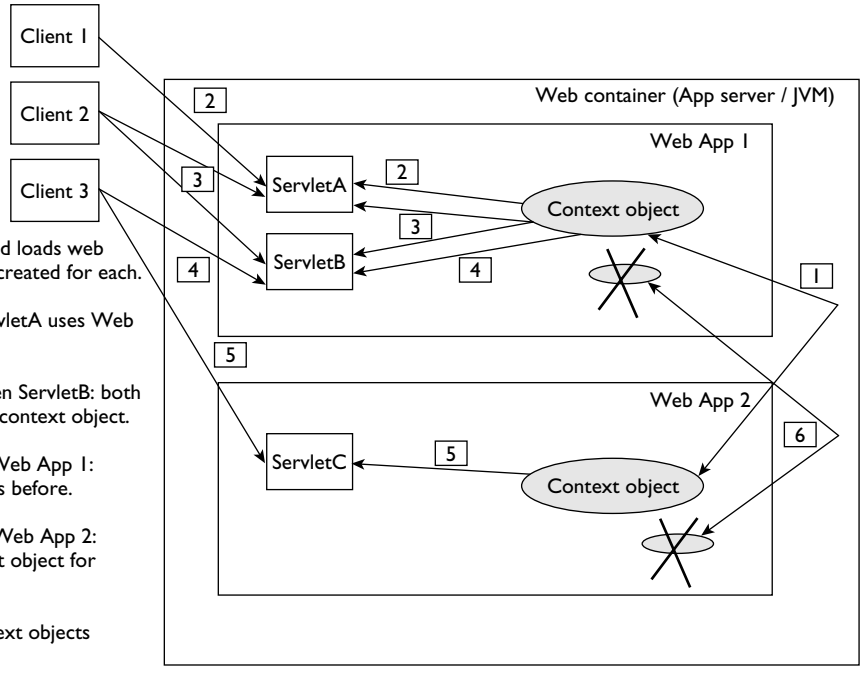
Context (Application) Scope

Let's now deal with context scope, which is the longest-lived of the three scopes available to you. Figure 3-2 shows a web container with two different web applications. On startup of the container, a servlet context is created for each web application (at (1) in Figure 3-2). A client web browser — Client 1 — then requests the services of ServletA, which in turn uses the context object — at (2) in Figure 3-2. Client 2 uses both ServletA and ServletB: Both servlets use the same context object ((3) in Figure 3-2).

So far, all the action has taken place inside one web application — Web App 1. Now Client 3 requests the services of ServletB and ServletC. ServletB uses the context object we've already met in Web App 1. However, ServletC — located in Web App 2 — uses the separate context object that belongs to Web App 2. The point here is that context objects do not straddle different web applications: There is strictly one per web application.

The loss of context scope is shown at (6) in Figure 3-2. If the web container is stopped, then the web applications die — and this includes the context objects for each web application. If a web application is reloaded, then the context object is

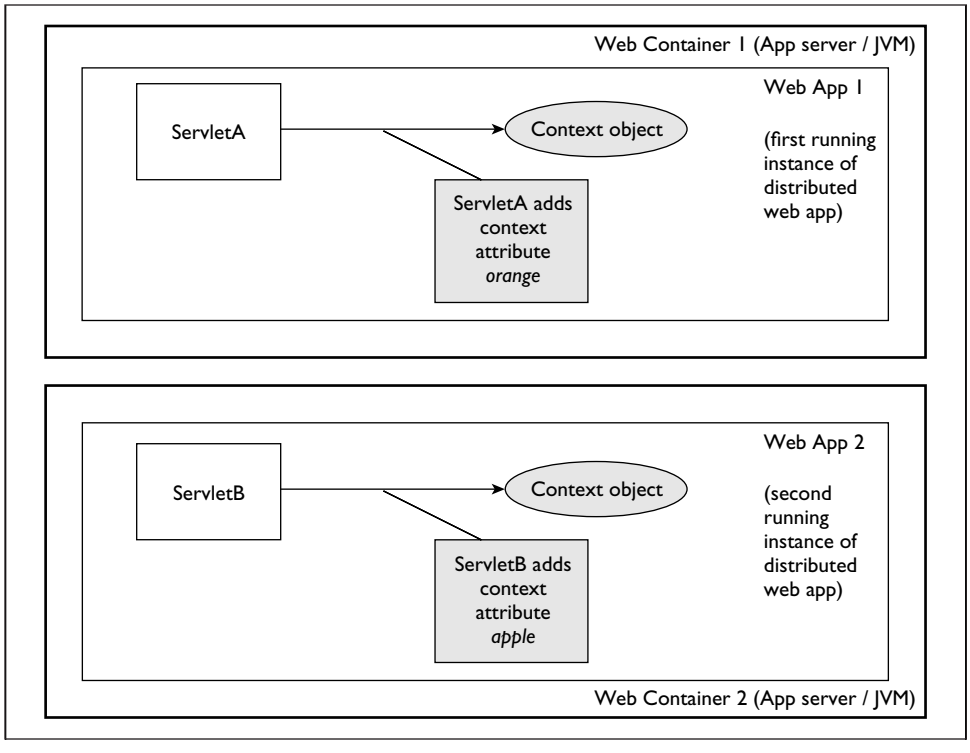
FIGURE 3-2 Context Scope



1. Application server starts up and loads web apps: separate context object created for each.
2. Client1 requests ServletA; ServletA uses Web App 1 context object.
3. Client 2 requests ServletA, then ServletB: both servlets use same Web App 1 context object.
4. Client 3 requests ServletB in Web App 1: ServletB uses context object as before.
5. Client 3 requests ServletC in Web App 2: ServletC uses separate context object for this web app.
6. Application server stops; context objects destroyed.

destroyed and recreated. And if an individual web application is taken out of service, the context object necessarily dies.

When a web application is distributed across different machines—in other words, different JVMs—you might consider the web application to be logically the same wherever it occurs. Don't fall into the trap of thinking that the context object is therefore the same instance whichever clone of the web application you use. Of course, it can't be literally the same instance because we're talking about different JVMs. But neither is the context object logically the same across the identikit copies of the web application: There is one context object per web application per JVM. If you add an attribute to the context in one web application's JVM, it won't be present in the cloned web application's JVM unless you explicitly add it there as well. This is shown in the following illlustration:



You can see the cloned web application Web App 1 appearing twice, each in a separate JVM. Some overall application server architecture (represented by the outer box) will manage issues such as which requests get routed to which clone of the JVM. You see the different servlets in the separate clones adding different named

context attributes to the context object—"apple" in one, "orange" in the other—there is nothing to stop the instances drifting apart and holding different information. If you do need attributes with the same values across JVMs, you need to use session attributes in a distributed web application (see Chapter 4). For distributed sessions, the overall application server architecture is supposed to replicate session information across cloned web applications in different JVMs.

Choosing Scopes

Mostly, you care about scopes for attaching attributes. The question is this: Which is the best scope to use for a given scenario? Should you ignore request attributes because the information placed there evaporates so soon? Should everything go in the context object, because then anything in the web container can access the information? Of course, it depends on circumstances.

SCENARIO & SOLUTION	
You have information retrieved from a database, which is required for a one-off web page and isn't part of a longer transaction.	Store the information in a request attribute. If the information truly isn't needed beyond one production of one HTTP response, then don't keep it around cluttering up the web application JVM memory.
You have some XML files that store essential information about the way your web application works: perhaps some information about table headings and columns. This information changes only when the web application undergoes development changes and is redelivered.	Store the information derived from the XML files in one or more context attributes. The information is then in JVM memory for any web resource to access—no need to keep trawling through the XML file every time a request needs information.
You have information retrieved from a database to a web page in one request. The user makes updates on the web page to the information, and submits the changes, which are subject to some complex, server-side validation. The validation fails, so the user is presented with the changes made and a list of problems that need resolving.	This scenario is a transaction spanning several requests. However, it is information unique to a particular client transaction—not appropriate to attach to the "public" context object. The information that persists across web pages should be held in one or more *session attributes:* more on these in Chapter 4!

Multithreading

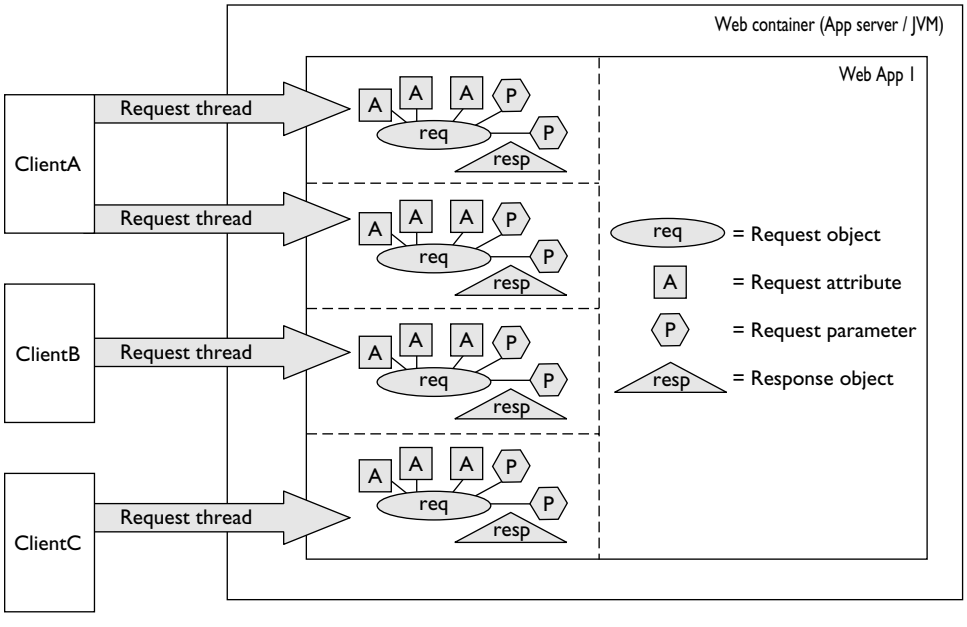
Java is a multithreaded language, but you can pursue a viable Java development career without ever attempting multithreaded programming (apart from what you had to do to pass the SCJD exam, anyway!). However, development for the web container model (not to mention the certification exam) requires some knowledge of the way threads work. Although you are not likely to create threads of your own during servlet and JSP programming (some other areas of J2EE programming ac-

tively ban you from making your own threads), you need to be aware that the web container has the potential to create many threads of its own. And your developed servlets have to live within this multithreaded model.

The issue—which is no different from any other multithreaded environment—is when you have two (or more) concurrently running threads that require access to the same resource. And in the context of threads and web applications, "resource"—more often than not—means an object that is held as the value of some named attribute in some scope or other—request, session, or context. Is one thread changing an attribute value under the feet of another thread reading that value? Worse, are the threads both trying to update an attribute value at the same time? What we consider in this section is the three scopes again: which are thread safe, which are not, and what you need to do about it as a developer.

Multithreading and Request Attributes

We'll first consider requests. The good news here is that request attributes are thread safe. In fact, everything to do with the request and response objects (results of method calls on these objects, request attributes, request parameters, etc.) will only ever be accessed by one thread and one thread alone. The web container provides a new (or as good as new) instance of the request and response objects whenever a new request is received. Here's how it looks:

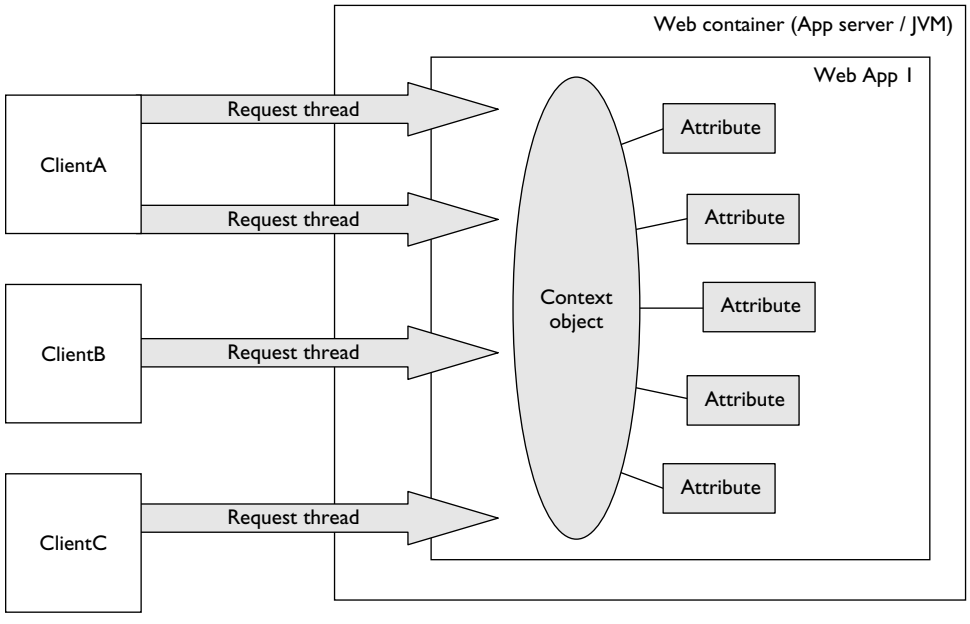


Multithreading and Session Attributes

Again, we're going to defer a full discussion of this until Chapter 4, on sessions. Suffice to say for the moment that session attributes are *officially* not thread safe. For most common web applications, though, you can assume thread safety. But until you've passed your SJWCD exam, assume not (just as you wouldn't cross your hands on the steering wheel until *after* you've passed your driving test).

Multithreading and Context Attributes

The servlet context represents the opposite extreme from the request. The world and his wife have access to the servlet context object. To speak more technically, each and every client requesting thread to a web application can potentially update a servlet context attribute. So you *do* need to worry about thread safety.



You have two approaches to solve the multithreading dilemma:

1. Treat servlet context attributes like servlet context parameters. Set up servlet context attributes in the `init()` method of a servlet that loads on the startup of the server, and at no other time. Thereafter, treat these attributes as "read only"— only ever call `getAttribute()`, never `setAttribute()`, on the context object.

2. If there are context attributes where you have no option but to update them later, surround the updates with synchronization blocks. If it's crucial that no other thread reads the value of these attributes mid-update, you'll need to synchronize the `getAttribute()` calls as well.

Bear in mind that synchronization—especially on a much-accessed context attribute—might create a huge bottleneck in your application. Keep it to a minimum! Also bear in mind that the recommendations above are not enforced by anything in the web container model. It's down to programming standards to enforce these approaches.

exam Watch

As of this version of the exam, you will no longer get questions about the SingleThreadModel interface. This interface is deprecated in the version 2.4 of the servlet specification. The idea was that if a servlet implemented this interface, the web container would guarantee that only one thread at a time would be able to access any given instance of the servlet. Web containers providing this facility paid a high cost in performance terms. And so it was deemed much better to avoid this approach altogether: better to rely on other ways of enforcing thread safety where needed (in general, synchronize what you have to, but keep this to a minimum).

on the Job

The old exam syllabus required awareness of thread safety issues regarding types of variables you might use in servlets: local, instance, and class. Although this isn't part of the exam anymore, it's still good to have some knowledge when out in the field.

Local Variables: ***These are either parameters to servlet methods, or variables declared within methods. Just as elsewhere in Java, local variables are completely thread safe: The JVM guarantees that there can be only one thread ever accessing a local variable. Even if the same instance of a servlet is running simultaneously in multiple threads, the*** `service()` ***and other methods are effectively separate.***

Instance Variables: ***These are considered harmful! They are not thread safe, so add your own synchronization if you have to. You never know when there will be more than one thread accessing a single instance of a servlet.***

But more than that, servlet instance variables are not particularly useful. Except for the very specialized case in which you want to keep track of information about individual servlet instances (perhaps to record a usage count per instance), there isn't much you can usefully define as servlet instance information. You can't guarantee that the same client targeting the same servlet URL more than once will get the same instance, so servlet instance information is no substitute for session attributes.

Class Variables: ***Again, these are not thread safe. However, they are potentially a bit more useful than servlet instance variables because the information is shared across all instances of a particular servlet. So a usage counter on a class variable does tell you how many times the actual servlet class was accessed—which is more likely to be something you want to know (as opposed to separate counters for individual instances of the servlet).***

EXERCISE 3-2

Displaying All Attributes in All Scopes

In this exercise, you'll display the attributes available to a web page in all scopes. The context directory for this exercise is ex0302, so set up your web application structure under a directory of this name.

For this exercise, there's a solution in the CD in the file sourcecode/ch03/ex0302.war—check there if you get stuck.

Set Up the Deployment Descriptor

1. Declare a servlet named AttributesAllScopes, with a suitable servlet mapping. If needed, refer to Chapter 2 to refresh yourself on `<servlet>` element setup.

Write the AttributesAllScopes Servlet

2. Create a Java source file AttributesAllScopes.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
3. Override the `doGet()` method in AttributesAllScopes.

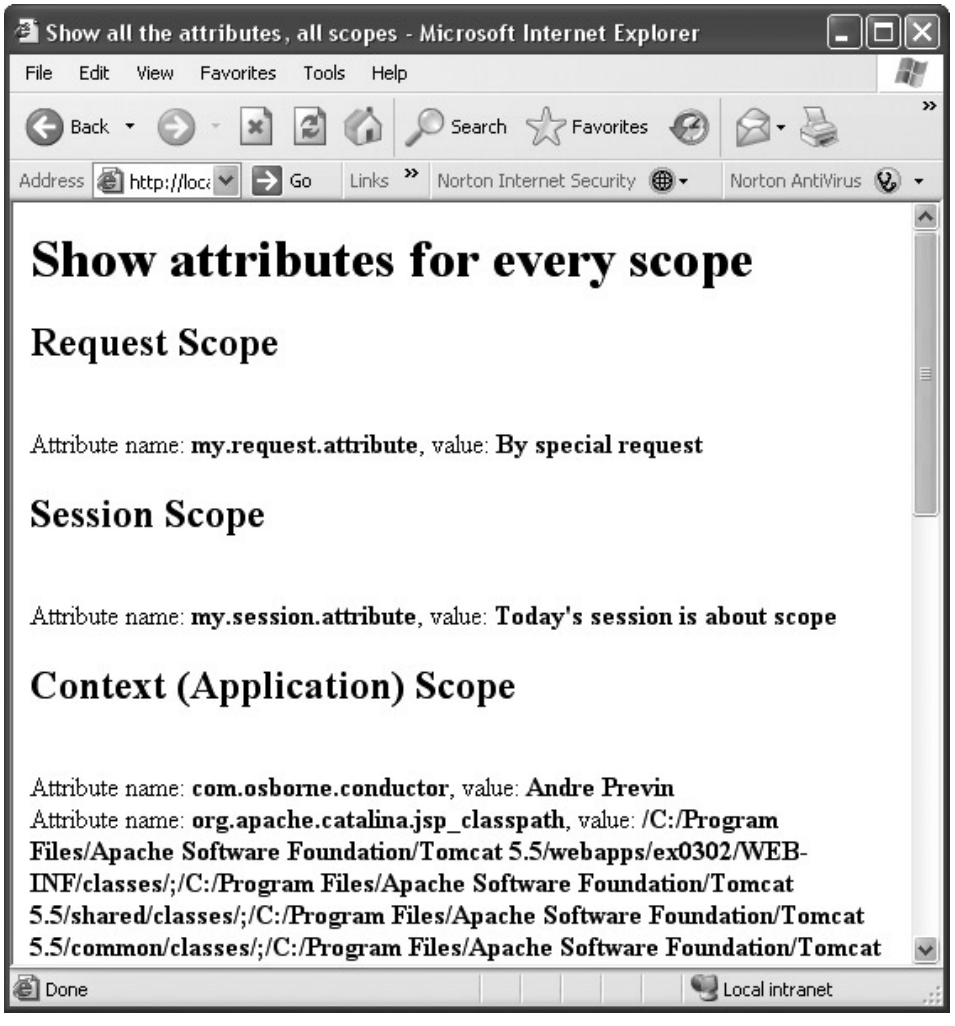
4. Do the necessary preliminaries to obtain the response's PrintWriter, and set the content type to "text/html."
5. Using the request object (of type HttpServletRequest) that's passed as a parameter into the `doGet()` method, retrieve an Enumeration using the `getAttributeNames()` method.
6. Write code to go through all the elements in the Enumeration. For each parameter name retrieved, display the attribute name. Get hold of the corresponding attribute value using the `getAttribute()` method on the request object.
7. Now obtain the session object, using the `getSession()` method on the request object. Repeat steps 5 and 6—getting an Enumeration of parameter names and displaying the attribute names and values—but with the session rather than the request object.
8. Now obtain the context object, using the servlet's `getServletContext()` method. Again, repeat steps 5 and 6 for context attributes.

Run the Application

9. Having compiled your code, copy the entire ex0302 directory to your server's web applications directory. Stop and start your server (if necessary to deploy the application).
10. Point your browser (using the appropriate servlet mapping that you set up in step 1) to the AttributesAllScopes servlet, and check that a web page appears displaying at least some attributes. Here's the URL you are likely to use with the Tomcat server:

```
http://localhost:8080/ex0302/AttributesAllScopes
```

11. It's almost certain you will get some context attributes appearing. However, it's quite likely that request and session scopes will come up blank. Add some request and session attributes to the request and session objects (using the `setAttribute()` method) at the beginning of your `doGet()` method to prove that your retrieval code works. The following illustration shows a screen shot from the solution code.


Show all the attributes, all scopes - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://loc Go Links Norton Internet Security Norton AntiVirus
Show attributes for every scope
Request Scope
Attribute name: my.request.attribute, value: By special request
Session Scope
Attribute name: my.session.attribute, value: Today's session is about scope
Context (Application) Scope
Attribute name: com.osborne.conductor, value: Andre Previn
Attribute name: org.apache.catalina.jsp_classpath, value: /C:/Program Files/Apache Software Foundation/Tomcat 5.5/webapps/ex0302/WEB-INF/classes/;/C:/Program Files/Apache Software Foundation/Tomcat 5.5/shared/classes/;/C:/Program Files/Apache Software Foundation/Tomcat 5.5/common/classes/;/C:/Program Files/Apache Software Foundation/Tomcat
Done
Local intranet

CERTIFICATION OBJECTIVE

Request Dispatching (Exam Objective 3.5)

Describe the RequestDispatcher mechanism; write servlet code to create a request dispatcher; write servlet code to forward or include the target resource; and identify and describe the additional request-scoped attributes provided by the container to the target resource.

So far, we've considered the case of calling one single servlet to accomplish a task: request made, servlet executes, response generated—job done. We have also learned about helper classes that a servlet can use. We can include those in the web application's /WEB-INF/classes directory or in a JAR file in /WEB-INF/lib, as we saw in Chapter 2.

What, though, if the servlet we summon wants to call on the services of another servlet within the web application? Or some other web resource, maybe a JSP? Or even hand off all responsibility and let a different servlet handle the work? Enter the RequestDispatcher: a mechanism for controlling the flow of control within the web resources in your web application.

Obtaining a RequestDispatcher

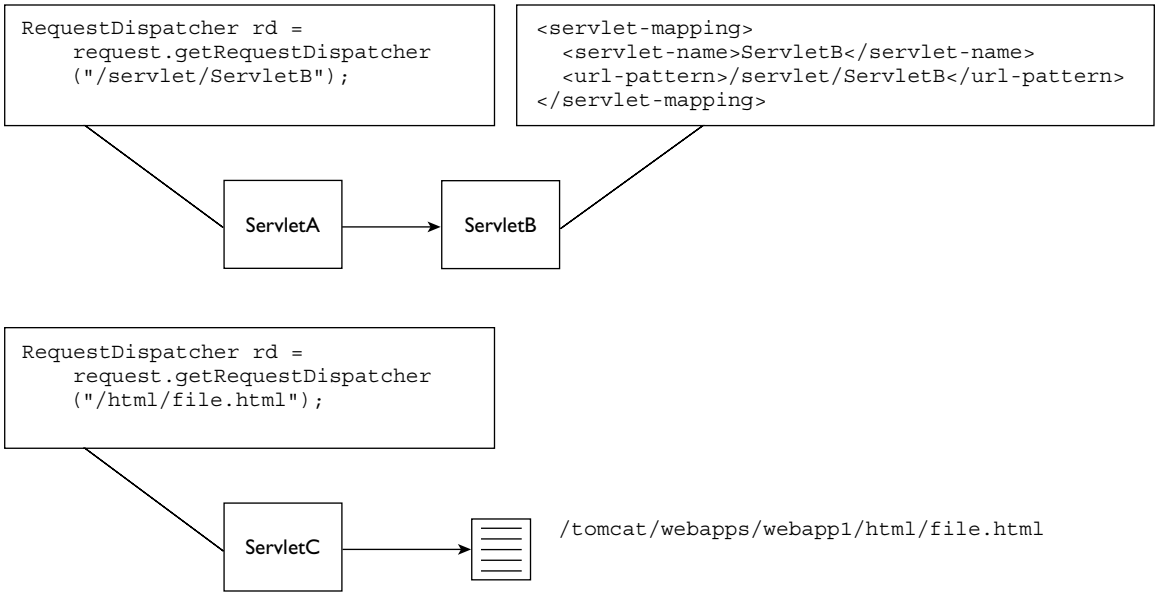
You can obtain a RequestDispatcher either from the request (ServletRequest) or the web application context (ServletContext). ServletRequest has one method for getting hold of a RequestDispatcher, and ServletContext has two—making three possible ways of getting hold of a RequestDispatcher. You can be sure that exam questions will focus on the subtle shades of difference among these three methods!

We'll first of all look at the "how" of getting a RequestDispatcher from the three methods in question.

From ServletRequest

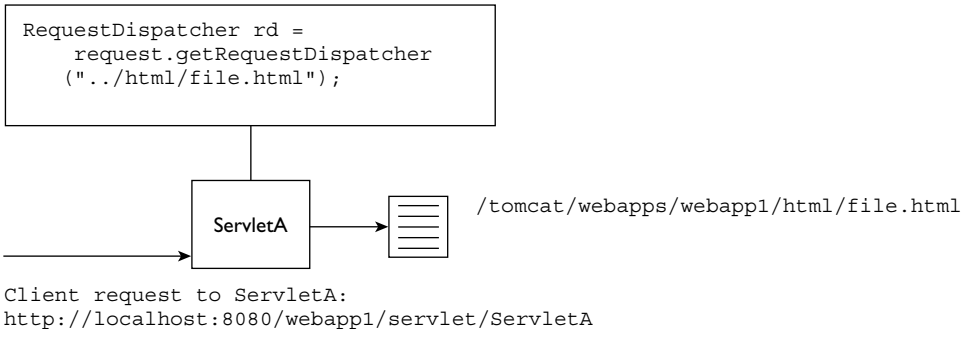
ServletRequest has one method for getting a RequestDispatcher: `getRequestDispatcher(String path)`. Note that the method is part of parent interface ServletRequest. Of course, HttpServletRequest has it too by virtue of inheriting from ServletRequest. But if you're faced with a question about where the method originates, you need to know this information!

The path parameter can be a full path beginning at the context root. This means a path to the resource without naming the context root itself. The following illustration gives some examples.



The forward slash ("/") at the beginning of the path denotes the context root. After this, you can append whatever path leads to a web resource—typically a dynamic one (another servlet or JSP), though it doesn't have to be. You can point your RequestDispatcher to a plain old static HTML page if that's your wish. So when ServletA calls ServletB, it uses a matching `<url-pattern>` for ServletB when creating the RequestDispatcher. When ServletC dispatches to a static HTML file (file.html in the html directory of webapp1), it supplies the full path starting from the context root (e.g., "/html/file.html").

There's also the possibility of specifying a path without an initial forward slash to `ServletRequest.getRequestDispatcher()`.The following illustration shows how this might work.



You see how the client request to ServletA—from the web application root—is "/servlet/servletA." The parameter to the `getRequestDispatcher()` method is relative to something—but what? It's not the context root. This time, it's the directory containing the resource requested (ServletA)—notionally the directory /servlet. Yes, I know this may not be a *real* directory—just a logical fiction made up in a servlet mapping—because the real servlet class probably inhabits some more involved location such as "/WEB-INF/classes/com/osborne/servlets." The point is that relative requests are relative to the URL as given. So now the `getRequestDispatcher()` receives the parameter "../html/file.html." The ".." means go up one directory in the request—in other words, from servlet to the root directory of the web app. The rest of the parameter "/html/file.html" now works as if expressed from the root directory of the web app.

So you see that this form of relative path syntax gives considerable flexibility. That said, I try to keep things simple—and favor straightforward full paths from the context root beginning with a forward slash. You can see how the use of ".." to go up to the parent directory is quite vulnerable to later restructuring of resources in the web application (for example, through servlet mapping changes).

From ServletContext

ServletContext has two methods of getting hold of a RequestDispatcher. These are `getRequestDispatcher(String path)` and `getNamedDispatcher(String name)`.

`getRequestDispatcher(String path)` This method works in exactly the same way as the same named method on ServletRequest. There is a restriction: Only full paths are allowed (i.e., paths beginning with a forward slash "/," which denotes the context root). Paths without the initial forward slash will not work. You'll get a runtime exception—IllegalArgumentException—and text along the lines of "Path myPath does not begin with a "/" character."

`getNamedDispatcher(String name)` This method does bring something new to the party. Instead of specifying a path, you supply a name for the resource you want to execute. The name must match one of the `<servlet-name>` values you have set up in the deployment descriptor, so it can refer to a named servlet or a JSP.

You may have idly wondered whether there was any point in setting up a `<servlet>` entry in the deployment descriptor without a corresponding `<servlet-mapping>`. The `getNamedDispatcher()` method is the point—it gives a means of executing a servlet (or JSP) that doesn't have any other means of access. This is potentially very useful. There may be some services within your application that are

exam Watch

What happens if an incorrect path is fed to a RequestDispatcher method? Answer: the method returns* null*. So that's what your code should test for (you don't attempt to catch any kind of exception).

available only in particular circumstances, or only internally to your application. You may not want these to be sitting on any kind of public path that can be typed into the address line, and a `<servlet-mapping>` gives just that sort of public access. If you do exploit this technique, make sure to switch off any server loopholes—such as the ability to execute servlets if you happen to know their name. Most servers have such capabilities as a convenience for developers, but they have no place in a production environment.

exam Watch

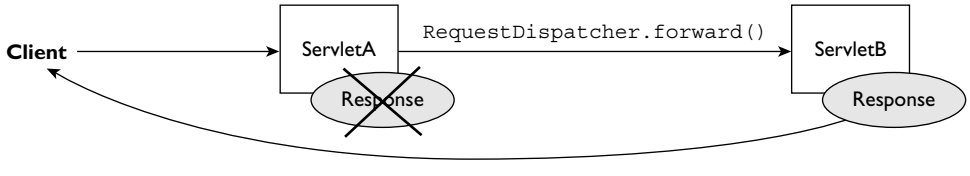
There is no way of escaping a web application context with any of the methods that return a RequestDispatcher. However, there's nothing stopping you from using ServletContext's* `getContext()` *method to get yourself another web application context, then obtaining a RequestDispatcher on that other context.

Using a RequestDispatcher

Having obtained a RequestDispatcher, your servlet can do one of two things with it. Either it forwards to another web resource (washing its hands of the responsibility of returning a response) or includes another web resource within its own output. The RequestDispatcher interface has only two methods—`forward()` and `include()`—so no surprises there. We'll now look at these methods in some detail.

Forwarding

We'll first consider the case of forwarding. The following illustration gives a graphic account of what happens when a servlet forwards to another servlet.



You see that the first servlet is effectively forgotten. Although it can have code that writes output to the response, the contents of the response buffer are lost at the point of forwarding to the second servlet. For this reason, if the first servlet is past the point of no return and has committed any of its response to the client, then a forward call is illegal—and will, indeed, result in an IllegalStateException at runtime.

The forward method accepts two parameters—a ServletRequest and the Servlet Response. All you have scope to do is to pass on the request and response received into the forwarding servlet's `service()` method (which you more likely get hold of in the `doGet()` or `doPost()` servlet method). You mustn't manufacture your own servlet requests and responses and plug these in instead. But then, why would you?

So, let's consider a servlet class that exists solely to forward somewhere else (import statements omitted to save space):

```
public class FlexRequestDispatcher extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    String fwdPath = request.getParameter("fwd");
    System.out.println("The dispatch path is: " + fwdPath);
    RequestDispatcher rd = request.getRequestDispatcher(fwdPath);
    // The following two lines are a waste of effort: the
    // response output will be binned...
    PrintWriter out = response.getPrintWriter();
    out.write("This text will be lost");
    // ...in favor of the response from the resource you are
    // forwarding to...
    rd.forward(request, response);
  }
}
```

The code expects a parameter named *fwd* which contains a String with the path to forward to. So you might call it with a request such as the following:

```
http://localhost:8080/mywebapp/FlexRequestDispatcher?fwd=/AnotherServlet
```

The *fwd* parameter is then translated from the query string, so the *fwdPath* string local variable would have a value of "/AnotherServlet." This is passed into the ServletRequest's `getRequestDispatcher` method. Because the path begins with

a forward slash ("/"), it will be interpreted relative to the context root—equivalent to the address

```
http://localhost:8080/mywebapp/AnotherServlet
```

Assuming that "/AnotherServlet" is a valid resource in the web application (presumably a valid servlet mapping), then the RequestDispatcher instance *rd* will have a value. All that remains is to execute `rd.forward()`, supplying the request and response as passed into the `doGet()` method. A well-behaved servlet might first test whether *rd* is **null** before attempting to execute the `forward()` method, to protect against a NullPointerException.

Note the two lines that obtain the PrintWriter from the response and write to it. These are effectively a waste of effort. As soon as the FlexRequestDispatcher servlet forwards to the requested resource, the response of FlexRequestDispatcher will effectively be nullified—only the forwarded-to resource's output will be visible in the resulting response.

Special Attributes for Forwarding

Let's now suppose you've arrived in the "forwarded-to" servlet AnotherServlet, which contains the following code:

```
public class AnotherServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    String servletPath = request.getServletPath();
    System.out.println("The servlet path is: " + servletPath);
  }
}
```

Which servlet path is printed to the server console—that of the *forwarding* servlet FlexRequestDispatcher, or the *forwarded-to* servlet AnotherServlet? The answer is the *forwarded-to* servlet AnotherServlet. So the output in the server console might look like this:

```
The servlet path is: /AnotherServlet
```

This is because a *forwarded-to* servlet has complete control over the request—it's as if the *forwarding* servlet had never been called.

What, though, if you want to get at the original servlet path for the request while within AnotherServlet's code? The web container provides for this. Five special attributes are set up that reflect "original" values about the request path, instead of the request path, which has been modified to fit the forwarded to servlet. To get one of these values, simply use the `request.getAttribute()` method. The following table shows all five attributes, together with a description of what they represent and the request method for which the attributes provide a necessary substitute. Assume that the full URL to the forwarding servlet is

```
http://localhost:8080/myapp/ForwardingServlet/pathinfo?fruit=orange
```

Attribute Name	Description	"Equivalent" Method on ServletRequest
javax.servlet.forward.request_uri	The URI of the original request to the forwarding servlet (e.g., /myapp/ ForwardingServlet /pathinfo)	`getRequestURI()`
javax.servlet.forward.context_path	The context path for the forwarding servlet (e.g., /myapp)	`getContextPath()`
javax.servlet.forward.servlet_path	The servlet path for the forwarding servlet (e.g., /ForwardingServlet)	`getServletPath()`
java.servlet.forward.path_info	The path information for the forwarding servlet (e.g., /pathinfo)	`getPathInfo()`
java.servlet.forward.query_string	The query string attaching to the original request for the forwarding servlet (e.g., fruit=orange)	`getQueryString()`

The web container is contracted (by the servlet specification) to provide these attributes. Of course, the attributes are not present if the value returned by them would be **null** anyway (e.g. you won't find a java.servlet.forward.query_string attribute when there is no query string on the request URI).

The "equivalent" methods shown in the table are not really equivalent at all. The point of supplying the attributes is that they give alternative information that is

otherwise invisible through the apparently equivalent method. This is summarized in Table 3-2, after we look at the set of special attributes that arise when a Request Dispatcher is used to include a web resource.

on the job

Forwarding is not so very different from request redirection (`Servlet Request.sendRedirect()`). However, forwarding has an advantage—the request information (parameters and attributes) are preserved. Redirection effectively initiates a new request from the client; the original request parameters and attributes are lost (though you can add new parameters—or preserve existing ones—by adding them to the query string in the URL that is the parameter for `sendRedirect()`). So on the face of it, forwarding is always better—information is preserved, and it's more efficient, for there's no return trip to the client. However, beware of any relative URLs in the response from the servlet to which you forward. The requesting browser will still think it's dealing with the original URL (i.e., of the servlet that did the forwarding). You can generally see this in the address line of the browser: If ServletA did the forwarding, and ServletB is forwarded to, you'll still see http://www.myco.com/webapp/ServletA. If your image links for ServletB's output are relative links, they'll be fine—unless the relative path from ServletB to the images is different from that of ServletA.

exam watch

When you forward to another servlet, you might be tempted to think that control never returns to the servlet you are forwarding from. Not so. Consider the following code, where ServletA forwards to ServletB, but there is code following the `forward()` method in ServletA:

```
public class ServletA extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    String fwdPath = "/ServletB";
    RequestDispatcher rd = request.getRequestDispatcher(fwdPath);
    rd.forward(request, response);
    System.out.println("Back in ServletA");
  }
}
```

```
public class ServletB extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    System.out.println("Now in ServletB");
  }
}
```

When you call ServletA, the output to the server console (note: not the response/web pages!) is as follows:

```
Now in ServletB
Back in ServletA
```

What you can't do in ServletA—after the forward call—is anything that might attempt to affect the response. Well, you can do it—and the lines of code will execute harmlessly, having no effect. But code that does things unrelated to the response (such as outputting text to the console, setting attributes, and writing to logs) will execute as normal.

Including

The alternative to the `forward()` method on RequestDispatcher is the `include()` method. Instead of "passing the buck," an including servlet takes the contents of the included web resource and adds this to its own response. Let's adapt an example from before, now using the `include()` method instead of `forward()`:

```
public class FlexRequestDispatcher extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    String incPath = request.getParameter("inc");
    System.out.println("The dispatch path is: " + incPath);
    RequestDispatcher rd = request.getRequestDispatcher(incPath);
    PrintWriter out = response.getPrintWriter();
    out.write("The output will start with this text, ");
    rd.forward(request, response);
    out.write("and finish with this text.");
  }
}
```

Suppose we fed a parameter such as *inc=/IncludedServlet* to FlexRequestDispatcher, and IncludedServlet includes this code:

```
PrintWriter out = response.getPrintWriter();
out.write("continue with this included text, ");
```

The sum of the output should look something like this in the resulting response:

```
The output will start with this text, continue with this
included text, and finish with this text.
```

exam Watch

What if you try to call the `forward()` or `include()` methods when a response has already been committed to the client? The answer is that you get an IllegalStateException for the `forward()` method—as we mentioned before. But you don't get any exception if you call an `include()` after the response is committed. After all, you are not trying to throw away the original response with an `include()`—only augmenting the original response.

Special Attributes and Including

Just as a "forwarded-to" servlet has access to some special attributes, so does an "included" servlet. There are a similarly named set of five attributes, though their significance is almost opposite to the forwarding set. We'll assume a URL this time of

```
http://localhost:8080/myapp/IncludingServlet/pathinfo?fruit=orange
```

and a code snippet from Includ**ing**Servlet that includes Includ**ed**Servlet, as follows:

```
RequestDispatcher rd =
  req.getRequestDispatcher("/IncludedServlet/newPathInfo?fruit=apple");
rd.forward(req, resp);
```

Now the attributes (as you access them in Includ**ed**Servlet) have the following values:

Attribute Name	Description	"Equivalent" Method on ServletRequest
javax.servlet.include.request_uri	The URI of the revised request to the included servlet (e.g., /myapp/IncludedServlet/newPathInfo)	`getRequestURI()`
javax.servlet. include.context_path	The context path for the included servlet (e.g., /myapp)	`getContextPath()`
javax.servlet. include.servlet_path	The servlet path for the included servlet (e.g., /IncludedServlet)	`getServletPath()`
java.servlet. include.path_info	The path information for the including servlet (e.g., /newPathInfo)	`getPathInfo()`
java.servlet. include.query_string	The query string attaching to the *revised* request to the included servlet (e.g., fruit=apple)	`getQueryString()`

It's not an easy picture to grasp! The value derived from the attribute is different from the value returned from the "equivalent" method. When in the *included* servlet, use the method to get information about the *including* servlet, and the attribute to get information about the *included* servlet (i.e., the servlet you are in). Table 3-2 summarizes what servlet you get information about, dependent on (a) what kind of servlet you are in (forwarding, forwarded to, including, or included) and (b) whether you are using request methods, special forward attributes, or special include attributes.

TABLE 3-2 Summary of Information from Request Methods and Special Attributes

In the Code of	Forwarding Servlet	Forwarded to Servlet	Including Servlet	Included Servlet
servlet method call (e.g., `getServletPath()`)	Forwarding	Forwarded to	Including	Including
forward attribute (e.g., javax.servlet.**forward**.servlet_path)	N/A	Forwarding	N/A	N/A
include attribute (e.g., javax.servlet.**include**.servlet_path)	N/A	N/A	N/A	Included

Special Attributes and `getNamedDispatcher()`

One last point about special attributes (both the forward set and the include set, so this is as applicable to javax.servlet.**forward.**context_path as it is to javax.servlet.**include.**query_string): When you use `getNamedDispatcher()` on ServletContext to get hold of a RequestDispatcher (as opposed to `getRequestDispatcher()` on ServletContext or ServletRequest), these special attributes *are not set*. The rationale is that you are not forwarding or including via an external request. Because all these special attributes pertain to features of external requests (mostly URL information), they are not deemed relevant to an internal server call to a named resource.

EXERCISE 3-3

Implementing a RequestDispatcher and Viewing Special Attributes

This exercise will implement two servlets, one that dispatches to the other. By accepting a parameter, we'll make the dispatching servlet behave flexibly so that it may either forward or include the dispatched-to servlet. In either case, we'll display the special attributes that are associated with the forward or include so that these appear on the web page produced by the servlet(s). The context directory for this exercise is ex0303, so set up your web application structure under a directory of this name.

For this exercise, there's a solution in the CD in the file sourcecode/ch03/ex0303.war—check there if you get stuck.

Set Up the Deployment Descriptor

1. Declare a servlet named Dispatcher, with a suitable servlet mapping. If needed, refer to Chapter 2 to refresh yourself on `<servlet>` element setup.
2. Also declare a servlet named Receiver with a servlet mapping that will trap path information—a `<url-pattern>` of (e.g.) /Receiver/*.

Write the Dispatcher Servlet

3. Create a Java source file Dispatcher.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
4. Override the `doGet()` method in Dispatcher.
5. In the `doGet()` code, retrieve a request parameter whose name is "mode," and hold this in a local String variable.

6. Obtain a RequestDispatcher from the request or context object—set the path parameter to "/Receiver/pathInfo?fruit=orange."
7. Test the value of the "mode" parameter obtained in step 5. If the value is "forward," call the forward method on the request dispatcher object obtained in step 6; if it is "include," call include instead.
8. Write something to the response object so that you know from the web page that this is the Dispatcher servlet (not the Receiver).

Write the Receiver Servlet

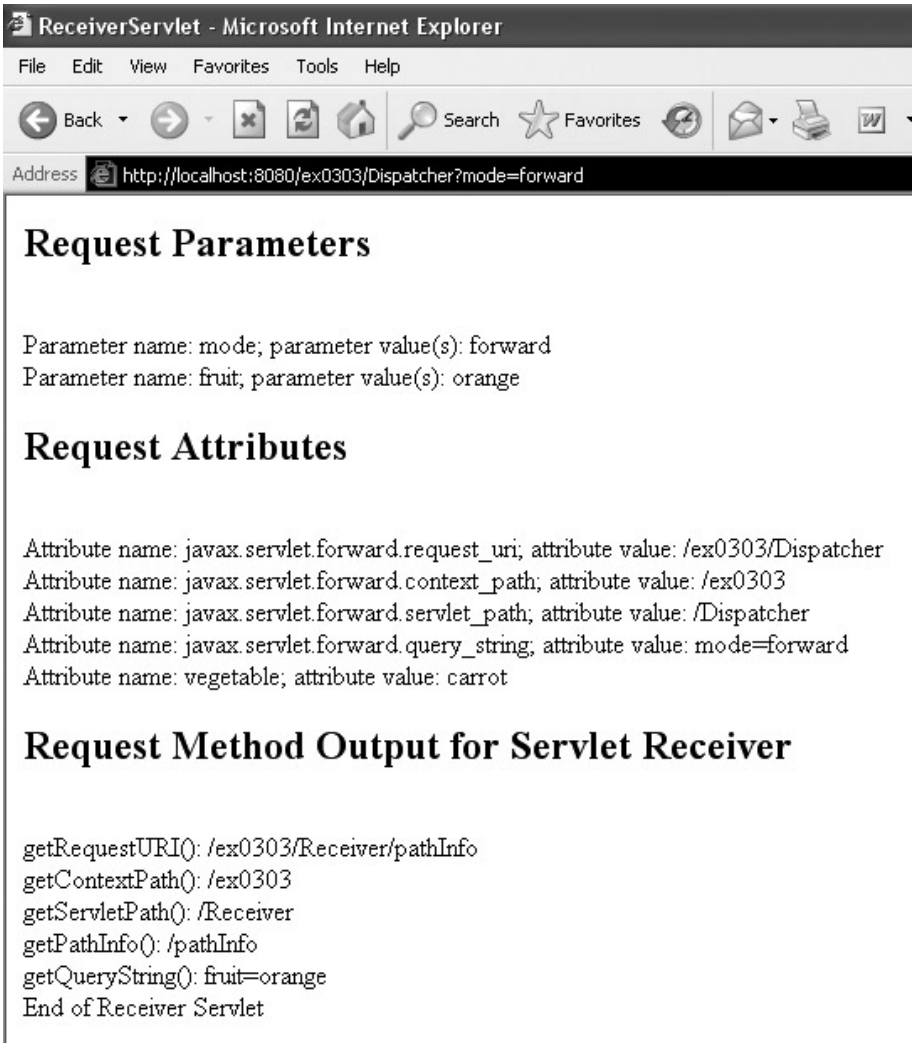
9. Create a Java source file Receiver.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
10. Override the `doGet()` method in Receiver.
11. In the `doGet()` code, obtain the Enumeration of parameter names from the request object, and display to the web page all parameter names and values.
12. Do the same for request attribute names and values. This will display the special attributes supplied by the web container for a forwarded or included servlet.

Run the Application

13. Having compiled your code, copy the entire ex0303 directory to your server's web applications directory. Stop and start your server (if necessary to deploy the application).
14. Point your browser (using the appropriate servlet mapping that you set up in step 1) to the Dispatcher servlet, ensuring that you pass the "mode" parameter: Use a URL such as the following:

```
http://localhost:8080/ex0303/Dispatcher?mode=forward
```

15. Compare the outputs you get from forwarding and including. Note that the text output by the Dispatcher servlet is simply not present when you forward. However, when you include, the text of the Dispatcher and Receiver servlets should appear. The following illustration shows the solution code output when mode=forward.


ReceiverServlet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/ex0303/Dispatcher?mode=forward
Request Parameters
Parameter name: mode; parameter value(s): forward
Parameter name: fruit; parameter value(s): orange
Request Attributes
Attribute name: javax.servlet.forward.request_uri; attribute value: /ex0303/Dispatcher
Attribute name: javax.servlet.forward.context_path; attribute value: /ex0303
Attribute name: javax.servlet.forward.servlet_path; attribute value: /Dispatcher
Attribute name: javax.servlet.forward.query_string; attribute value: mode=forward
Attribute name: vegetable; attribute value: carrot
Request Method Output for Servlet Receiver
getRequestURI(): /ex0303/Receiver/pathInfo
getContextPath(): /ex0303
getServletPath(): /Receiver
getPathInfo(): /pathInfo
getQueryString(): fruit=orange
End of Receiver Servlet

CERTIFICATION OBJECTIVE

Filters and Wrappers (Exam Objective 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 a wrapper.

We're now going to look at the interfaces and classes that make up the filtering mechanism in the web container model. Filters are intriguing beasts. In many respects they are like servlets: They receive requests and responses that they can manipulate, they have access to the servlet context, and (like request dispatchers for the servlet) they have an inclusion mechanism whereby a filter can pass control to another filter or a servlet.

Their main purpose is to intervene before and after a request for a web resource. The web resource itself need not be aware that it has been nested in a filter. An example might help. Suppose you want all the output from your web application to be encrypted. You can write a filter that triggers on any request to your web application (whatever the web resource requested). The filter will trap the response from the web resource, run some kind of encryption algorithm over it, then assume responsibility for returning the response to the requester.

Because you so often use a filter to transform the response—and sometimes the request—you may want specialized request and response objects with their own specialized behavior. There is provision for this in the so-called "wrapper" classes—such as HttpServletResponseWrapper. You can subclass these wrapper classes, then substitute them for the original response (or request) that is passed to the filter. We'll talk about filters and wrappers in some detail.

Filters

The servlet specification gives a useful and fairly comprehensive list of the uses you might find for filters:

- Authentication filters
- Logging and auditing filters
- Image conversion filters

- Data compression filters
- Encryption filters
- Tokenizing filters
- Filters that trigger resource access events
- XSL/T filters that transform XML content
- MIME-type chain filters
- Caching filters

You may gather from this list that filters might be used for pre-processing requests for resources, as would be the case for an authentication filter. If your credentials aren't up to scratch, a filter has the power to deny access to the requested resource. A data compression filter is most likely to kick in on the response, perhaps converting the output to a zipped output stream before allowing the response to return.

Now that we've seen what filters are capable of, let's take a look at what you need to know to write and implement one: which interfaces and classes are involved, and what you need to declare in the deployment descriptor.

Writing the Filter Code

When writing a filter, these are the steps:

1. Write a class that implements the javax.servlet.Filter interface.
2. Implement the three methods of the Filter interface: `init()`, `destroy()`, and `doFilter()`.
3. Optionally, provide a no-argument constructor.

Not much to it, really—but there is a bit of devilry in the detail. Let's consider those three methods further. Together, they constitute the filter life cycle, as shown in Figure 3-3.

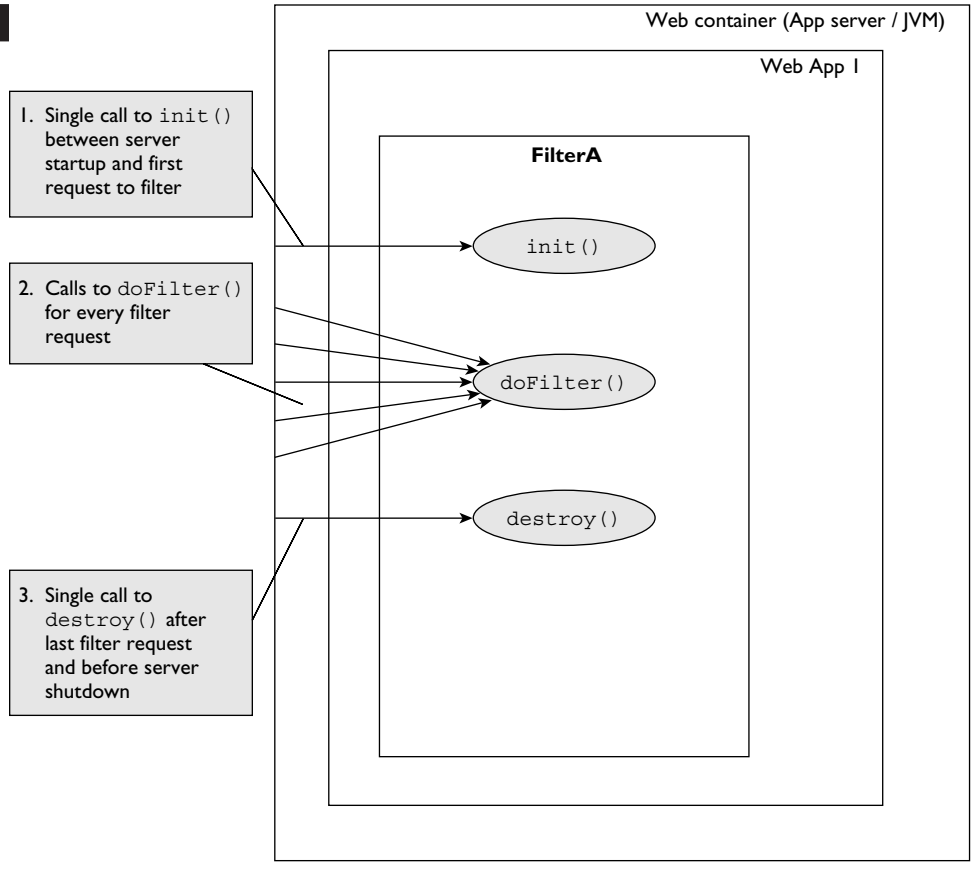
`init()` The full signature is

```
public void init(FilterConfig config) throws ServletException.
```

The `init()` method is called once only—when the web container creates the instance of the filter. This could be on server startup, and at latest will occur just before the filter is pressed into service (because someone has requested a web resource

FIGURE 3-3

Filter Life Cycle



that triggers the filtration). You have one shot at this point to capture the Filter Config object that is passed as a parameter to the method and to keep it available for later use—typically as a private instance variable. Here's an extract from a Filter which does just that:

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

There's no compulsion to do this, but the FilterConfig object has some handy methods that you might want to use later:

- `getFilterName()`, which returns a String returning the name of the filter as defined in the deployment descriptor.
- `getInitParameter(String name)`, which returns a String value for the named parameter. This is identical in concept to the mechanism for setting up ServletContext parameters, which we met at the beginning of this chapter. Unsurprisingly, FilterConfig also has a `getInitParameterNames()` method, returning an Enumeration of the names of all the initialization parameters defined for this filter.
- `getServletContext()`, which returns a handle to the servlet context for the web application.

You can use the `init()` method to do other initialization tasks you deem necessary. You're not restricted in any way as to what these might be: Do whatever the Java language lets you do.

`destroy()` The full signature is `public void destroy()`: no parameters in, nothing returned. You are guaranteed that this method will be called once and once only when the filter is taken out of service, which means, usually, when the web application closes down. This gives you the opportunity to do some cleanup and resource reclamation, typically unpicking the initialization you performed in the `init()` method.

`doFilter()` The full signature is lengthy and is very close to a servlet's `service()` (or `doGet()` or `doPut()`) method.

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

So you can see that—like a servlet—the method accepts a request and a response object. There's another object as well, though—the FilterChain object—and it's this that allows a filter to pass control (or deny access) to other filters and web resources. Here's an abbreviated list of what you generally do in a `doFilter()` method:

1. Look at the request.
2. Wrapper the request and response object—if required.
3. Add or change things about the request, through the wrapper.
4. Call the next filter (or servlet) in the chain (using `doFilter()` on the Filter Chain object passed as a parameter), *or*

5. Block the request by *not* calling the FilterChain's `doFilter()` method.
6. On return from the FilterChain's `doFilter()` method (or even if it wasn't called), amend the response—headers or content—through the wrapper.

Constructor If you wish, you can supply a no-argument constructor (or rely on the default constructor that the Java compiler provides in the absence of other constructors). There's no point at all in providing constructors with arguments: After all, the web application model is the framework that instantiates filters, and the framework is not set up to call constructors with arguments.

An Example Filter

So now let's take a look at an example filter. It's a logging filter that makes no attempt to alter the request and response objects it receives. It simply writes the URL (and other default logging details) to a named log file. Here's the code:

```
10 import java.io.*;
11 import java.util.logging.*;
12 import javax.servlet.*;
13 import javax.servlet.http.*;
14 public class LogFilter implements Filter {
15   private FilterConfig config;
16   private static Logger logger = Logger.getLogger("com.osborne.accesslog");
17   public void init(FilterConfig config) throws ServletException {
18     // Initialize the logger
19     try {
20       Handler fh = new FileHandler("C:\\temp\\accessLog.txt");
21       logger.addHandler(fh);
22     } catch (IOException ioe) {
23       throw new ServletException(ioe);
24     }
25     logger.setLevel(Level.INFO);
26     // Capture the config object
27     this.config = config;
28   }
29   public void doFilter(ServletRequest request, ServletResponse response,
30     FilterChain chain) throws IOException, ServletException {
31     HttpServletRequest httpReq = (HttpServletRequest) request;
32     String path = httpReq.getRequestURI();
33     logger.info("The following path was requested: " + path);
34     chain.doFilter(request, response);
35   }
36   public void destroy() {
```

```
37      logger = null;
38      config = null;
39    }
40 }
```

Let's talk through some parts of this code:

- Line 14 declares the class—called LogFilter—and shows that it implements the Filter interface.
- Line 15 declares an instance variable of type FilterConfig. We go on to initialize this from the parameter passed to the `init()` method at line 27, just in case we wanted to make use of the FilterConfig object in the filter (which we don't, as it happens—but you might amend this code and choose to do so later!).
- Line 16 declares a static variable of type Logger, from the javax.util.logging package. Space doesn't permit a full explanation of Java logging—take a look at the Java Logging Overview in the J2SDK documentation.
- Lines 17 to 28 encompass the `init()` method. Apart from trapping the FilterConfig parameter, as we discussed earlier, the code here is devoted to setting up the Logger: tying this to a file on the file system called accessLog.txt (in directory C:\Temp) and setting it to receive informational (or more serious) messages.
- Lines 29 to 35 make up the `doFilter()` method. At line 29, this accepts a standard parameter: request, of type ServletRequest. Since we know we will be running this filter in an HTTP environment, we know it's safe to cast the parameter to an HttpServletRequest reference at line 31. This enables us to execute the `getRequestURI()` method at line 32 to get a String showing the web resource requested. This we pass as a parameter into the logger's `info()` method at line 33, so it's written to the access log. Finally—at line 34—we simply call the FilterChain's `chain()` method, passing on the request and response entirely unaltered.
- Lines 36 to 40 cover the `destroy()` method, which cleans up by setting references to *null*.

Defining Deployment Descriptor Elements for Filters

So now that we have written our Filter, all we have to do is to ensure that the web container will call it when we require. This is achieved through setting up `<filter>` and `<filter-mapping>` elements in the deployment descriptor. Figure 3-4 shows graphically how the `<filter>` element looks.

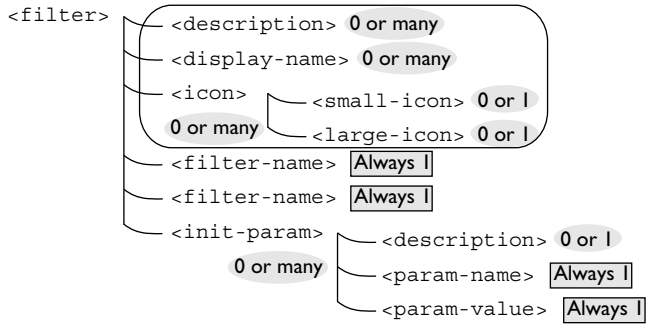


FIGURE 3-4

Filter Declaration in the Deployment Descriptor

The first three optional trio of elements—`<description>`, `<display-name>`, `<icon>`—are no different in function or form from other places where they occur (see Chapter 2, Figure 2-3, and the accompanying explanation), except, of course, that they apply specifically to the filter you're setting up.

The meat of the `<filter>` element is in the two subelements `<filter-name>` and `<filter-class>`, which are mandatory. Here you give your Filter a logical name (which can be used to tie into later filter mappings) and the full qualified name of your filter's class file.

Optionally, supply as many `<init-param>` elements as you like, and use the FilterConfig's `getInitParameter()` and `getInitParameterNames()` methods to get at them in your Filter code.

Let's now take a look at `<filter-mapping>`, which is shown in Figure 3-5. This is a little more complex than `<servlet-mapping>`, which we met in Chapter 2. You see that it has three subelements, two of them mandatory. The first is `<filter-name>`, which must tie back to a `<filter-name>` specified in a `<filter>` element. The second subelement is also mandatory, but you have a choice: either `<url-pattern>` or `<servlet-name>`. This is the element that actually ties your filter to an incoming request for a web resource.

- `<url-pattern>`: Suppose your client request has a URL that matches the `<url-pattern>` on a particular `<servlet-mapping>`. Suppose then that this same `<url-pattern>` matches a `<filter-mapping>` as well. That's the trigger for the filter to run ahead of the servlet that has been targeted. The rules for legal filter mapping URL patterns are exactly the same as those embedded in servlet mappings: We explored them in Chapter 2. We'll see some example URL patterns for filters a little later in this section.
- `<servlet-name>`: The servlets in your web application have a `<servlet-name>` as a mandatory subelement of `<servlet>`. If the filter mappings's

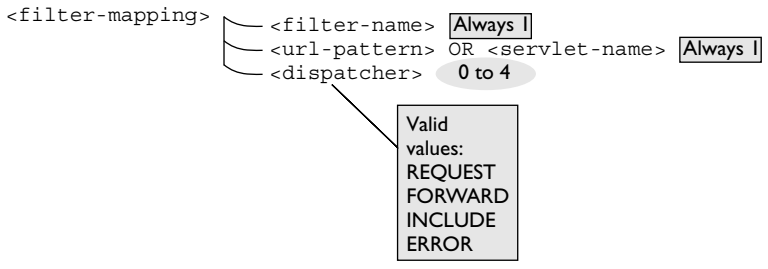


FIGURE 3-5

Filter Mapping Declaration in the Deployment Descriptor

`<servlet-name>` matches a requested servlet's `<servlet-name>`, it's a trigger for the filter to run ahead of the servlet.

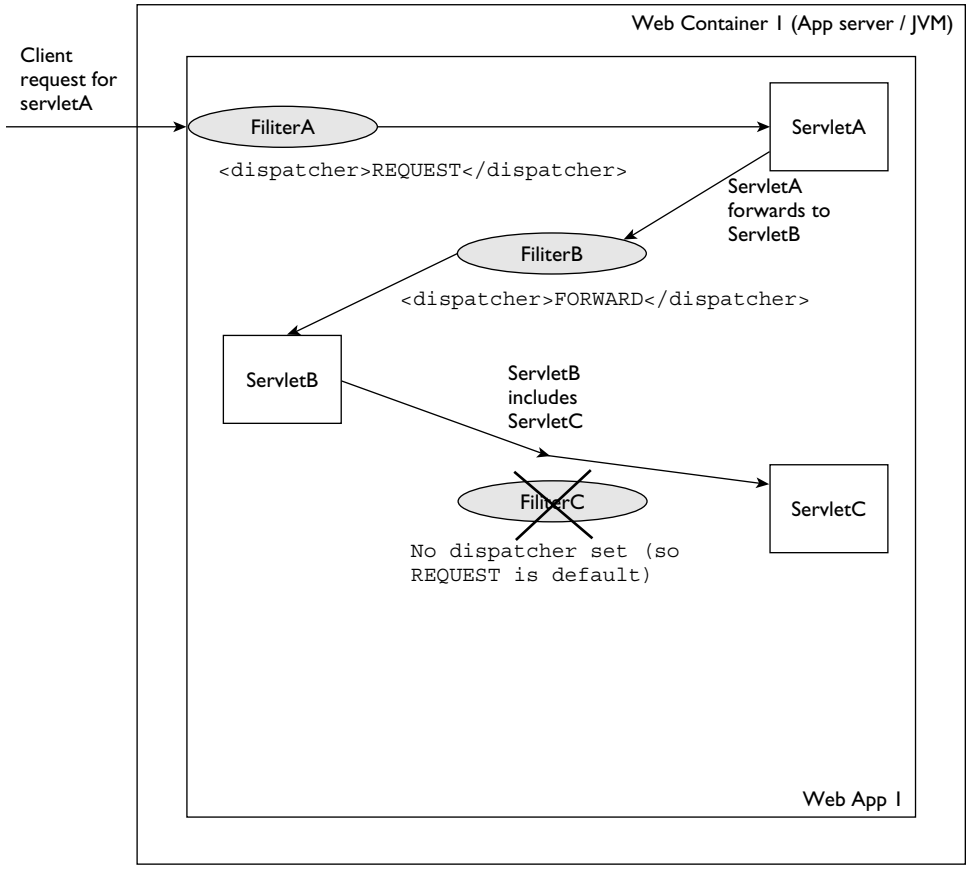
The third subelement—`<dispatcher>`—is optional; when you leave it out, though, it's equivalent to explicitly stating `<dispatcher>REQUEST</dispatcher>`. This feature—introduced in servlet specification 2.4—acknowledges that there are many routes into a web resource on your web application. It could well be that you want the filter to kick in dependent on one of these routes. Here are the four valid values for the dispatcher element, with a description of the route:

- REQUEST (the default)—a direct client request for a web resource.
- FORWARD—an internal web server request for a web resource via the `forward()` method on a RequestDispatcher.
- INCLUDE—an internal web server request for a web resource via the `include()` method on a RequestDispatcher.
- ERROR—an internal web application request for a resource that has been set up as an `<error-page>`.

When you supply a `<dispatcher>` value, you are giving permission for a filter to trigger for the route specified. The normal situation is probably that you want your filter to apply only to bona fide external client requests. If a servlet is called internally via a RequestDispatcher, chances are you want that servlet to run without the filter intervening. But if you do want the filter to run even on RequestDispatcher calls as well as client requests, include

```
<dispatcher>REQUEST</dispatcher>
<dispatcher>FORWARD</dispatcher>
<dispatcher>INCLUDE</dispatcher>
```

in your `<filter-mapping>`. Here's an illustration of this. FilterA applies to ServletA, FilterB to ServletB, and FilterC to ServletC.



FilterA is explicitly set up to fire when its corresponding ServletA gets a client request, so FilterA runs and calls ServletA. ServletA forwards to ServletB. Because FilterB (attached to ServletB) allows forwarding requests, FilterB runs and calls ServletB as the next item in the chain. Now ServletB includes the output of ServletC. Even though FilterC attaches to ServletC, it won't run—because it will reject "includes" as a valid route in. So FilterC is bypassed, and ServletC is called directly by ServletB.

Here's how a complete deployment descriptor might look for the Log Filter we set up earlier:

```
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>com.osborne.LogFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>
```

You see how the URL pattern for the LogFilter is "/." This is the catchall: Whatever resource is requested matches this mapping, so the LogFilter will trigger—at least for direct client requests.

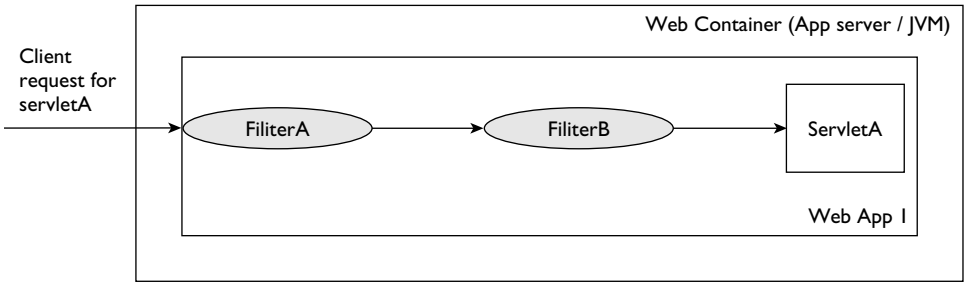
Stacking Filters

What if we want to run not just a LogFilter for every request but an Authorization Filter as well? Well, the answer is that we can. Simply stack them up in the deployment descriptor:

```
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>com.osborne.LogFilter</filter-class>
</filter>
<filter>
  <filter-name>AuthorizationFilter</filter-name>
  <filter-class>com.osborne.AuthorizationFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>AuthorizationFilter</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>
```

In this case, both filters have the same generic mapping: The URL pattern matches any request that comes in. Which will run first? That's determined by the order

of `<filter-mapping>` declarations in the deployment descriptor. Suppose a client requests ServletA in this web application. The "filter chain" formed in this case will be



The general case is this: Given a request that matches more than one filter mapping,

- First, all matching filters will run for `<filter-mappings>` with `<url-pattern>` matches, in order of `<filter-mapping>` declaration.
- Second, all matching filters will run for `<filter-mappings>` with `<servlet-name>` matches, in order of `<filter-mapping>` declaration.

exam Watch

Note that filters will run equally well in front of (or after) static content (such as a plain-vanilla HTML file); they are not solely for fronting (or backing) servlets and JSPs.

INSIDE THE EXAM

Servlet/Filter Comparison

It's uncanny how many parallels there are between servlets and filters (well, not really—it's the product of intentional design!). For exam purposes, I find it really helpful to explore the similarities and also the nuances of difference: It reinforces my memory of both. The table below should get you started on this process.

INSIDE THE EXAM (continued)

	Servlet	Interface
Type	Typically built by extending the javax.servlet.GenericServlet or javax.servlet.http.HttpServlet class.	Built by implementing the javax .servlet.Filter interface.
Construction	Optional, no-argument constructor (not often used)	Same.
Initialization	Override the `init(ServletConfig con-fig)` method — or the plain init() method.	Implement the `init(FilterConfig config)` method.
Destruction	Override the `destroy()` method.	Same.
Parameters	Can have initialization parameters declared in the deployment descriptor.	Same.
Declaration element in deployment descriptor	Declared in a `<servlet>` element, which contains `<servlet-name>` and `<servlet-class>` elements.	Declared in a `<filter>` element, which contains `<filter-name>` and `<filter-class>` elements.
Mapping element in deployment descriptor	Declared in a `<servlet-mapping>` element, which contains `<servlet-name>` and `<url-pattern>` elements.	Declared in a `<filter-mapping>` element, which contains `<filter-name>` and `<url-pattern>` elements. `<servlet-name>` can be substituted for `<url-pattern>`.
Instantiation	There will be one instance of a servlet per `<servlet>` element in the deployment descriptor. The same actual class may have separate `<servlet>` declarations: Each occurrence will result in a separate instance of this class.	There will be one instance of a filter per `<filter>` element in the deployment descriptor. The same actual class may have separate `<filter>` declarations: Each occurrence will result in a separate instance of this class.

exam Watch

A Filter can throw an UnavailableException, which is a subclass of ServletException. UnavailableException has an `isPermanent()` method: If this returns* true, *the web container gives up calling the filter; if this returns* false, *the web container will try again after a specified time interval. Whether it will return true or false depends on how the exception is created. If you use the version of the constructor that simply accepts a String message, the exception is construed as permanent. If, however, you use the two-parameter constructor that accepts a String message and an "int seconds," the web container should deem the exception as temporary and try to call the Filter again after the specified number of seconds.

on the Job

The drawback with filtering static content is that every request to your web server has to be processed by the web container. Big production applications normally consist of a straight web server (such as Apache) that forward requests to J2EE-aware web containers only when it's necessary. That leaves them free to serve the static content more efficiently: There's no additional "hop" to get the information. If you want all static content to be subject to Filter processing, then there is nothing for it but to make the additional round trip to the J2EE-aware web container. There's nothing wrong with that if you really need the filter processing for every piece of static content you serve up. However, place only the static content you need to under J2EE control: Leave the rest for plain-vanilla web serving.

Filtering vs. Dispatching

You might wonder what the point is of using filters at all. Why not just use a chain of servlets that dispatch from one to another? Before the invention of filters (at servlet specification level 2.3), that's exactly what happened. Methods existed (now deprecated) to construct a servlet chain. Filtering was intended to provide a more flexible replacement. So what advantages are there?

- A filter chain can be reshuffled fairly easily by moving entries up or down in the deployment descriptor. You can easily insert additional filters at a later stage, without any programming required.
- Filters can trap requests for any kind of resource, again with no programming required to forward on the request.

Wrappers

In our LogFilter example, we were only interested in trapping the request to log a URL to an audit file. No attempt was made to change the request or response. Finally in this chapter, we are going to consider how you should program a Filter when you do want to make such an intervention. That's where we need to deftly substitute a wrapper class in the `chain.doFilter()` invocation.

Why We Need a Wrapper

Let's consider the following snippet from a longer `doFilter()` method that doesn't use a wrapper, but nonetheless writes to the response object:

```
chain.doFilter(request, response);
PrintWriter out = response.getWriter();
out.write("<BR />A line of text at the bottom of your web page");
```

If you try out this filter code in front of a servlet of your choosing, it stands a fair chance of working— outputting the line of text promised. However, what if the servlet code that is the target of the `chain.doFilter` call does the following?

```
PrintWriter out = response.getWriter();
out.write("<BR />This is the servlet speaking");
out.close();
```

You don't have to close the PrintWriter, but a scrupulous servlet developer might well do so in the spirit of tidy resource management. In this case, the filter code will run without failing, but the line of text will no longer appear at the bottom of the web page. How can we ensure it does? Use a response wrapper, which in this case will be a class you write that subclasses javax.servlet.http.HttpServletResponse Wrapper.

Four Wrappers from Which to Choose

There are four wrapper classes that you might choose to subclass according to circumstance:

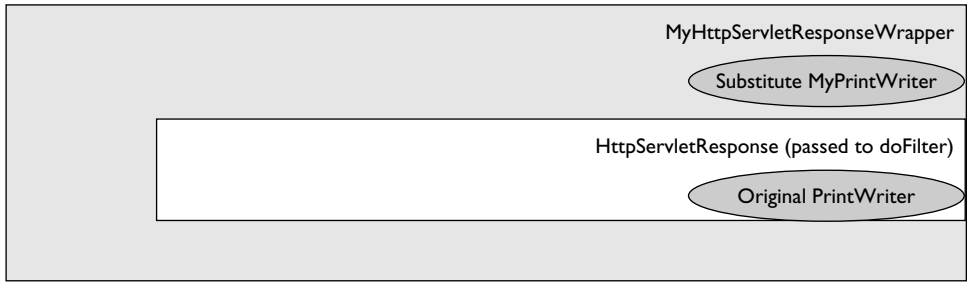
- javax.servlet.ServletRequestWrapper
- javax.servlet.ServletResponseWrapper
- javax.servlet.http.HttpServletRequestWrapper
- javax.servlet.http.HttpServletResponseWrapper

It's self-evident from the names which ones are used to wrapper requests and which ones to wrapper responses, and which ones pertain to HTTP web containers as opposed to plain servlet containers.

So let's return to that problem where we want to prevent the closing of the Print Writer. The solution is a little involved, but not too challenging:

- Write a subclass of PrintWriter called MyPrintWriter. Override the `close()` method—to do nothing. Optionally, write a `trueClose()` method that calls the superclass `close()` method (i.e., the one in PrintWriter that actually does the closing).
- Write a subclass of HttpServletResponseWrapper called MyHttpServletResponse Wrapper. This should contain an instance variable of MyPrintWriter type. Override the `getWriter()` method to return this instance variable. Make sure you reproduce the single-parameter constructor from HttpServletResponseWrapper (which takes an HttpServletResponse as its argument).
- Back in the `doFilter()` method of the Filter class, create a MyHttpServlet ResponseWrapper using the single-parameter constructor. Pass to this the response that comes as a parameter to the `doFilter()` method.
- Call the `chain.doFilter()` method, with the request (unchanged) and the wrappered response.

We'll see the code for this in a moment, but let's briefly reflect on the design pattern in use here, known most often as the "decorator" pattern (if "pattern" is an unfamiliar term, then take comfort that patterns are the subject of Chapter 10). You take a class, then wrap around it another class, which might mimic, extend, or change the functionality. It's exactly the same principle at work as in those java.io classes that you learned about for your SCJP exam, where you used constructors to nest InputStreams in BufferedStreams (one of the myriad possibilities). So you have the genuine response muffled by your own response wrapper class, like this:



Now we'll see how this looks in code. First MyPrintWriter. This shows the essential "do nothing" `close()` method, and also one constructor. This copies a signature from one of its parent PrintWriter constructors, and simply calls the parent constructor using `super()`.

```
public class MyPrintWriter extends PrintWriter {
  public MyPrintWriter(Writer out) {
    super(out);
  }
  public void close() {
    // do nothing
  }
}
```

More interesting is the MyHttpServletResponseWrapper class. This also calls its superclass constructor because the parent HttpServletResponseWrapper is already set up to do the response wrappering part. Then comes the crafty code to do some more wrappering: creating a new MyPrintWriter object by passing in the writer from the original HttpServletResponse. This is held as an instance variable. Now any unsuspecting call to `getWriter()` will return a MyPrintWriter object—unbeknown to the other servlets and filters using it somewhere down the chain.

```
public class MyHttpServletResponseWrapper extends
HttpServletResponseWrapper {
    private MyPrintWriter out;
    public MyHttpServletResponseWrapper(HttpServletResponse response) {
        super(response);
        try {
        out = new MyPrintWriter(response.getWriter());
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
    public PrintWriter getWriter() throws IOException {
      return out;
    }
}
```

Now here's the relevant code from the `doFilter()` method of the Filter that wants to do the wrappering.

```
01 HttpServletResponse httpResponse = (HttpServletResponse) response;
02 MyHttpServletResponseWrapper wrapperedResponse =
     new MyHttpServletResponseWrapper(httpResponse);
```

```
03 chain.doFilter(request, wrapperedResponse);
04 PrintWriter out = wrapperedResponse.getWriter();
05 out.write("<BR />This was put here by the WrappingFilter");
```

Because the response passed into the `doFilter()` method is a ServletResponse, line 1 does some discrete casting to an HttpServletResponse, for we know that this filter will be used only in an HTTP environment. Line 2 creates the wrapper for the response, by passing in the HttpResponse instance to the constructor of our newly created response wrapper class. Line 3 actually does the call to the next `doFilter()` in the chain—but notice that while the request is left alone, the wrapperedResponse is put in place of the response variable. On return from whatever is invoked by the chain, the filter proves it has worked by adding an extra line of HTML text to the response. Finally, here's some servlet code that won't be able to close the writer when invoked through this filter code:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.write("<HTML><HEAD><TITLE>ItsAWrap Servlet</TITLE></HEAD><BODY>");
out.write("<H1>This servlet attempts to close the PrintWriter!</H1>");
out.write("<P>But as the response is wrappered, it doesn't succeed.</P>");
out.write("</BODY></HTML>");
out.flush();
out.close();
```

Because the response received by the servlet is a MyHttpServletResponseWrapper object, the PrintWriter obtained by the code is in fact of type MyPrintWriter. So when—at the end—the overridden `close()` method is called, it does nothing. Here's the output as I see it in my browser:

One thing you might observe about the code above is that it results in improperly formed HTML. The page has already been terminated with `</body>` and `</html>` tags, and then the filter adds an additional line of text. Most browsers are fault-tolerant of such sloppiness, and just display the text anyway. But the real point is that the response wrappers you write should be more sophisticated than this simple

example; you should unpick and rework responses as is necessary for well-formedness or other requirements.

EXERCISE 3-4

Using a Filter for Micropayments

This exercise gets you to build a filter that makes micropayments. It's software to support fortune-making referral schemes. You know the kind: If someone makes a request to your web application, the filter will deposit a fraction of a cent in the referrer's PayPal account. We'll set the filter up to make the payment when you invoke a servlet called MicroPaymentServlet in your web application. The context directory for this exercise is ex0304, so set up your web application structure under a directory of this name.

For this exercise, there's a solution in the CD in the file sourcecode/ch03/ex0304.war—check there if you get stuck.

Set Up the Deployment Descriptor

1. Declare a servlet named MicroPaymentServlet, with a generic URL mapping (e.g., "/MicroPayment/*") so that you can trap path information. If needed, refer to Chapter 2 to refresh yourself on `<servlet>` element setup.
2. Declare a filter named MicroPaymentFilter that is tied to the named servlet MicroPaymentServlet in its filter mapping. It doesn't actually matter if the `<filter>` and `<filter-mapping>` elements come before or after the `<servlet>` and `<servlet-mapping>` you have already set up. However, to stay compatible with the old rules, I would be inclined to place filter elements before servlet elements.

Write the MicroPaymentFilter

3. Create a Java source file MicroPaymentFilter.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, ensuring that it implements the Filter interface.
4. Override the `doGet()` method in MicroPaymentServlet.
5. Supply "do nothing" `init()` and `destroy()` methods. You may prefer at least to put in some code to hang on to the FilterConfig object even though we won't use it in this exercise, but it gets you into good habits:

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

6. Write a `doFilter()` method. As the first thing you do in this method, take the path information from the request: Use this as the value for a request attribute you set up called "referrer." You may want to strip out the leading slash from the path information.
7. Still in the `doFilter()` method, call the `doFilter()`on the FilterChain object. Pass on the request and response (unwrappered).
8. Still in the `doFilter()` method, and after the chain call in step 7, get the PrintWriter from the response object. Output some text to indicate that the micropayment has been made.

Write the MicroPaymentServlet

9. Create a Java source file MicroPaymentServlet.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
10. Override the `doGet()` method in MicroPaymentServlet.
11. Get hold of the referrer name—this will be the value pulled from the request attribute "referrer" that you set up in the filter during step 6.
12. Write some HTML output (set the response appropriately and obtain the response's writer). It doesn't matter what the web page says, but at least include the referrer name.

Run the Application

13. Deploy and run the application. Use a URL containing appropriate path information such as the one below—in place of "referrerName" at the end of the path, substitute your own first name:

    ```
    http://localhost:8080/ex0304/MicroPayment/referrerName
    ```

14. Optional: Alter the MicroPaymentServlet so that it closes the response's PrintWriter, and redeploy and run the code again. My findings are that no exceptions are thrown, but the words that MicroPaymentFilter writes to the web page do not appear. This is because the writer is already closed by the time the filter code gets hold of it—and demonstrates why you are better off working with wrapper classes. With a (response) wrapper that is a subclass of HttpServletResponseWrapper, you can substitute your own writer. This writer can override the `close()` method to prevent closing; you can always have a method of a different name that truly closes the writer back in the filter code.
15. Optional: Have the filter operate on any servlet in the web application by changing the `<servlet-name>` element to a `<url-pattern>` of "/" instead. Create another servlet that dispatches to the MicroPaymentServlet. Change the filter so that it comes into play only when another servlet dispatches to the MicroPaymentServlet, but not when the MicroPaymentServlet is requested directly. Hint: You'll want appropriate `<dispatcher>` elements in your filter mapping definition.
16. The following illustration shows how typical output looks with a referrer name (see step 13) of David:

CERTIFICATION SUMMARY

In this chapter we covered a great deal of ground—all under the umbrella subject of the "web container model." We started with an easy topic: how to set up initialization parameters on the ServletContext object. We saw that the process is not dissimilar from setting up servlet initialization parameters. We met a new deployment descriptor element—`<context-param>`—which houses a `<param-name>` and `<param-value>` element pairing and can appear for as many parameters as you need inside the `<web-app>` root element. We covered the two ServletContext methods you can use—`getInitParameter(String name)`, to get hold of individual parameters by name, and `getInitParameterNames()`, to get an Enumeration of all names of context parameters to be found in a web application.

We widened the scope then—in every sense!—by looking at the three scopes available in a web application: request, session, and context. We saw that request scope addresses a single client request to the web application. Request scope begins on entry to a servlet's `service()` method, ends on exit from the method, and lasts through whatever else the servlet might invoke during the `service()` method. We had a tantalizing, pre–Chapter 4 glimpse of session scope, and learned that this offers continuity over a series of requests from the same client. Finally, we discussed context scope, represented by a single ServletContext object, and saw that this object is available during the whole of a web application's life.

We saw that a principal reason for having scopes—and objects representing them—is to hold information important to the application, in the form of attributes. We saw that attribute values aren't restricted to Strings; an object of any type can be held as an attribute. We met the fundamental methods used to manipulate attributes—`getAttribute(String name)`, to get hold of the object value for a single named attribute, and `setAttribute(String name, Object value)`, to put a key/value pairing into a given scope. We also saw `getAttributeNames()`, returning an Enumeration of all the names of attributes for a scope, and `removeAttribute(String name)`, to delete an attribute from a scope. We saw that these four methods are available on each of the three objects representing scopes: ServletRequest (for request scope), HttpSession (for session scope), and ServletContext (for context scope).

We then touched on the web container as a multithreaded environment, able to service many client requests simultaneously. We saw the impact this has on the attributes set up in different scopes. We saw that because a single request is confined to a single thread in the web application, request attributes are thread safe. You took my word for it that session attributes are not quite thread safe and that we would

learn more about session scope and multithreading in Chapter 4. You appreciated that context attributes cannot possibly be thread safe, for they are available to any thread running in the web application—or even in other web applications, because one web application can get hold of another's context.

We then moved on to the subject of dispatching. We found that it was possible to use a RequestDispatcher object to obtain the services of some other resource in the web application. We also saw that this could be any resource, static (such as an HTML page) or—more usually—dynamic (another servlet or JSP). You learned that there are two ways of getting a RequestDispatcher object: either through the `getRequestDispatcher()` method on ServletRequest or through the `getRequestDispatcher()` or `getNamedDispatcher()` method on ServletContext. We discussed the nuances of these methods: how the `getRequestDispatcher()` method on ServletRequest is the most flexible because it can accept paths beginning with a forward slash or not, and how the same method on ServletContext can accept only paths beginning with a forward slash. We also saw how `getNamedDispatcher()` on ServletContext is used in an entirely different way—to obtain a servlet or JSP identified by `<servlet-name>` within the deployment descriptor's `<servlet>` elements. We also learned that it might make sense to have a `<servlet>` without a `<servlet-mapping>` so that any access to this can be controlled through the `getNamedDispatcher()` method, while it remains impervious to direct client requests.

We identified that all paths fed to the `getRequestDispatcher()` methods are bound to be inside the web application to which the request dispatcher belongs. Paths beginning with a forward slash ("/") are relative to the context root of the web application. Paths without the initial forward slash are relative to the path of the resource invoking the RequestDispatcher, and can't use "`..`" (double dot: go up to parent directory) to escape the context root. However, we did learn that request dispatchers can come from other web application contexts, if web server security allows access to them through the `ServletContext.getContext(String otherContext)` method.

After a lot of discussion about how to get RequestDispatcher objects, we finally learned how to use them—by invoking the `forward()` or `include()` method. We saw that the `forward()` method effectively hands responsibility to the target of the request dispatcher: The forwarded-to servlet (or JSP) has sole responsibility for producing the response. We learned that the `include()` method does as its name implies—it includes the output of the target inside the response of the servlet doing the including. We saw that the forwarded-to or included servlets have access to special request attributes (such as javax.servlet.forward.context_path and javax.servlet

.include.path_info). We found that the information in these attributes supplements methods on the request, such as `getContextPath()` and `getServletPath()`, because these methods cannot return information both about the request to the dispatching and the request to the dispatched-to servlet at one and the same time. As an aside, we learned that these special request attributes are not present when the dispatcher is obtained as a named dispatcher—because this is seen as an internal request, not properly associated with a request URL. Finally, we saw how important it is not to call `forward()` after a response has been committed, and that doing so leads to an IllegalStateException.

The final topic in this web container model chapter—and the most complex—was about filters and wrappers. We learned how filters can be used to pre-process a request for a web resource, or to completely transform the response, even supplying entirely alternative responses when the need arises. We listed common uses for filters, such as encryption, caching, logging, authentication, and XML transformation. We then looked at the mechanics of filter creation: the classes you have to write and the deployment descriptor elements you have to set up. We saw that a filter is a class that implements the Filter interface, with its three life cycle methods: `init()`, `doFilter()`, and `destroy()`. We learned that `init()` and `destroy()` are called only once apiece: `init()` before the first request for a filter and `destroy()` after the last request has been processed (on web application shutdown). We saw that every request to the filter results in a call to the `doFilter()` method. We learned about the FilterConfig object, passed to the filter's `init()` method, and saw how this can be used to get at the filter name, the servlet context, and information about any initialization parameters set up for the filter.

We then looked at the deployment descriptor requirements for filters, comprising `<filter>` and `<filter-mapping>` elements. We saw that `<filter>` has some minor and some crucial subelements, the mandatory ones being `<filter-name>` (to supply a logical name for the filter) and `<filter-class>` (for the fully qualified class name of the filter); `<init-param>` is a nonmandatory subelement for setting up initialization parameters. We tried out `<filter-mapping>` elements, which have to have a `<filter-name>` subelement matching an existing filter, and most crucially, a `<url-pattern>` or `<servlet-name>` subelement as the means for making a match to the filter. We learned that the rules for a `<url-pattern>` mapping are identical to those for servlets. We saw that multiple filter mappings can match a given request and that the web container assembles these into a chain, with the web container processing matches by `<url-pattern>` first and `<servlet-name>` next—both sweeps based on the order of the matching `<filter-mapping>` elements in the deployment descriptor.

We saw then how the chain is started by the web container calling the first filter's `doFilter()` method, passing in the request, response, and FilterChain object—this last parameter representing the next item in the chain. We learned that this next item can be another filter or the resource targeted by the request when there are no more filters left in the chain. We wrote code that executed `chain.doFilter()` (from within Filter's `doFilter()` method) and learned that only by including this call would the next item in the chain execute.

We learned that when a filter wants to alter the request or response, it should wrapper up the original request and response objects passed as parameters to the `doFilter()` method. We saw how to subclass an appropriate wrapper class and override methods on this class (or add new ones) if we want to customize request or response behavior. We saw how to replace the original request or response with the wrappered class in the `chain.doFilter()` method call.

Finally, we learned that filters will—by default—trigger only on direct client requests. However, we saw that we can make filters trigger through request dispatcher calls to forward or include, and through the error page mechanism. We learned that this can be achieved by adding `<dispatcher>` subelements to `<filter-mapping>`, with appropriate values of FORWARD, INCLUDE, or ERROR. We saw that as a consequence of this, we have a fourth valid value for `<dispatcher>` of REQUEST and that this has to be included whenever one of the other three values is used, if the filter is still to trigger on direct client requests.

TWO-MINUTE DRILL

ServletContext

- ❏ The ServletContext is the closest thing your application has to an object that represents the web application itself.
- ❏ You can use the deployment descriptor to set up initialization parameters at ServletContext level. Each initialization parameter is housed in a `<context-param>` element; each one of these elements contains a mandatory `<param-name>` and `<param-value>` set of elements.
- ❏ The parent element for `<context-param>` is the root element `<web-app>`: This makes perfect sense, for ServletContext parameters belong at the web application level.
- ❏ The `getInitParameter(String parmName)` method on ServletContext is used to retrieve a parameter value whose name is known.
- ❏ The `getInitParameterNames()` method on ServletContext is used to retrieve an Enumeration of all context parameter names known to the web application.
- ❏ You can have as many context parameters as you want (none, one, several, or many).

Attributes, Scope, and Multithreading

- ❏ Attributes are a two-way street: You can set them as well as get them in your code.
- ❏ The web container can set attributes as well as your code.
- ❏ Attributes are not the same thing as parameters. Parameters flow into your application—from the client request or from deployment descriptor elements—and are read-only. Attributes flow within your application and can be read, created, updated, and deleted.
- ❏ There are three fundamental scopes—request, session, and context. (There is a fourth scope—page—which you learn about with JavaServer Pages from Chapter 6 onward.)
- ❏ Attribute manipulation methods look and behave almost identically, whatever the scope. There are four relevant methods: `getAttribute(String`

`name)`, `getAttributeNames()`, `setAttribute(String name, Object value)`, and `removeAttribute(String name)`.

- Attribute methods don't throw exceptions (with one small exception for session-related attribute methods).
- A call to `getAttribute(String name)` can result in a null object reference being returned when the object doesn't exist.
- A call to `getAttributeNames()` will always result in a valid reference to an Enumeration, though the Enumeration itself may be empty if there are no attributes for the scope.
- You can only have one attribute of a particular name. Subsequent calls to `setAttribute(String name, Object value)` for the same name will overwrite the previous value. A value of **null** will remove the attribute (having the same effect as a call to `removeAttribute(String name)`).
- Request scope begins on entry to a servlet's `service()` method and ends on exit from that method.
- Request scope is bound to a single thread, so it's thread safe.
- Request scope is represented by the HttpServletRequest (or ServletRequest) object.
- Session scope exists across multiple requests from the same client to the same web application.
- Session scope is represented by the HttpSession object, obtainable using the `HttpServletRequest.getSession()` method.
- There is no "non-HTTP" equivalent of session scope.
- Context scope is sometimes thought of and referred to as web application scope.
- Context scope is represented by the ServletContext object, obtainable through the `getServletContext()` servlet method.
- The web container provides one ServletContext object per web application per JVM. So if the web application is distributed, the servlet context objects in different clones of the application are separate.
- Context scope lasts from when a web application is put into service to the point where it is removed from service.
- Context attributes are not thread safe: Practically every thread in your web application can access them.

Dispatching

- Dispatching is a means of delegating control from one web resource to another.
- You use a RequestDispatcher to represent the web resource to which you want to delegate. A RequestDispatcher can be obtained from one of two places: the ServletRequest or the ServletContext.
- ServletContext has two methods for obtaining a RequestDispatcher: `getRequestDispatcher(String pathFromContextRoot)` and `getNamedDispatcher(String nameOfServlet)`.
- ServletRequest has only one method for obtaining a RequestDispatcher, which is also `getRequestDispatcher(String path)`. The difference from the ServletContext method of this name is that the method will accept a path relative to the context root *or* a path relative to the current resource.
- A path relative to the context root begins with a forward slash ("/") . You don't include the context root name itself when forming a path like this.
- A path relative to the current resource does not begin with a forward slash and is relative to the client request URI within the context. So for a client request to /webappname/servlet/ServletA, a relative path of "ServletB" would translate to a path relative to the context root of /servlet/ServletB.
- Paths used as parameters to the three RequestDispatcher methods are restricted to the context to which the RequestDispatcher belongs. You can't go outside a single web application.
- You can obtain RequestDispatchers for other web applications by obtaining another web application's context and getting a RequestDispatcher from that (but note that your web container's default security settings may prevent you, by causing request dispatchers returned from other contexts to be **null**).
- Having obtained a request dispatcher, you can call either the `forward()` or `include()` methods on it, passing in the ServletRequest and Servlet Response objects.
- The `forward()` method passes responsibility to another web resource. The response will come entirely from the target of the forward. Any work that the *forwarding* servlet has done on the response will be discarded.
- The `include()` method includes the output of the included web resource inside the including servlet. On return from the `include()` call, the including servlet can still add more to the response.

- If a response has already been committed in a servlet, a call to `forward()`will result in an IllegalStateException.
- If a response has already been committed in a servlet, a call to `include()` will still work.
- A forwarded-to servlet has access to five special request attributes, which describe the state of the request in the *forwarding* servlet. The special attribute names all begin javax.servlet.forward.
- An included servlet has access to five different special request attributes, which describe the state of the request in the *included* servlet. The special attribute names all begin javax.servlet.include.

Filters and Wrappers

- Filters are used for pre-processing requests or post-processing responses, before they reach a target resource in a web application.
- Common uses for filters include authentication, logging, data compression, encryption, and caching.
- Filters you write must implement the javax.servlet.Filter interface. This has three methods—`init(FilterConfig config)`, `doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`, and `destroy()`.
- A filter's `init()` method is called only once, at some point between server startup and definitely before the first request is intercepted by the filter.
- A filter's `doFilter()` method is called by the web container whenever it intercepts an appropriate request for the filter.
- A filter's `destroy()` method is called only once, at some point after the last filter request is processed in the `doFilter()` method, and before the web application closes down.
- A typical use of the `init()` method is to capture the FilterConfig object passed in as a parameter and keep this as an instance variable on the filter for later use.
- The FilterConfig object has a `getServletContext()` method to return the current servlet context , a `getFilterName()` method to get the filter name as declared in the `<filter-name>` element in the deployment descriptor, and `getInitParameter(String paramName)` and `getInitParameter`

Names() methods to return initialization parameters set up as <init-param> elements within the <filter> element.

- Filters are declared in the deployment descriptor using <filter> elements. The mandatory embedded elements are <filter-name> and <filter-class>.
- Filters are accessed by the web container dependent on matches to URL patterns or servlet names, set up in <filter-mapping> elements of the deployment descriptor.
- If a request is made for a servlet, the web container will first match this request to each <filter-mapping> with a matching <url-pattern>. Each filter will be processed in the order that each matching <filter-mapping> appears in the deployment descriptor.
- After processing <url-pattern> matches, the web container will match a servlet (or JSP) request against <filter-mapping> elements with a corresponding <servlet-name>. Again, the web container observes the order of matching <filter-mapping> elements in the deployment descriptor.
- Several filters can execute before a request reaches a resource—this sequence of filters (and the web resource requested) is known as the filter chain.
- The FilterChain object, passed as parameter to the filter's doFilter() method, represents the next thing in the chain—either another filter or the web resource ultimately requested. Thus, the filter's doFilter() method accepts three parameters—the request, the response, and the FilterChain object.
- The FilterChain object passed to the doFilter() method *also* has a doFilter() method. This has only two parameters: the request and response. You call this method—chain.doFilter()—if you want to keep the request passing through the chain. The chain is broken if you don't make this call (so this can be a mechanism for denying access to a resource, for whatever reason).
- Whenever you need to use a Filter to change the request or the response (which you typically do, though it's not inevitable), you are encouraged to "wrapper" the request or response in a suitable wrapper object.
- Suitable wrapper objects extend appropriate wrapper classes in javax.servlet—ServletRequestWrapper or ServletResponseWrapper, or corresponding wrapper classes in java.servlet.http—HttpServletRequestWrapper or HttpServletResponseWrapper.

- You pass the real request or response to the constructor of the wrapper class, hence the wrappering effect (an example of the decorator design pattern).
- The wrapper class may override methods in the request or response and add specialized methods of its own—to transform output to XML, for example.
- The `<dispatcher>` subelement of `<filter-mapping>` can be used to allow filters to trigger on certain routes into the filter: via a client request, through a request dispatcher's forward or include, or as the result of a web container directing to an error page. The valid values are REQUEST, FORWARD, INCLUDE, and ERROR, respectively.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. The number of correct choices to make is stated in the question, as in the real SCWCD exam.

ServletContext

1. What is the result of loading the web-app with the following deployment descriptor and attempting to execute the following servlet? (Choose two.)

```
<web-app>
  <context-param>
    <paramname>author</paramname>
    <paramvalue>Elmore Leonard</paramvalue>
  </context-param>
</web-app>


public class ContextInitParms extends HttpServlet {
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.write("<HTML><HEAD></HEAD><BODY>");
    ServletContext sc = getServletContext();
    out.write(sc.getInitParameter("auther"));
    out.close();
  }
}
```

   A. ParameterNotFoundException is thrown.
   B. Some other exception is thrown.
   C. "Elmore Leonard" is output on the web page.
   D. An application failure occurs.
   E. **null** is output on the web page.
   F. A 404 error occurs in the browser.
   G. Some other error (status code in the 500s) occurs in the browser.

2. What results from a call to the `getInitParameterNames()` method on ServletContext when there are no context parameters set up in the deployment descriptor? (Choose two.)
   A. A NoParametersExistException is thrown.
   B. An empty Enumeration object is returned.
   C. **null** is returned.
   D. An ArrayList object of size zero is returned.
   E. No exceptions are thrown.
   F. An empty Iterator object is returned.
3. Identify true statements about context parameters from the list below. (Choose one.)
   A. The deployment descriptor elements used to describe context parameter names and values are unique to the context parameter element.
   B. Context parameters must be declared in the deployment descriptor before servlets.
   C. Context parameters are available to all web applications loaded by an application server.
   D. In distributable applications, context parameters are duplicated between JVMs.
   E. None of the above.
4. Given a servlet containing the following code, what is the outcome of attempting to compile and run the servlet? (Choose one.)

```
ServletContext context = getServletContext();
String s = context.getAttribute("javax.servlet.context.tempdir");
```

   A. The servlet won't compile.
   B. The servlet won't run.
   C. String *s* has a **null** value.
   D. String *s* has a valid directory as its value.

Attributes, Scope, and Multithreading

5. What is the likely result from attempting to compile and execute the following servlet code? (Choose one.)

```
HttpSession session = getSession();
String s = session.getAttribute("javax.servlet.session.tempdir");
```

A. Won't compile for one reason.
B. Won't compile for more than one reason.
C. Runtime exception when attempting to get access to the attribute.
D. *s* contains **null**.
E. *s* contains a valid String, denoting a temporary directory.

6. Identify true statements from the list below. (Choose two.)
   A. Attribute methods don't throw exceptions.
   B. You cannot remove request parameters.
   C. Attributes can be set by the web container or by application code.
   D. Attribute values are String objects.
   E. "malhereusement" is an illegal name for an attribute.

7. (drag-and-drop question) In the servlet code shown in the following illustration, fill in all the concealed (lettered) parts of the source code with (numbered) choices from the right-hand side, such that the output when the servlet is run is:

```
nulltwothree
```

```
protected void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  PrintWriter out = response.getWriter();
  response.setContentType("text/plain");
  [ A ] session = request.getSession();
  session.setAttribute("one", "one");
  ServletContext context = getServletContext();
  context.setAttribute("two", "two");
  [ B ].setAttribute("three", "three");
  [ C ].[ D ]Attribute("one", [ E ]);
  out.print(session.getAttribute("one"));
  out.print(context.getAttribute("two"));
  out.print(request.getAttribute("three"));
}
```

1	Session
2	session
3	HttpSession
4	set
5	remove
6	get
7	delete
8	null
9	""
10	request
11	context

8. What is result of attempting to run the following code? (Choose one.)

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
  request.setAttribute("a", "request");
  System.out.print(request.getAttribute("a"));
  request.setAttribute("a", "2nd request");
  System.out.print(",");
  System.out.print(request.getAttribute("a"));
  request.removeAttribute("a");
  request.removeAttribute("a");
  System.out.print(",");
  Object o = request.getAttribute("a");
  System.out.print(o);
}
```

   A. "request, request, 2nd request, null" written to standard output
   B. NullPointerException at line 22
   C. AttributeAlreadyRemovedException at line 22
   D. NullPointerException at line 24
   E. "request, 2nd request, null" written to standard output
   F. "request, 2nd request" written to standard output
   G. "request, request, 2nd request" written to standard output

9. From the following list, what is a probable outcome from a call to the `ServletContext.getAttributeNames()` method? (Choose one.)

   A. A **null** reference is returned.
   B. An empty Enumeration is returned.
   C. A nonempty Enumeration is returned.
   D. An empty ArrayList is returned.
   E. A nonempty ArrayList is returned.
   F. A NoAttributesFoundException is thrown.
   G. Some other exception is thrown.

10. Identify true statements about scope from the following list. (Choose two.)

   A. Context scope can span JVMs.
   B. Session scope can span JVMs.

C. Requests can span web apps.
D. Sessions can span web apps.
E. Requests can span JVMs.

Dispatching

11. What is the outcome of executing ServletA? You can assume that (1) ServletB has a mapping of "/ServletB" and a name of "ServletB," and (2) imports have been omitted from the code for brevity; the code will compile successfully. (Choose one.)

```
public class ServletA extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
       RequestDispatcher rd = getServletContext().getNamedDispatcher(
               "ServletB");
       rd.forward(req, resp);
    }
}

public class ServletB {
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
           throws ServletException, IOException {
        String attr = (String)
req.getAttribute("javax.servlet.forward.servlet_path");
        PrintWriter out = resp.getWriter();
        out.write("Attribute value: " + attr);
    }
}
```

A. NullPointerException thrown
B. "Attribute value: null" output to the web page
C. A blank web page
D. ServletNotFoundException thrown
E. "Attribute value: /ServletB" output to the web page
F. "Attribute value: ServletB" output to the web page
G. ClassCastException thrown

12. Identify which of the following are names of special attributes associated with the dispatching mechanism. (Choose two.)

    A. java.servlet.include.servlet_name

    B. javax.http.servlet.include.query_name

    C. javax.servlet.include.servlet_path

    D. javax.servlet.forward.request_url

    E. javax.servlet.include.path_info

    F. java.servlet.forward.context_path

13. What are possible outcomes from executing the `doGet` method in ServletC below? (Choose two.)

```
public class ServletC extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        RequestDispatcher rd = getServletContext().getRequestDispatcher(
                "ServletB");
        rd.forward(req, resp);
    }
}
```

    A. HTTP 500 error (error in 500s).

    B. NullPointerException.

    C. HTTP 404 error.

    D. Some other exception.

    E. ServletNotFoundException.

    F. ServletB runs.

    G. A file called ServletB is served from the context directory.

14. What is the web page output from executing ServletD with the URL below? (Choose one.)

```
http://localhost:8080/myapp/ServletD?fruit=orange
public class ServletD extends HttpServlet {
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
           throws ServletException, IOException {
```

```
        RequestDispatcher rd = getServletContext().getRequestDispatcher(
                "/ServletE?fruit=pear");
        rd.forward(req, resp);
    }
}

public class ServletE extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        String[] valueArray = request.getParameterValues("fruit");
        for (int i = 0; i < valueArray.length; i++) {
            if (i > 0) {
                out.write(", ");
            }
            out.write(valueArray[i]);
         }
        String queryString = (String)
request.getAttribute("javax.servlet.forward.query_string");
        int pos = queryString.indexOf("=") + 1;
        String values = queryString.substring(pos);
        out.write(", " + values);
    }
}
```

A. pear, pear

B. pear, orange, orange

C. orange, pear, orange

D. orange, pear, pear

E. orange, pear

F. pear, orange, **null**

G. orange, pear, **null**

H. pear, orange, pear, orange

15. ServletA forwards to ServletB, which includes Servlet C, which forwards to ServletD, which includes ServletE. When ServletA is requested, which servlets might contribute to the final response? (Choose one.)

A. ServletD and ServletE
B. ServletB, ServletC, ServletD, and ServletE
C. ServletD only
E. ServletB only
F. All of them

Filters

16. Identify true statements about filters. (Choose one.)

A. You cannot work directly with the request object that is passed as a parameter to the filter.
B. The order of filter processing is arbitrarily determined by the web container.
C. Only URL patterns can be used by filters to target specific web resources.
D. You must implement the `doChain(request, response)` method to pass control from filter to filter, or filter to servlet.

17. Which of the following is a legal filter mapping declaration in the deployment descriptor? (Choose one.)

A.

```
<filter-mapping>
  <filter-name>MicroPaymentFilter</filter-name>
  <servlet-name>MicroPaymentServlet</servlet-name>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

B.

```
<filter>
  <filter-name>MicroPaymentFilter</filter-name>
  <filter-class>webcert.ch03.MicroPaymentFilter</filter-class>
  <filter-mapping>
    <url-pattern>/MicroPaymentServlet</url-pattern>
  </filter-mapping>
</filter>
```

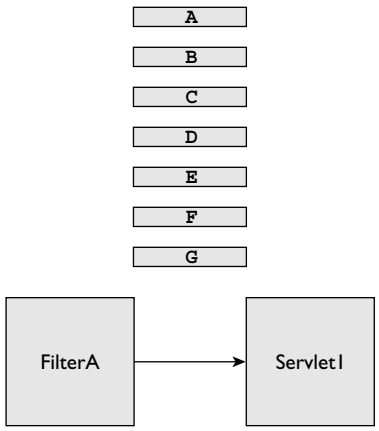
C.

```
<filter-mapping>
  <filter-name>MicroPaymentFilter</filter-name>
  <url-pattern>MicroPayment/*</url-pattern>
</filter-mapping>
```

D.

```
<filter>
  <filter-name>MicroPaymentFilter</filter-name>
  <filter-class>webcert.ch03.MicroPaymentFilter</filter-class>
  <filter-mapping>
    <servlet-name>MicroPaymentServlet</servlet-name>
  </filter-mapping>
</filter>
```

18. (drag-and-drop question) In the following illustration, FilterA chains to Servlet1, which extends HttpServlet. Neither of these components is loaded on startup. Imagining that this is the first invocation for each of these components, match the numbered method calls to the lettered sequence if Servlet1 is requested.



1	`init()`
2	`init(FilterConfig fc)`
3	`init(ServletConfig sc)`
4	`init(ServletContext sc)`
5	`init(FilterContext fc)`
6	`doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`
7	`doChain(ServletRequest request, ServletResponse response, Filter filter)`
8	`doPost(ServletRequest request, ServletResponse response)`
9	`doGet(HttpServletRequest request, HttpServletResponse response)`
10	`service(HttpServletRequest request, HttpServletResponse response)`
11	`service(ServletRequest request, ServletResponse response)`

19. From the available options, what is the likely outcome from running the code below? (Choose one.)

```
protected void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
  RequestDispatcher dispatcher =
    getServletContext().getNamedDispatcher("/ServletB");
  dispatcher.forward(request, response);
}
```

A. DispatcherNotFoundException.

B. Runtime error because of incorrectly formed parameter to `getNamedDispatcher()` method.

C. NullPointerException.

D. ServletB can obtain request attribute javax.servlet.forward.request_uri.

20. Given the following deployment descriptor, identify the sequence of filters that execute on a direct client request for ServletA. (Choose one.)

```
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <servlet-name>ServletA</servlet-name>
</filter-mapping>
<filter-mapping>
  <filter-name>AuditFilter</filter-name>
  <url-pattern>/ServletA</url-pattern>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
<filter-mapping>
  <filter-name>EncryptionFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<servlet-mapping>
  <servlet-name>ServletA</servlet-name>
  <url-pattern>/ServletA</url-pattern>
</servlet-mapping>
```

A. LogFilter, AuditFilter, EncryptionFilter

B. LogFilter, EncryptionFilter

C. LogFilter

D. EncryptionFilter, AuditFilter, LogFilter

E. EncryptionFilter, LogFilter
F. AuditFilter, EncryptionFilter, LogFilter

LAB QUESTION

We'll use this lab as an opportunity to put together several of the concepts you have encountered in this chapter in order to write a censorship filter. If there are any words in a web page you disapprove of (such as "massive executive pay bonus"), your filter will throw a fit of pique, suppress the response entirely, and substitute an alternative message.

Have an AttributeSetter servlet, which will pick up a properties file name and location from servlet context initialization parameters. AttributeSetter should load the file as a properties object and write each named property as a request attribute. This mainly uses techniques you covered in the first two exercises in this chapter. Then have AttributeSetter forward to another servlet, Attribute Displayer—which simply displays all the request attributes it knows about.

Then write a filter—CensorshipFilter—which triggers on access to the AttributeSetter servlet. The filter should scan the (wrappered) response for any words it doesn't like. If any are encountered, the filter should clear the response completely and write its own response instead. Turn the filter on and off using a filter initialization parameter.

As a more subtle form of censorship, have the CensorshipFilter scan all the request attributes and remove any containing words it doesn't like—before passing control down the chain to Attribute Displayer. Then AttributeDisplayer will display as normal, but of course, the offensive attributes have already been purged.

SELF TEST ANSWERS

ServletContext

1. ☑ **D** and **F**. **D** is correct because the application will not load. The deployment descriptor is incorrectly formed: The element names should be `<param-name>` and `<param-value>`, with hyphens. **F** is also correct. The question says that you attempt to execute the servlet. In the event of the web application simply not being available, a "page not found" (404) error results.
   ☒ **A** is incorrect because you never get any kind of exception (including the made-up ParameterNotFoundException) from the `getInitParameter` method—even though the requested parameter name ("auther") doesn't match what is set up in the deployment descriptor. **B** is incorrect because the servlet code is perfectly fine in all respects and in any case never executes! Because the servlet is never loaded and never executes, **C** and **E** can be discounted (though had the deployment descriptor been correctly formed, **E** would have described the output correctly: null). Finally, **G** is incorrect because an error in the 500s results only when the target resource is actually found, but runs incorrectly.

2. ☑ **B** and **E**. **B** is correct because you do get an Enumeration object returned that has no elements. No exceptions are thrown just because there are no context parameters, so **E** is correct as well.
   ☒ **A** is incorrect because you never get any kind of exception (including the made-up NoParametersExistException) from the `getInitParameterNames` method. **C** is incorrect because you don't get a null object reference; the Enumeration returned has a valid reference, just no elements. **D** and **F** are incorrect because it is an Enumeration that's returned, not an ArrayList or Iterator (or Vector or any other sort of thing from the myriad Collection classes Java has available).

3. ☑ **E** is the correct answer: There are no true statements in the list!
   ☒ **A** is incorrect because `<param-name>` and `<param-value>` are used for servlets' initialization parameters as well as ServletContext initialization parameters. **B** was correct in previous versions of the exam and servlet specification. However, the XSD for servlet specification 2.4 gives you latitude to place context parameters wherever you like in the deployment descriptor (provided, or course, that each `<context-param>` element is bedded directly under the root element `<web-app>`). I have to say that I prefer to respect the old order when setting up the deployment descriptor—for backward compatibility if nothing else. But for exam answer purposes, you should identify answer **B** as false. **C** is incorrect: Context parameters are available only

to the web application to which they belong, no others. **D** is incorrect because an important limitation of servlet context information is that there is no mechanism to duplicate parameter information from one JVM to another in distributed apps. (Of course, chances are you have identical deployment descriptors with the same parameters declared in other JVMs supporting the distributed application—so, effectively, the data are available wherever the application runs. However, that doesn't make answer **D** any truer!)

4. ☑ **A** is the correct answer. The compilation fails with a ClassCastException. The output of the `ServletContext.getAttribute()` method is an object. Since the value of the standard attribute named javax.servlet.context.tempdir is a String, the output is safe to cast to a String.
☒ **B** is incorrect because the servlet never gets as far as running, which of course also discounts **C** and **D**. If the ClassCastException were corrected, then **D** should be the correct answer, for this standard attribute should always have a valid value set by the web container.

Attributes, Scope, and Multithreading

5. ☑ **B** is the correct answer. Although this looks like a question about attributes, it is also about session API knowledge. You retrieve a session from a request, not from the servlet itself, so that's one error. Furthermore, whatever scope you use the `getAttribute()` method in (in this case, session scope), you have to cast the object retrieved back to the type of variable you are using in the assignment (in this case, String). So there are two compilation errors.
☒ **A** is incorrect because there are two compilation errors. **C** is incorrect because the code never gets to run. If the compilation error were fixed, then **D** is likely to be correct: *s* would be null. The web context shouldn't have set up an attribute of this name, nor should your code (as javax.<anything> is reserved for web container attributes). **E** is incorrect, a deliberate attempt to confuse you with the context attribute javax.servlet.context.tempdir.

6. ☑ **B** and **C** are the correct answers. You can't remove request parameters: There are no methods to do this (don't confuse this with the fact that you can remove attributes). And attributes can be set up in two places: by the web container or in your code.
☒ A is incorrect because although most attribute methods don't throw exceptions, session attribute methods can throw an IllegalStateException. **D** is incorrect because you can hold any kind of object as an attribute value, not just Strings. E is incorrect: Although your attribute names should begin with a reverse domain name (e.g., com.myco.malhereusement), they don't have to do so—it's only a suggestion in the servlet specification, not a requirement.

7. ☑ **A** maps to **3** (you retrieve an HttpSession type, not Session), **B** maps to **10** (must be the `request` parameter), **C** maps to **2** (must be the `session` parameter), **D** maps to **4** (must be

setAttribute, for there are two parameters; removeAttribute takes only one), and **E** maps to 8 (**null** literal—so that session.getAttribute("one") will fail to find an attribute and thus return **null**).

☒ There are no other correct combinations.

8. ☑ **E** is the correct answer. The first value of the attribute is printed out, then the changed second value, then **null**, for the attribute has been removed.

☒ **A** is incorrect: The answer tries to persuade you that values added to attributes accumulate (a bit like parameters) instead of being totally replaced. **B** and **C** are incorrect—there's no reason for a NullPointerException, and there's no such thing as an AttributeAlreadyRemoved Exception. It doesn't matter how many times you remove the same-named attribute; the code doesn't blow up. **D** is incorrect—you don't get a NullPointerException from passing a null object reference into System.out.print. **F** is incorrect—you might think it was correct if the System.out.print at line 24 did go wrong. **G** is yet another red herring that plays on some of the wrong assumptions already described.

9. ☑ **C** is the correct answer: It's the only probable outcome. There should be at least one context attribute set by the servlet container; hence, the Enumeration is unlikely ever to be empty.

☒ **A** is incorrect; you will always get a valid reference. **B** is remotely possible, but not probable (it could occur because your code removed all context attributes, including ones set up by the web container). **D** and **E** are incorrect—you get old-fashioned Enumerations from this method, not any newer collection class such as ArrayList. Finally, the method shouldn't throw any exceptions, so **F** and **G** are incorrect.

10. ☑ **B** and **C** are correct answers. Session scope can span JVMs in a distributable application. Requests can span web applications when a request dispatcher is used from another context.

☒ **A** is incorrect; there is one context per web application per JVM. **D** is incorrect; threads dispatching across web applications find themselves dealing with separate session objects in each web application. **E** is incorrect; there is no mechanism to carry requests across JVMs, even in distributable applications.

Dispatching

11. ☑ **B** is the correct answer. The code executes correctly. However, because the method used to obtain a RequestDispatcher in ServletA is `getNamedDispatcher()`, the attribute javax.servlet.forward.servlet_path is not set up in the servlet that is the target of the forward, ServletB.

☒ **A**, **D**, and **G** are incorrect, for the code runs perfectly well. In ServletA, the line `rd.forward()` has the potential to throw a NullPointerException—but not when a valid servlet is found. The `getAttribute()` output cast to a String in ServletB is quite correct, hence no

ClassCastException. ServletNotFoundException does not exist. **C** is incorrect because there is output on the web page. **E** and **F** are incorrect—**E** would have been a correct version of the servlet path had the dispatcher used arisen from a `getRequestDispatcher()` method.

12. ☑ **C** and **E** are the correct answers.
☒ **A**, **B**, and **F** are incorrect: You can eliminate them immediately, for all the special attributes begin javax.servlet, which is then followed by .forward or .include. **D** is almost right—but the attribute name should end request_uri, not request_url.

13. ☑ **A** and **D** are the correct answers. An IllegalArgumentException occurs because the `getRequestDispatcher` method on ServletContext cannot accept a path that begins from somewhere other than the context root—in other words, the path parameter must begin with a forward slash. As a consequence, a server side error (error in 500s) will be returned to the client.
☒ **B** is incorrect. There won't be a NullPointerException from the `rd.forward()` line because it will never be reached. **C** is incorrect because there won't be a search for a file that cannot be found. **E**—ServletFoundNotException—is as made up now as it was in a previous bogus answer. **F** and **G** are incorrect, but would both be possible outcomes if the `getRequestDispatcher()` call were legal.

14. ☑ **B** is the correct answer. The parameter named "fruit" is passed as part of the query string to ServletD, with a value of "orange." When the request path is set for ServletE in the call to `getRequestDispatcher`, the query string contains the same-named parameter with a value of "pear." This doesn't overwrite the original parameter value. You can have multiple parameter values of the same name. Instead, it inserts the "pear" value ahead of the "orange" value, but both are valid parameter values for "fruit." So when the ServletE code prints out the parameter value for "fruit" obtained with request.getParameterValues("fruit"), it outputs "pear, orange" in that order. Then the query string is obtained from javax.servlet.forward.query_string. This contains the query string as it was in the forwarding servlet, ServletD, so fruit=orange. After some judicious string manipulation, the value "orange" is extracted from the query string and added to the response output, so "pear, orange, orange" is the final result.
☒ **A**, **C**, **D**, **E**, **F**, **G**, and **H** are all incorrect because of the reasoning above.

15. ☑ **A** is the correct answer. The last servlet in the dispatching sequence that is forwarded to is ServletD, so anything that previous servlets did to the response is ignored. ServletD includes ServletE, so both might contribute to the response.
☒ **B** is incorrect because the forward to ServletD obliterates the contribution of ServletB and ServletC, which also excludes answer **D**. **C** is incorrect, for ServletD includes ServletE, so ServletE's work on the response should be taken into account. **E** is incorrect because of the reasoning in the correct answer.

Filters

16. ☑ **A** is the correct answer (the only true statement). The request object passed as parameter to the filter must be wrappered in a ServletRequestWrapper or HttpServletRequestWrapper object.
☒ **B** is incorrect because the order of filters is determined by their placement in the deployment descriptor. **C** is incorrect because filters can target servlets by name as well as by URL pattern. **D** is incorrect: The thing you are implementing is a chain of filters, but the method used to pass control along the chain is called `doFilter(request, response)`, not `doChain()`.

17. ☑ **A** is again the correct answer. A filter mapping can be legally expressed with a filter name and a servlet name. Although the dispatcher element with a value of REQUEST is what you get by default when no dispatcher element is specified, there's nothing wrong with explicitly including the element like this.
☒ **B** is very incorrect; you don't include `<filter-mappings>` within `<filter>` elements. They are separate elements nested in the root element `<web-app>`. This makes answer **D** incorrect as well. **C** is incorrect, not because of incorrectly stacked elements, but because of an illegal value for the URL pattern—which should begin with a forward slash ("/").

18. ☑ **A** maps to **2** (init with FlterConfig parameter), **B** maps to **6** (doFilter method), **C** maps to **3** (init with ServletConfig parameter), **D** maps to **1** (init with no parameters), **E** maps to **11** (service method passing ServletRequest and ServletResponse), **F** maps to **10** (protected service method passing HttpServletRequest and HttpServletResponse), and **G** maps to **9** (`doGet` method). This is as much a question about servlet life cycle as filter life cycle—mean, but you do get questions that cross over different objectives from time to time.
☒ This is the only sequence that can be guaranteed to occur.

19. ☑ **C** is the correct answer from the available options. If the `getNamedDispatcher()` method fails to find the path to ServletB, the dispatcher reference will be null, so a Null PointerException will result on executing the `forward()` method.
☒ **A** is incorrect—DispatcherNotFoundException is made up (dispatcher methods that fail to find a dispatcher simply return null). **B** is incorrect because the parameter to `getNamed Dispatcher()` is legal. The name used begins with a forward slash, so looks more like a servlet name than a servlet mapping. However, while it is inadvisable to have a servlet name in this form, it does work. **D** is incorrect. Although the special attribute is correctly named, it is not available when the forward is on a named dispatcher.

20. ☑ **E** is the correct answer. First, the processing works through the filter-mappings with a matching URL pattern. EncryptionFilter runs because the URL pattern of "/*" matches any request. Then processing works through the filter mappings with matching servlet names.

LogFilter has a matching name, so it executes. Note that filters mapped by URL pattern are executed before filters mapped by servlet name.

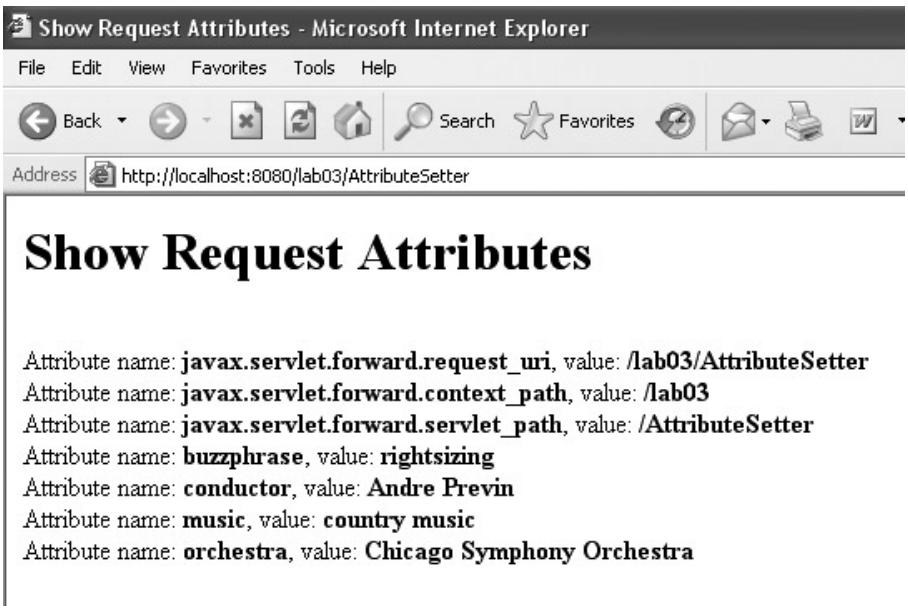
☒ **A**, **B**, **C**, **D**, and **F** are incorrect because of the reasoning in the correct answer above. Note that AuditFilter doesn't execute through a direct client request. AuditFilter will only be invoked as the result of calling ServletA via the `forward()` method on a RequestDispatcher object.

LAB ANSWER

Deploy the WAR file from the CD called lab03.war, in the /sourcecode/chapter03 directory. This contains a sample solution. If you look in the deployment descriptor web.xml, you'll find a definition for a filter named CensorshipFilter. This has an initialization parameter named *censorship*. As delivered, this has its value set to "off." When you run the application using the URL

```
http://localhost:8080/lab03/AttributeSetter
```

You should see a list of attributes like this:



Some attributes arise from container-provided "forwarding" attributes (because AttributeSetter executes a `forward()` method on a request dispatcher object).

It also displays some attributes that originate from the properties file lab03.properties. If you change the *censorship* initialization parameter in web.xml to have a value of "on," restart your server, and call AttributeSetter with the same URL, you should see a screen like this:

This message comes about because the filter class (CensorshipFilter) has logic that discovers "banned words" in the attributes that subsequent servlets would otherwise display, so it suppresses the call to those servlets. If you look in the source of CensorshipWrapper.java, you'll find out what the banned words are. You can change the text of the lab03.properties file (in the /WEB-INF directory) to avoid the banned words—see then if a refresh of your browser will display the properties from the file.