# 4

# Sessions and Listeners

## CERTIFICATION OBJECTIVES

- Session Life Cycle
- Session Management
- Request and Context Listeners
- Session Listeners

✓ Two-Minute Drill

Q&A Self Test

If you need a way to associate a series of related requests—and let's face it, most web applications do—then you need a session. Sessions are a central plank in the web container framework. We began to explore them in Chapter 3, but they have enough facets to justify this chapter (almost) to themselves.

We have already learned that sessions provide one of the three principal "scopes" of the web container model, the other two being request and context. We have also seen that (like requests and contexts) an object represents each session and that you can attach attributes to this object representing information of any sort. What we will do in this chapter is to more fully explore the boundaries of session scope and find out exactly what causes the beginning and end of a session. We'll find out when a session is regarded as "new," and what happens to make it lose that newness.

We'll also see what mechanisms a web container might employ to maintain its sessions. The mechanism of choice uses a cookie, a small file traded between the web container and the client browser and containing a unique identifier for the session. A poor man's substitute (in environments where cookies are disallowed or unsupported) is "URL rewriting"—having the session ID embedded in the URL—and we'll explore that also.

Then we'll cover "listeners," an aspect of the web container model deferred from Chapter 3. These are classes with methods called by the web container under particular circumstances—when an attribute is changed, for example, or when a context comes into being. We'll depart from the session focus for a section to find out how listeners pertain to the request and context life cycles. But we then return to session with a vengeance, which boasts the most listeners to support its more complex life cycle. And that will round off the exploration both of sessions and the web container model.

## CERTIFICATION OBJECTIVE

# Session Life Cycle (Exam Objective 4.2)

*Given a scenario, describe the APIs used to access the session object, explain when the session object was created, and describe the mechanisms used to destroy the session object, and when it was destroyed.*

In this section, you'll learn what you need to know about session basics for the exam. You'll see how the J2EE spec gets around the fundamental problem of HTTP communication: a series of unconnected requests and responses between client and server, often described as "fire and forget." What if you're the client and you don't want to be forgotten? Or if you're the server, and want to know when the client last got in touch with you? What if the client has finished with you the server: How do you know, and once you do know, how can you gracefully let go of that client knowledge?

## The Life History of a Session

Of course, a session isn't just about continuity across a series of requests—though we'll have plenty to say about that. There are good application reasons for wanting sessions—usually to store information to ensure the smooth running of your application. You will often have a transaction running over several requests. Perhaps I set up a new order header on one page but don't want to commit the results until I've set up order lines on a separate page. My application could hold the order header information in attributes attaching to a session. Only after hitting the button on the order line setup page would header and line information be taken out of the session's memory and planted permanently in the database.

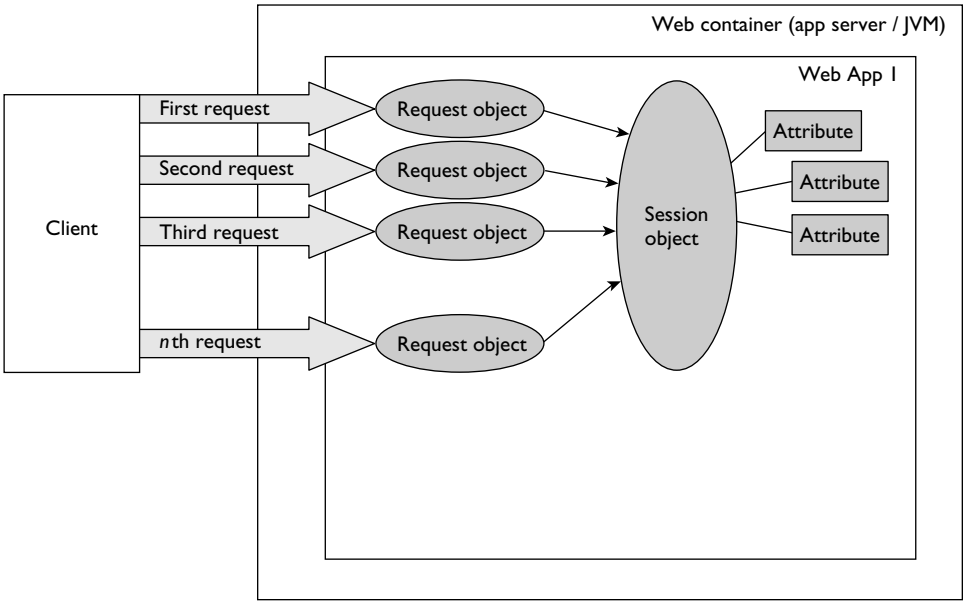
And this is mostly what session APIs are about. Besides that, it's about control of a session—dictating when it will end. You have mechanisms to time out a session if no requests are received for a certain time. Or you can use session APIs to apply more stringent timing rules. You might want to ensure that the session will not exceed a certain time limit, regardless of the requests it receives—perhaps in a game application. And you'll see in this chapter that your scope is wider than that: You can invalidate a session for any reason you see fit.

But first we'll explore a fundamental question: How can you get hold of a session in the first place?

### Getting Hold of a Session

When a client makes its very first request to a particular web application, neither party knows about each other. How could they? It's as if the client has just rung the doorbell. At this point, there's no session. And there doesn't need to be a session. If the person on the doorstep is merely asking you to sign for a parcel delivery, then it's a one-off request. If he wants you to participate in a survey about your buying habits (and you agree), that requires a series of interactions—in other words, a session.

So a session exists only if your servlet decides it needs one. The servlet gets hold of that session from the HttpServletRequest object. This has a `getSession()` method, which returns an HttpSession object. And the idea is that a series of calls to `HttpServletRequest.getSession()` from a related set of requests (ones emanating from the same client) will always retrieve the same session object, as shown in the following illustration.




As far as the J2EE servlet API is concerned, sessions and session scope belong to the world of HTTP. HttpSession is an interface that lives in the javax.servlet.http package. You can get HttpSession objects only from HttpServletRequest, not from ServletRequest. There is no equivalent in the non-HTTP servlet world—for example, a Session interface provided in the javax.servlet package. Of course, there's nothing stopping you from providing your own infrastructure, but outside of HTTP, you're on your own.

Actually, `HttpServletRequest.getSession()` is an overloaded method. It exists in the no-argument form, or can accept a single **boolean** parameter: `HttpServletRequest.getSession(boolean create)`. First you should know that

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

is a shorthand form of

```
HttpSession session = HttpServletRequest.getSession(true);
```

These two calls are functionally equivalent. These calls will return an HttpSession come what may, returning the existing object if it already exists or *creating a session object if one does not exist already*.

Alternatively, you might call

```
HttpSession session = HttpServletRequest.getSession(false);
```

This call will return a session object, but *only* if one already exists. Why might you want to do this? Perhaps your application is designed so that initial requests should pass through a "Login" servlet, which establishes a session. As a security measure, all other servlets in the application make a `getSession(false)` call when they need the session object. Only if the user has legitimately passed through the "Login" servlet will the other servlets get the session object they need for the application to function.

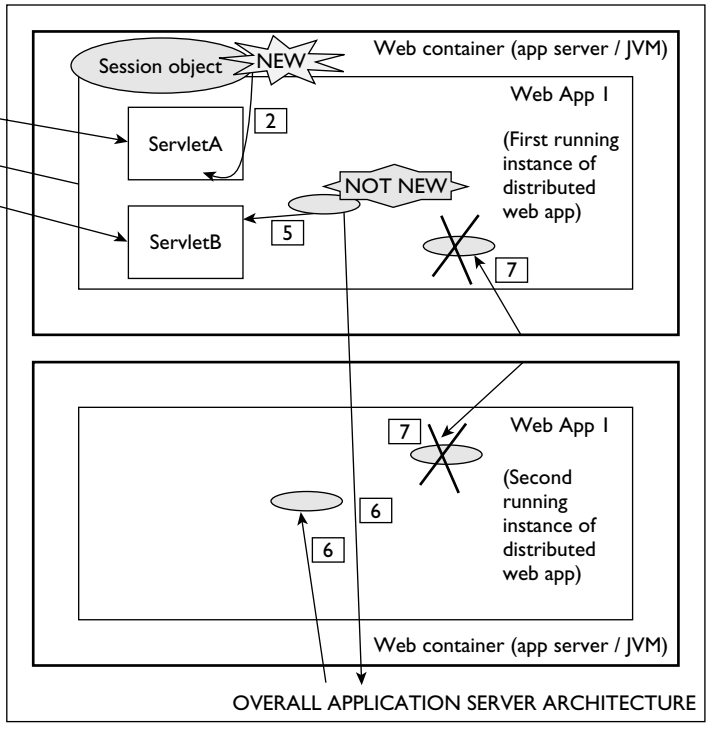
## Session Scope Revisited

Let's now revisit session scope, which we briefly explored in Chapter 2. Figure 4-1 shows a possible session "lifetime." At (1) in Figure 4-1, a client (web browser window) makes its first request to your web application—as it happens, for ServletA. ServletA obtains a session object through the request, supplied by the web container ((2) in Figure 4-1). At this point, the session is deemed to be "new." The client doesn't yet know about the session's existence—after all, it has only just come into being. You can test the session state as follows:

```
HttpSession session = request.getSession();
if (session.isNew()) {
  // Do something conditional on session newness
}
```

At (2) in Figure 4-1, the boolean value returned by `HttpSession.isNew()` will be true. So how does a session lose its newness? Along with the response to ServletA, the web container (at (3) in Figure 4-1) returns some sort of tracking information that uniquely identifies the session that has just been created.

**FIGURE 4-1** Session Scope




1. Client makes its first request to web container (for ServletA).
2. ServletA requests session object: web container provides new session object.
3. Web container passes back session tracking info to client.
4. Client returns acknowledgment of tracking info on next request to ServletB: this flags session as no longer new.
5. ServletB accesses (no longer new) session object.
6. Overall application server architecture replicates session object from one app server to another (separate JVM).
7. Client doesn't access session for specified time interval: web container removes session objects.

on the Job

***What constitutes a "new browser session"? Here's an observation on Internet Explorer's behavior. If you launch Internet Explorer afresh, then access a session-aware servlet—that's a new session. If Internet Explorer itself launches a new Internet Explorer window (e.g., by running File | New Window or by running some appropriate script) and that new window accesses a session-aware servlet—it shares the session object with the Internet Explorer window from which it was launched. This makes reasonable sense, and it's probably not the only browser that exhibits this behavior. I point it out for two reasons. First, for the sake of your application logic, you will need to know what circumstances cause the client software on a particular PC to maintain or drop the session. Second, this behavior makes session attributes officially not thread safe. It is imaginable that your user might toggle between two Internet Explorer windows, one spawned from the other, and set off a long-running request from each. Both these request threads could access the same session object.***

**exam Watch**

***You will often encounter questions where you need to know the two circumstances whereby sessions remain "new." The first is when the client doesn't yet know about the session because this is a first request to a web application. The second is because a client declines to join the session, which it typically does by refusing to return the session tracking information. Under these circumstances, the web application treats each later request from the client as if it were the first, providing a new session each time.***

The client now makes another request (at (4) in Figure 4-1), this time to ServletB (though for the purposes of this account, it wouldn't matter if the request were made to the same servlet again, ServletA). The client has agreed to join the session, which is typically achieved by passing the tracking information (the session key) back to the web container along with the new request. ServletB (at (5) in Figure 4-1) obtains the session from the request. This is exactly the same session object that ServletA had access to, but now—as the client knows about the session and has agreed to join it—the session is no longer new (as tested with the `isNew()` method).

## Distributed Sessions

That's normally as far as a session gets. However, it is possible—for the sake of load balancing or fail-over or both—to mark a web application as distributable, if it is supported by your application server. All you need do is place the element `<distributable />` (or `<distributable></distributable>`) somewhere underneath parent element `<web-app>` in your deployment descriptor. Note that this step may have no effect whatsoever: This element works *if and only if* your application server supports distributed applications. If it does, the effect should be as shown in Figure 4-1: The same web application running in two (or more) different JVMs. In Figure 4-1, there are two "clones" of the same web application. The lower half of the figure depicts a second running instance of Web App 1.

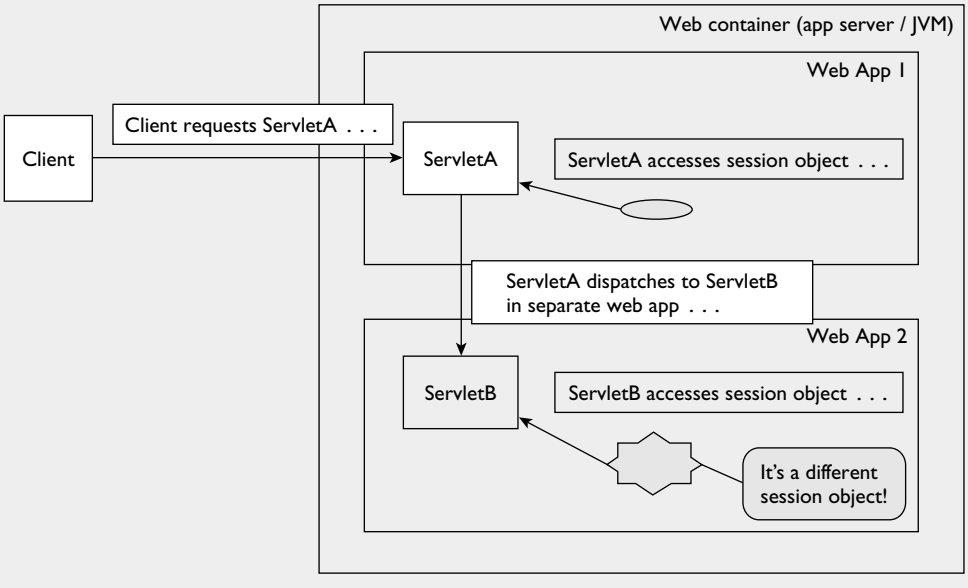
Why would you do this? If the first running instance should fail, your architecture might have a fail-over mechanism in place to divert requests to the second running instance. Would this disrupt a client session if the failure came between two requests from a client? Not if the architecture had migrated your session object from one JVM

to another. And this is what is shown at (6) in Figure 4-1: The session is replicated from the first running instance of Web App 1 to the second running instance of Web App 1. And with the session go all the objects attaching to the session. The only condition is that the attributes you place in a session should implement the Serializable interface.

The exact mechanism by which this is achieved varies from one application server to another and is beyond the scope of the exam. The phrase in Figure 4-1 "overall application server architecture," is deliberately vague. There might be some direct communication in place between the two application servers shown in Figure 4-1. There might be some additional application server process polling each web application clone, and maybe a database that stores session objects. You don't need to care at this point—all that's required is that you understand the implications for sessions when a web application is marked distributable.

**exam Watch**

***A session is available to be shared between web resources in a single web application. Note I say a* single *web application: A session cannot cross web application boundaries. If you use the RequestDispatcher mechanism to get at a servlet in another web application and that servlet accesses the session object, it will be a different session object, as shown in the following illustration.***



## Session Death

So how does a session die? There is no obvious trigger. As we well know, servlets work on a series of requests and responses. The protocol doesn't demand a continuous connection between the client making the request and the server providing the response. So how do you know when the client has made its last request, and take this as a cue to free up the session?

The answer is that you *don't* necessarily know. The HttpSession API provides an `invalidate()` method—so if your application has a "Log Off" button and the user clicks it, a LogOff servlet should summon the session and call the method. But what if this user, despite repeated and prolonged training followed up with heavy threats, just closes the browser window? The session is over because the client has gone. Even if the user reopens the browser and connects back to the same web application, the web container will interpret this as a new session. Yet the server is blissfully unaware that the *original* client (aka browser session) won't be making any more requests.

The solution is to have a time-out mechanism, and that's precisely what is built into the web container model. If a session has not been used for a prescribed amount of time, the web container invalidates the session itself. This is what's illustrated at (7) in Figure 4-1. For the clock to start ticking for a session's time-out, all requests using the session must have come to an end—that is, their servlets must have exited their `service()` method, and any enclosing filters exited their `doFilter()` method.

The time-out value is controlled in one of three ways.

**Application Server Global Default** Most application servers provide their own mechanism for imposing a global default on session length. The specifics of this don't matter for the exam, and they vary by server in any case.

**Web Application Default** You can set up a default value in the deployment descriptor. Here's an example of how to do this:

```
<web-app>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

The value in the `<session-timeout>` element is expressed as whole *minutes*. Any integer value is fine. A value of 0, or a negative value, denotes that the default for sessions created in the web application is to never expire. Note that

`<session-timeout>` is `<session-config>`'s only subelement, and `<session-config>` is a direct child of the root element `<web-app>`.

**Individual Session Setting** On obtaining a session, your servlet code can set the time-out value for that individual session only using the `HttpSession.setMaxInactiveInterval(int seconds)` API. Note that the unit of time is in *seconds*—contrasting with the deployment descriptor `<session-timeout>` element, which contains a value in minutes. Again, a negative value supplied as an argument causes the session to never expire. But by contrast with the deployment descriptor, a value of zero *doesn't* have the same effect. `setMaxInactiveInterval(0)` causes the session to expire immediately—which is rarely desirable!

Let's set the deployment descriptor and API characteristics side by side in a table, for you can pretty much count on seeing a question on this theme:

| | `<session-timeout>` element in deployment descriptor | `HttpSession.setMaxInactiveInterval(int seconds)` API |
|---|---|---|
| Scope: | The default value for all sessions created in the web application. | The value for the session for which the method is called *only*. |
| Unit: | Minutes. | Seconds. |
| Zero value denotes: | Sessions should never expire. | Immediate expiration of session. |
| Negative value denotes: | Sessions should never expire. | Session should never expire. |

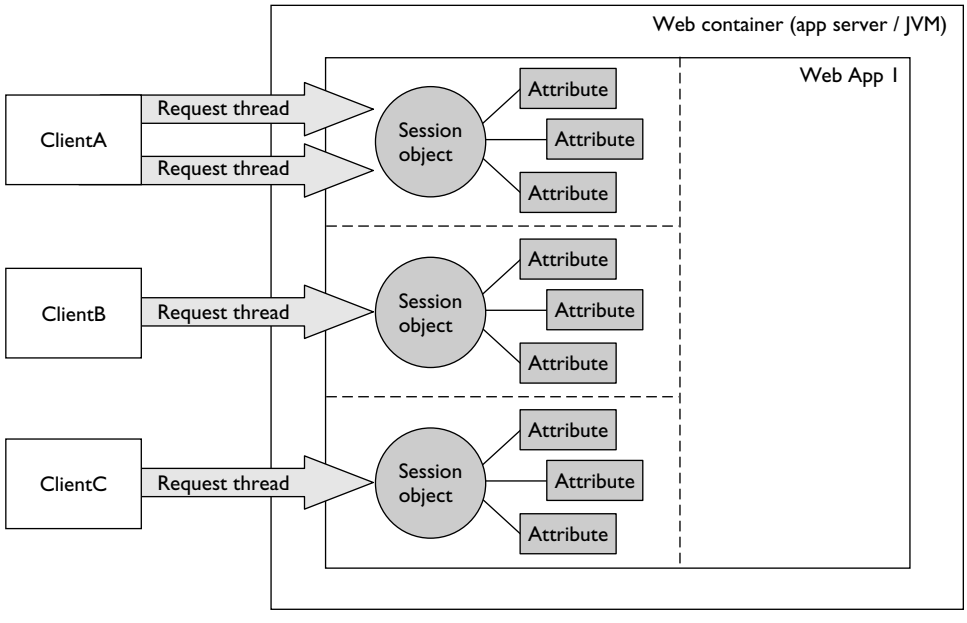
Note that there is a corresponding `HttpSession.getMaxInactiveInterval()` method. This returns an int primitive representing the number of seconds permitted between client requests before the web container invalidates the related session. It doesn't matter which of the three time-out-setting approaches you use for this method to return a value. So—for example—you might use the deployment descriptor and set a 30-minute time-out: `<session-timeout>30</session-timeout>`. The `getMaxInactiveInterval()` method will then return a value of 1800—the equivalent in seconds.

A final word of warning: Don't try to prolong a session's life by artificial means—or, at least, don't hold me responsible for the consequences! How could you contrive such an infringement? Perhaps by attaching a session object as an attribute to your context, then trying to get it back when the session scope has passed. The results are likely to be unpredictable, or even catastrophic to your web application.

That's not to say that an application server might not preserve session objects. Just as for request objects, it is probably more efficient to maintain a pool of session objects instead of going through instantiation and garbage collection whenever a new session is required or jettisoned. The web container just has to provide the appearance of a new session object, stripped of attributes and returning "true" to `isNew()` requests. Whether this is a session object that has been newly constructed or has already been through many previous incarnations is something about which you should not have to know or care. As a web developer, confine yourself to the session APIs you have at your disposal.

## Multithreading and Session Attributes

In most circumstances you can regard session objects and their associated attributes as thread safe. The following illustration shows how this looks for the client request threads to a web application.




However, as far as the servlet specification goes—and therefore the exam as well—you can't rely on session attributes being thread safe. You see in the preceding illustration—for ClientA—two requests, which both access the same session object. Nothing strange about that: After all, that's the point of the session object—to provide continuity between requests. The issue is whether those two requests from the client could ever overlap. It's my experience that in normal web browser usage,

you don't (and mainly, can't) overlap requests from the same main browser window. However, it is a theoretical possibility: See the "On the Job" feature (see page 242) to learn how it might happen. So you should design your web applications such that access to session attributes is synchronized. Or if you don't—perhaps for performance reasons—ensure that multithreaded access to your session attributes will not compromise the well-being of your web application.

## Other Session APIs

We touched on `HttpSession.invalidate()` above. This invalidates the session and then removes any attributes associated with the session. Invalidation works by making (almost) all of the methods on HttpSession unworkable: If you try to use them, an IllegalStateException is thrown. This is even true on the `invalidate()` method itself: You can't use this method on an already invalid session! The three HttpSession methods that don't throw this exception are `get` and `setMaxInactiveInterval()`, and `getServletContext()`—quite why these are unlike the other methods is not clear to me—but I point it out in case some meaner-than-usual exam question tries to trip you up.

There are two methods that return a date and time (as a **long** primitive that you will most likely feed to the appropriate java.util.Date constructor):

- `getCreationTime()`—unsurprisingly, the time the session was created.
- `getLastAccessedTime()`—the last time the client sent a request associated with the session. Clearly, this API is useful to the web container itself in determining when to invalidate the session according to the time-out value.

This leaves `getServletContext()`, `getId()`, and a sprinkling of deprecated methods. `HttpSession.getServletContext()` returns the SessionContext to which the session is attached, so it can be used as an alternative to `ServletConfig.getServletContext()` (the method you normally invoke directly from a servlet). `HttpSession.getId()` we explore in the next section. You won't be tested on the deprecated APIs. That said, you might want to familiarize yourself with them if you are maintaining older code (supporting Servlet Spec 2.2 and before). Most have to do with a standardization of method names whereby session attributes used to be known as session values—for example, `get` and `putValue()` were used once upon a time instead of `get` and `setAttribute()`.

## EXERCISE 4-1


### Displaying the Session Life Cycle

In this exercise, you'll write a servlet that associates itself with a session and displays information about the session: how many times the session has been accessed, the session's age, and whether it's a new session.

Create the usual web application directory structure under a directory called ex0401, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch04/ex0401.war—check there if you get stuck.

#### Set Up the Deployment Descriptor

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

#### Write the SessionDisplayer Servlet

2. Create a Java source file SessionDisplayer.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
3. Do the necessary preliminaries to obtain the reponse's PrintWriter, and set the content type to "text/html."
4. Check for a parameter called *getSession* (`request.getParameter ("getSession")`). Most of the remaining steps should depend on the value of *getSession* being "true."
5. Get hold of the session from the request.
6. If this is a new session, display the fact. Also, if the session is new, set up a session attribute that records the number of accesses to the session, initialized to a value of 1.
7. Increment this session attribute on every subsequent access to the session.
8. Display the number of times this session has been accessed, retrieving the information from the session attribute.
9. Get the time the session was created. Display the session's age in minutes and seconds by obtaining the current time and by working out the difference between this and the session creation time.

### Run the SessionDisplayer Servlet

10. Deploy and run the servlet, using a URL such as

```
http://localhost:8080/ex0401/SessionDisplayer?getSession=true
```

11. Try recalling the servlet with the above URL, sometimes changing the *getSession* parameter to a value of "false."

### Optional Experiments

12. Add capabilities to the servlet to reset the maximum inactive interval and to immediately invalidate the session. These could be controlled by parameters into the session.
13. Try various session method calls immediately after the session is invalidated. Remind yourself which three methods can (strangely but legally) still be called even though the session is invalid.

---

**CERTIFICATION OBJECTIVE**

# Session Management (Exam Objective 4.4)

*Given a scenario, describe which session management mechanism the web container could employ, how cookies might be used to manage sessions, and how URL rewriting might be used to manage sessions, and write servlet code to perform URL rewriting.*

So now you have a good grasp of session scope. To complete your understanding of the session life cycle, we need to look at the actual mechanisms used to keep a session up and running. So far we have talked about exchanging tracking information between the client (browser) and the web application. Now we'll see precisely what that tracking information might consist of, according to which type of session management mechanism is in force.

## Session Management

There are two principal methods for session management "officially" recognized by the servlet API. One method is management by cookie exchange, and the other is management by rewriting URLs. In this section we'll explore in some detail what these mean.

That's not to say that these are the only approaches. Secure Sockets Layer (SSL), which ensures secure communications between browsers and web applications, has its own session data built into it. You can always manufacture your own mechanism: A typical technique involves holding session data in hidden fields on a web form. However, you don't need understanding of these other approaches for the web component exam.

## Session Management: General Principles

Whatever session management approach you take, the end game is the same: to associate a group of requests. Each of these requests needs to carry a unique ID, which identifies the session to which it belongs. Indeed, once you have a handle to the session, you can display this ID using the `HttpSession.getId()` method. This returns a String whose contents will depend on the application server you use—you can usually expect it to be long!

The web application will allocate this unique ID on the first request from the client (at which point the client has no idea about its value). The ID must be passed back to the client so that the client can pass it back again with its next request. In this way, the web application will know to which session the request belongs. This implies that the client must need to store the unique ID somewhere—and that's where session management mechanisms come in.

## Cookies

The preferred way for a web application to pass session IDs back to a client is via a cookie in the response. This is a small text file that the client stores somewhere. Storage can be on disk but is just as likely to in memory: This is the case with a "transient" cookie, which is useful for a short time only. You are no doubt well aware of cookies just from your general use of the Internet, for many web sites employ them. Cookies are used to support all kinds of session-like activity on the Web, regardless of whether the back-end technology is Java based.

In the case of J2EE web applications, the cookie returned has a standard name—JSESSIONID—all uppercase, just as written here. Even if you can't inspect the cookie at the browser end (because it's probably transient—in memory, and not available to view from the hard disk using a text editor), you can catch the cookie coming back from the browser using the `HttpServletRequest.getCookies()` method that we examined in Chapter 1. You can then prove to your own satisfaction that the value of the cookie matches the session ID you can view with `HttpSession.getId()`.

You probably know also that client browsers—mostly—have the ability to switch cookies off. Although most cookies are benign in nature, and designed to enhance a

user's web experience, the malign few have given cookies a bad name. So for privacy reasons, a user may choose to reject cookies completely. For this reason, you might need to use the second (and second-best) standard session-tracking mechanism—URL rewriting.

## URL Rewriting

We have already seen in earlier chapters how to pass information to a web application via the URL. This might take the form of path information (supplementary "directory" information appended to a servlet path) or parameters in a query string (name/value pairs after a question mark).

URL rewriting is an extension of this idea. A "pseudo-parameter" called *jsessionid* is placed in the URL between the servlet name (with path information, if present) and the query string. Here's an example—note that this would be one continuous line on your browser address line with no breaks:

```
http://localhost:8080/examp0401/SessionExample;jsessionid=
58112645388D9380808A726A27F92997?name=value&othername=othervalue
```

You see a semicolon after the servlet mapping name (`SessionExample`), then the *jsessionid* pseudo-parameter (`jsessionid=verylongstring`), followed by a question mark, which introduces the query string information.

This is fine, but it gives your web application a real problem. Every web page your application returns in the response is likely to have a number of hyperlinks within it of one sort or other—regular links, buttons, image links, or whatever. Each one of these links must contain `jsessionid=<correctLongString>` as part of the URL.

This is very hard to achieve, unless you have servlets dynamically generating those web pages. There is a method—`HttpServletResponse.encodeURL()`—which accepts a String (representing the URL link on the web page minus session information) and returns a String (the same URL link, but now with the session information embedded). This method is, in fact, clever enough to know that if some other session mechanism is in force—at least one it recognizes, such as cookies—then there is no need to bother embedding any session information. Under these circumstances, it returns the String representing the URL unchanged. Best practice dictates that every URL link you create in a servlet should be put through this method. Then, even if you are expecting your application to operate in a cookie-friendly environment, it will still survive when it unexpectedly finds itself in cookie-hostile territory.

As a postscript to this, there is another method, `HttpServletResponse.encodeRedirectURL()`, which operates in pretty much the same way as

`encodeURL()`. You give it a URL String; it gives back a URL String, with *jsessionid* embedded where necessary. This resulting URL String should then be used to plug into the `HttpServletResponse.sendRedirect()` method. The output String will look no different from a similar call to `encodeURL()`; the reason for providing `encodeRedirectURL()` is that the logic for determining whether or not to embed session information may be different when considering normal URLs versus redirect URLs.

**exam Watch**

***Beware of the deprecated methods `encodeUrl()` and `encodeRedirectUrl()`. These have "Url" in mixed case and were deprecated in a Java standardization exercise that mandated capitals for the abbreviation URL wherever it appeared in method or other names. It would be a mean exam question that attempted to trip you up on this arcane point, but you have to remember that examiners are entitled to get their kicks in any way they can.***

## Request Methods

There are a couple of HttpServletRequest methods that identify which of the two standard session mechanisms are in use—cookies or URL rewriting. You might find it surprising that these methods belong to the request object—not to HttpSession. There's a minor advantage in that you can execute these methods without having to first access the session object via the request. Here are the two methods:

- `HttpServletRequest.isRequestedSessionIdFromCookie()`
- `HttpServletRequest.isRequestedSessionIdFromURL()` (and yes, there is a deprecated version of this method, where "URL" is in mixed case: "Url")

Never use a pithy name for a method when you can use a sentence instead! Both methods return a primitive **boolean**, set to **true** if the session mechanism specified is in force. There can be circumstances where both of these methods, called consecutively for the same request, both return **false**—even though a session is present. This will happen:

- for SSL sessions.
- for bespoke session mechanism logic (hidden form fields, for example).
- when the session is new! Because at this point, the session ID isn't coming from a URL or a cookie—but it has been generated by the web container.

## EXERCISE 4-2


### Displaying the Session Management Mechanism

This exercise builds on the SessionDisplayer servlet that you wrote in Exercise 4-1. You'll now add some capabilities to display what kind of mechanism is in use for supporting the session.

Create the usual web application directory structure under a directory called ex0402, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch04/ex0402.war—check there if you get stuck.

#### Set Up the Deployment Descriptor

1. Copy the web.xml deployment descriptor from Exercise 4-1. You'll recall that this declared the SessionDisplayer servlet.
2. Change each occurrence of SessionDisplayer (name of servlet, name of class, URL pattern for mapping) to SessionDisplayer2.

#### Update the SessionDisplayer2 Servlet

3. Copy the Java source file for SessionDisplayer.java from ex0401/WEB-INF/classes (or its appropriate package directory) to ex0402/WEB-INF/classes (or appropriate package directory), and rename it to SessionDisplayer2 (make sure the class declaration reflects this as well).
4. In the code—if you haven't done so already—put an <a href> link that recalls this same SessionDisplayer2 servlet. Encode the URL using the appropriate HttpServletResponse method.
5. In the web page, display some text that shows whether the session came from the JSESSIONID cookie or from the URL.
6. Optionally, use the `request.getCookies()` method to get hold of the JSESSIONID cookie, and display all the attributes of the cookie that you can on the web page.

#### Run the Updated SessionDisplayer2 Servlet

7. Deploy and run the servlet, using a URL such as

```
http://localhost:8080/ex0402/SessionDisplayer2?getSession=true
```

8. You're most likely to find that your browser uses cookies as the session mechanism. To test out the URL redirection method, try turning off cookies altogether in your browser — or target the domain where your web server (Tomcat) is running, usually localhost / 127.0.0.1, and turn off cookies for that. Ensure that you restart your browser before expecting URLs to be used instead of cookies.
9. The screen print below shows part of the output from the solution code, when the session information is delivered through the JSESSIONID cookie.

## CERTIFICATION OBJECTIVE

# Request and Context Listeners (Exam Objective 3.4)

*Describe the web container lifecycle event model for requests, sessions, and web applications; create and configure listener classes for each scope lifecycle; create and configure scope attribute listener classes; and, given a scenario, identify the proper attribute listener to use.*

We're going to take a brief departure from sessions now and begin to explore the world of listeners. The certification objective actually pertains to the "web container model" and so is a leftover from Chapter 3. For veterans of the old Sun Certified Java Programmer (SCJP) exam—which included Swing user interface mechanisms on its syllabus—listeners will not be a new idea. The idea is simple and elegant: Something of interest happens in your framework, and the framework lets the interested parties know. The interested parties are called "listeners" in Java (and design pattern) parlance. And whereas the Swing framework has listeners for mouse movements and keyboard strokes, the J2EE web application model has a set of server-side events that you can listen for. These are what we're going to cover in the next two sections.

## Listeners

We'll start in this section with the listeners that apply to request and context objects. There are two for each object, and their function is very similar. The following table shows the listeners, what objects they apply to, and the listener function.

| Listener Interface Name | Applies to | Function |
|---|---|---|
| ServletRequestListener | Request objects | Responds to the life and death of each request. |
| ServletContextListener | The context object | Responds to the life and death of the context for a web application. |
| ServletRequestAttributeListener | Request objects | Responds to any change to the set of attributes attached to a request object. |
| ServletContextAttributeListener | The context object | Responds to any change to the set of attributes attached to the context object. |

exam Watch

***There is no getting away from it—you have to memorize the names of all the listeners, together with their methods and their purpose: what events trigger calls to them and what object they affect. It's a lot to ask—and the SCWCD asks it.***

## Listener Preparation

There are two things you need to do to set up a listener in a web application:

- Write a class that implements the appropriate listener interface.
- Register the class name in the web application deployment descriptor, web.xml.

Let's first deal with the deployment descriptor aspects, which are reasonably trivial. All you need to do is to place a `<listener>` element somewhere underneath the root element, and with this embed a `<listener-class>` element. The value held in the `<listener-class>` element is the fully qualified class name of a listener class. No need to specify which type of listener you're talking about: The web container just works this out through Java's reflection capabilities. This neatly covers the fact that the same class could, potentially, implement more than one kind of interface. Here's how some listener declarations might look in web.xml:

```
<listener>
  <listener-class>com.osborne.RequestTrackingListener</listener-class>
</listener>
<listener>
  <listener-class>com.osborne.SessionLoggingListener</listener-class>
</listener>
```

It could be that you have more than one listener class implementing the same interface. Moreover, you might care about the order in which the classes are called when a triggering event occurs. Simply list your listener declarations in the desired order in the deployment descriptor, and let the web container ensure the correct invocation sequence.

exam Watch

***Closedown of a web application triggers a call to the matching closedown events in session and context listeners. The order in which listeners are called is then in reverse order of deployment description declaration, with session listeners being processed before context listeners.***

Writing a listener class simply involves providing the requisite methods to satisfy the interface you are implementing. You can always write a "do-nothing" method if there are some events that don't interest your web application. It's worth remembering that listener classes must have a no-argument constructor. You can let the Java compiler supply one automatically if no other constructors are present. However, this approach is vulnerable if constructors with arguments are added later. (But it's not obvious why they would be—the web container is going to instantiate your listener only through the no-argument constructor. And there wouldn't be a good reason to instantiate a listener in your own web application code.)

## The Request Listener

So now that we've explored the general features of listener classes, it's time to start looking in detail at each listener in turn. We'll start with ServletRequestListener. A class implementing this interface has two methods to implement: `requestInitialized()` and `requestDestroyed()`. The names of the methods pretty much describe the events that trigger the web container to call them. So `requestInitialized()` is called the moment that any request in the web container becomes newly available—in other words, at the beginning of any request's scope. This is at the beginning of a servlet's `service()` method—or earlier than that if a filter chain is involved (the request's scope begins at the first `doFilter()` method call of the chain). Conversely, `requestDestroyed()` is called for each request that comes to an end—either at the end of the servlet's `service()` method or at the end of the `doFilter()` method for the first filter in a chain. This is shown diagrammatically in Figure 4-2.

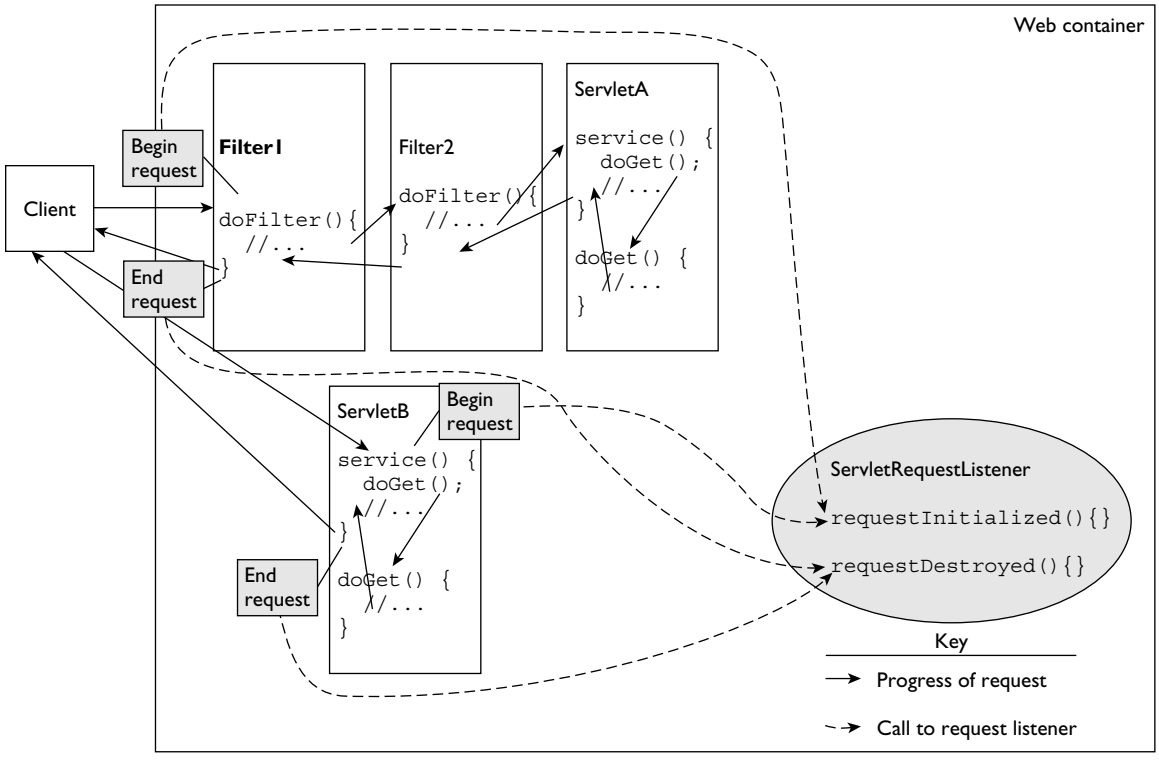
Each of these ServletRequestListener methods accepts a ServletRequestEvent as a parameter. This event object has two methods for access to useful objects:

- `getServletContext()` returns the ServletContext for a web application.
- `getServletRequest()` returns the ServletRequest object itself (cast this to HttpServletRequest if you need to).

You write code like the following in your ServletRequestListener class to preload an attribute into every request made to your web application:

```
public void requestInitialized(ServletRequestEvent requestEvent) {
  HttpServletRequest request = (HttpServletRequest)
                               requestEvent.getServletRequest();
  request.setAttribute("com.osborne.bookrecommendation",
                       "Core JSPs 2.0");
}
```

**FIGURE 4-2** Two Requests Triggering Request Events




## The Request Attribute Listener

So we've now dealt with our first listener—ServletRequestListener—which deals with the life cycle of each request object. What about the life cycle of the attributes attached to request objects? For these we have classes that implement the Servlet RequestAttributeListener interface. Here are the methods to implement:

- `attributeAdded(ServletRequestAttributeEvent srae)` is called whenever a new attribute is added to any request. In other words, *any* call (from *any* request object at *any* time) to `ServletRequest.setAttribute()` will trigger a call to this method—provided that the name of the attribute being added to the request is a not a name already in use as an attribute of that request.
- `attributeRemoved(ServletRequestAttributeEvent srae)` is called whenever an attribute is removed from a request (as a result of any call to `ServletRequest.removeAttribute()`).

- `attributeReplaced(ServletRequestAttributeEvent srae)` is called whenever an attribute is replaced (as a result of any call to `ServletRequest.setAttribute()` for an attribute name already in use on the request whose call this is).

Again, there are two useful methods on the event object passed as a parameter to these methods. The method `getName()` is straightforward: It returns the String holding the name of the attribute being added, removed, or replaced. `getValue()` is less clear-cut, for what's returned varies slightly in meaning:

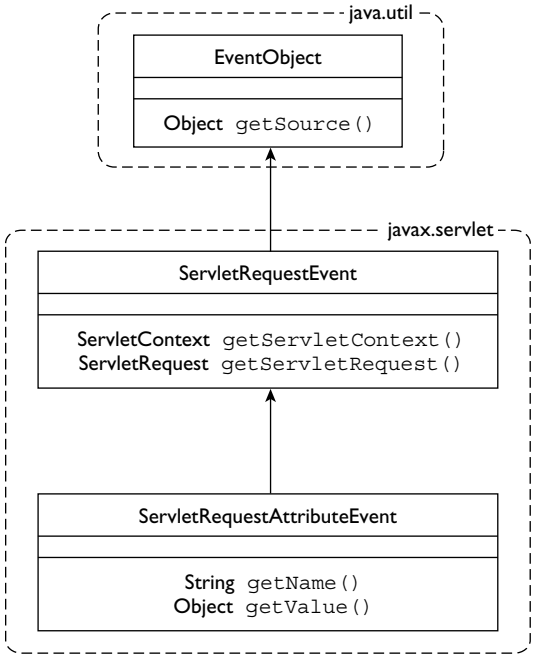
- `attributeAdded()`, `getValue()` returns the Object that is the value parameter on the `setAttribute()` call.
- `attributeRemoved()`, `getValue()` returns the Object that has been removed as a value from the request as a result of a `removeAttribute()` call.
- `attributeReplaced()`, `getValue()` returns the *old* value of the attribute before a call to `setAttribute()` changed it. Why not the new value? Because—as we'll see in a moment—it's possible to get at the new value by alternative means. But there's no other way of trapping the old value at this point.

Because ServletRequestAttributeEvent inherits from ServletRequestEvent, you get the two handy methods we looked at in the previous section—which allow you to get the context object and the request object. Having the request object, you can always get to the current value of an attribute that's just been replaced. Here's some code that displays to the server console the old and new values for a replaced attribute:

```
public void attributeReplaced(ServletRequestAttributeEvent event) {
  String name = event.getName();
  Object oldValue = event.getValue();
  Object newValue = event.getServletRequest().getAttribute(name);
  System.out.println("Name of attribute: " + name);
  System.out.println("Old value of attribute: " + oldValue);
  System.out.println("New value of attribute: " + newValue);
}
```

The inheritance chain for ServletRequestAttributeEvent doesn't stop there—as you can see in the illustration on the following page:

The "grandparent" of ServletRequestEvent is java.util.EventObject. You might remember this from user interface programming, for it features in the hierarchy of Swing events also. This has one method—`getSource()`—which returns the object that is the source of the event. This—surprisingly perhaps—proves to be the ServletContext object: It represents the web application framework, which is, ultimately, the source of all events.




## The Context Listener

Now that you've learned about ServletRequest Listeners, you'll find the ServletContextListener easy to learn, for it follows just the same pattern. Instead of `requestInitialized()` and `requestDestroyed()`, you have `contextInitialized()` and `contextDestroyed()` as the two methods to implement. And in life cycle terms, these are called at the beginning and end of scope—this time of the context, of course, rather than of the request. And as we learned previously, the context life cycle matches that of the web application: It's the first object made available on web application startup and the last to disappear at shutdown. So the `contextInitialized()` method gets called before any servlet's `init()` method or any filter's `doFilter()` method. And every filter and servlet `destroy()` method must have executed before the `contextDestroyed()` method is called.

Both the methods get passed a ServletContextEvent object, which just has the one method, `getServletContext()`, to get at the context object itself. So in `contextInitialized()`, you have a chance to attach context attributes before any servlet gets a crack of the whip.

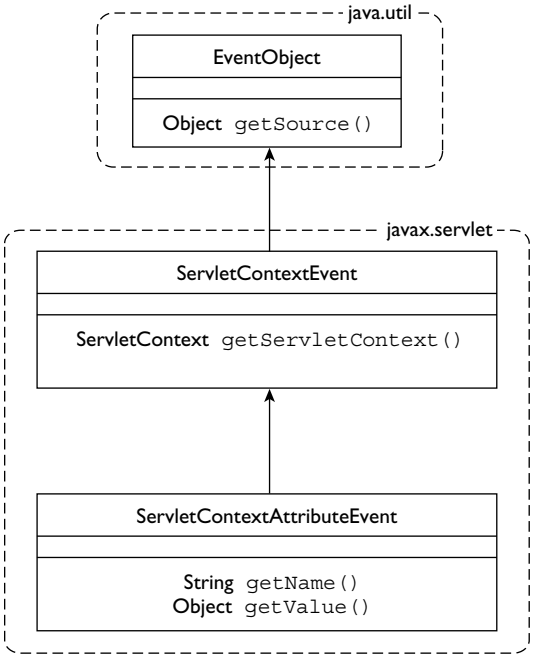
on the job

***The `contextInitialized()` method of a ServletContextListener is a great place to read in parameters from initialization files that are fundamental to the operation of your application. It's a better alternative than relying on the `init()` method in a servlet that loads on startup. Although you can configure your servlet to be the first one that loads in the application, that's vulnerable to later configuration changes. But you can guarantee that the `contextInitialized()` method will be the first piece of your code to run on startup of the web application.***

## The Context Attribute Listener

The ServletContextAttributeListener has the same trio of methods as the Servlet RequestAttributeListener: namely `attributeAdded()`, `attributeRemoved()`, and `attributeReplaced()`. They have the same function as their request equivalents—except, of course, that they fire when things happen to context attributes:

- `attributeAdded(ServletContextAttributeEvent scae)` is called whenever a new attribute is added to the servlet context. In other words, any call (from any web application code that has access to the servlet context) to `ServletContext.setAttribute()` will trigger a call to this method—provided that the name of the attribute being added to the context is a not a name already in use as an attribute of the context.
- `attributeRemoved(ServletContextAttributeEvent scae)` is called whenever an attribute is removed from the context (as a result of any call to `ServletContext.removeAttribute()`).
- `attributeReplaced(ServletContextAttributeEvent scae)` is called whenever an attribute is replaced (as a result of any call to `ServletContext.setAttribute()` for an attribute name already in use by the servlet context).




And lo—the ServletContextAttributeEvent received as a parameter by these methods has the same two methods as the equivalent ServletRequestAttributeEvent—namely `getName()` (to get the name of the attribute affected) and `getValue()` (to get the value of the attribute: added, removed, or—in the case of replacement—the *old* value of the attribute).

There's an inheritance hierarchy for this as well, back through ServletContextEvent and java.util.EventObject, as illustrated.

## EXERCISE 4-3


### Proving the Execution Order of Listeners

In this exercise you'll write code to explore context listeners. In particular, you will prove that the `contextInitialized()` method is called before any servlet is initialized. You'll also write code to trap changes to context attributes.

Create the usual web application directory structure under a directory called ex0403, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch04/ex0403.war—check there if you get stuck.

### Set Up the Deployment Descriptor

1. Declare a servlet named SetContextAttributes, with a suitable servlet mapping. Ensure that it loads on startup of the web server. If needed, refer to Chapter 2 to refresh yourself on `<servlet>` element setup.
2. Declare two listeners, MyContextListener and MyContextAttributeListener.

### Write the SetContextAttributes Servlet

3. Create a Java source file SetContextAttributes.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
4. Write an `init()` method, using `System.out.println()` to send a message to the console.
5. Write a `doGet()` method, which adds, replaces, and removes one or more context attributes. Optionally, output some text on the response so that you'll know when the servlet has been called successfully (to be useful, this might list all the context attributes).

### Write MyContextListener

6. Create a Java source file MyContextListener.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, implementing javax.servlet.ServletContextListener.
7. Write a `contextInitialized()` method, which sends a message to the console and creates a context attribute.
8. Write a `contextDestroyed()` method, which sends a message to the console and removes the context attribute you added in the `contextInitialized()` method.

### Write MyContextAttributeListener

9. Create a Java source file MyContextAttributeListener.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, implementing javax.servlet.ServletContextAttributeListener.
10. Write an `attributeAdded()` method, which displays the name and value of the added attribute on the server console.
11. Write an `attributeReplaced()` method, which displays the name, old value, and new value of the replaced attribute on the server console.

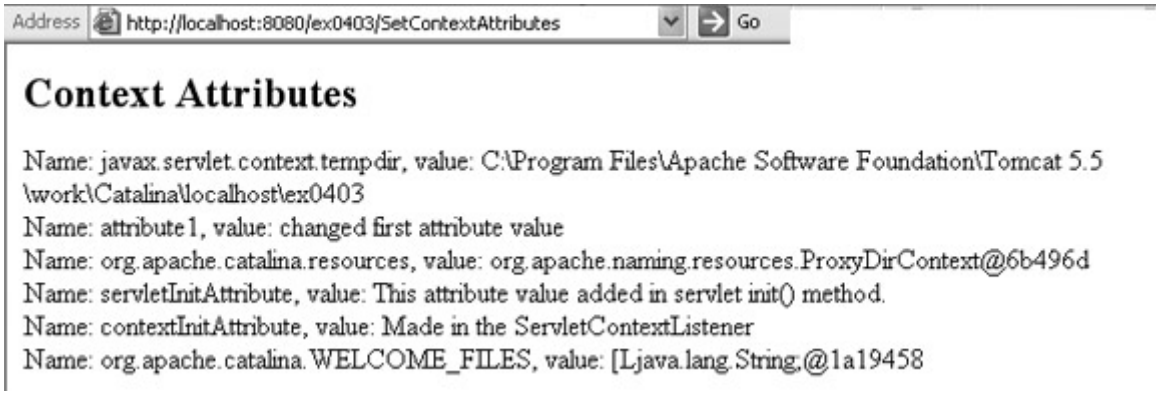
12. Write an `attributeRemoved()` method, which displays the name and value of the removed attribute on the server console.

### Deploy and Run

13. Deploy the WAR file. Check the console messages you get on deployment: Do they match what you expected to get?
14. Call the SetContextAttributes servlet with a URL such as

    ```
    http://localhost:8080/ex0403/SetContextAttributes
    ```

15. Again, check the console messages—do the methods get called in My ContextAttributeListener as you would expect?
16. Undeploy the WAR file (for instructions on how to do this for Tomcat, see Appendix B). Yet again, check the console messages. Does the `contextDestroyed()` method in MyContextListener still have access to context attributes?
17. Here is sample browser output from the SetContextAttributes servlet:




## CERTIFICATION OBJECTIVE

# Session Listeners (Exam Objective 4.3)

*Using session listeners, write code to respond to an event when an object is added to a session, and write code to respond to an event when a session object migrates from one VM to another.*

We covered request and context listeners in the last section—both those for the requests and context themselves, and separate pairs of listeners for their attributes. Now we're going to meet the set of listeners available to session objects. Because the session has—potentially—a more exciting and diverse lifetime than requests or contexts, it may not surprise you to learn that the session has some additional listener interfaces that don't occur in other scopes.

## Session-Related Listeners

Sessions have two listeners that are equivalent in every way to the lifetime and attribute listeners we've already met for context and request:

- HttpSessionListener, which is very like ServletContextListener and ServletRequestListener
- HttpSessionAttributeListener, which is very like ServletContextAttributeListener and ServletRequestAttributeListener

But there are also a couple of extra listeners related to sessions—or more correctly, session attribute value objects. In a typical web application, session attributes are more numerous and volatile than the attributes attached to request or context. Consequently, the value objects that might be attached to a named session attribute can implement a couple of interfaces of their own:

- HttpSessionBindingListener receives events when a value object is used as a session attribute.
- HttpSesssionActivationListener receives events when a value object is transported across JVMs. This happens when the object is an attribute of a session in a distributed environment.

So now we'll look in a little more detail at each of these listener interfaces in turn.

### HttpSessionListener

Like every other listener we have looked at so far, a class implementing HttpSessionListener must be set up in the deployment descriptor. The rules are described in the "Listener Preparation" section of the chapter.

HttpSessionListener has two methods, like its request and context counterparts—and again these fire at the beginning and end of scope. There's a nasty little difference in the naming convention for these methods, though. The method `sessionDestroyed()` matches the pattern of `requestDestroyed()` and `contextDestroyed()`, and it marks the end of a session. However, the beginning of a session

is marked by a `sessionCreated()` event—which doesn't follow the pattern of `requestInitalized()` and `contextInitialized()`.

Let's look at the methods in more detail.

**`sessionCreated(HttpSessionEvent event)`** This method is called by the web container the moment after a request first calls the `getSession()` method—in other words, whenever a new session is provided. All the subsequent `HttpServletRequest.getSession()` calls will return the same existing session—which is, of course, not a cue for firing a call to this event. The `sessionCreated()` method receives an event, of type HttpSessionEvent. This has only the one method of its own, which is `getSession()`—to return the HttpSession object that has just been created.

**`sessionDestroyed(HttpSessionEvent event)`** This method is called by the web container at the moment a session is about to be invalidated—within the call to `HttpSession.invalidate()`, but before the session becomes invalid and unusable. Just as for `sessionCreated()`, an HttpSessionEvent object is passed as a parameter to the method, which gives access to the about-to-be-invalidated session through its `getSession()` method. Whether the call to `HttpSession.invalidate()` comes about as a result of your own explicit call, or the web container timing out a session, the effect is the same: The `sessionDestroyed()` method will fire.

**exam Watch**

***The behavior of this method has changed since previous versions of the exam! It used to be that the call to `sessionDestroyed()` came after the session had been invalidated.***

***This isn't the case anymore. The method is now called before the invalidation process begins—before any attributes are stripped from the session and their corresponding life cycle events are called.***

## HttpSessionAttributeListener

HttpSessionAttributeListener is just like ServletRequestAttributeListener. Here are the methods to implement:

- `attributeAdded(HttpSessionBindingEvent hsbe)` is called whenever a new attribute is added to any session. In other words, any call (from any session

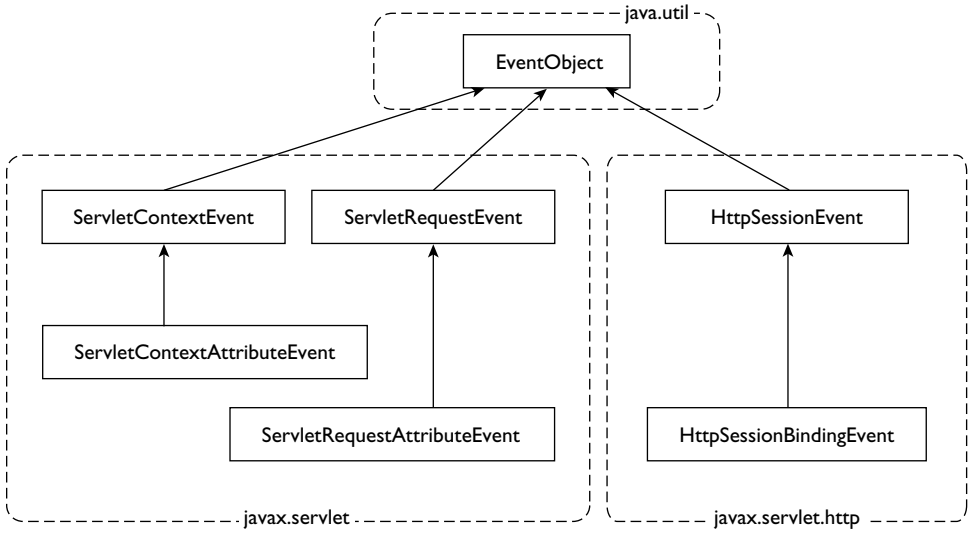
object at any time) to `HttpSession.setAttribute()` will trigger a call to this method—provided that the name of the attribute being added to the session is a not a name already in use as an attribute of that session.

- `attributeRemoved(HttpSessionBindingEvent srae)` is called whenever an attribute is removed from any session (as a result of any call to `HttpSession.removeAttribute()`).
- `attributeReplaced(HttpSessionBindingEvent srae)` is called whenever an attribute is replaced (as a result of any call to `HttpSession.setAttribute()` for an attribute name already in use on the session whose call this is).

Again, the event object passed as parameter—this time HttpSessionBinding Event—serves as a conduit to the name and value of the attribute affected, through the `getName()` and `getValue()` methods—see the notes above on ServletContext AttributeEvent and ServletRequestAttributeEvent for a full explanation: The rules are the same. You might expect HttpSessionBindingEvent to inherit from Http SessionEvent—following the pattern of ServletContextAttributeEvent inheriting from ServletContextEvent and ServletRequestAttributeEvent inheriting from ServletRequestEvent—and indeed it does. See the comparative inheritance hierarchies in Figure 4-3.

**FIGURE 4-3**

Comparative Inheritance Hierarchies for Event Classes



**exam Watch**

***Most of the time, you can rely on listeners and events having matching names. So* HttpSession*Listener goes with* HttpSession*Event,* ServletRequest*Listener goes with* ServletRequest*Event, and* ServletContextAttribute*Listener goes with* ServletContextAttribute*Event. But there's a mismatch for HttpSessionAttributeListener. Its methods take an HttpSession*Binding*Event as a parameter. HttpSessionAttribute*Event *does not exist—except in fallacious exam answers!***

## Session-Related Listeners Not Declared in the Deployment Descriptor

Now we move on to two other session-related listeners which are different in character to the listeners we have previously encountered, whether on session, request, or context. These listeners are:

- HttpSessionBindingListener
- HttpSessionActivationListener

Classes implementing these listener interfaces are *not* declared in the deployment descriptor. They become known to the web container through a different mechanism entirely. We learn about this and other aspects of session binding and activation listeners in the following sections.

### HttpSessionBindingListener

HttpSessionBindingListener is the next listener interface we'll consider. It's very easy to misunderstand its function and confuse it with HttpSessionAttributeListener. You'll see that its methods even receive the same kinds of event, namely HttpSession BindingEvent (so this time, the name of the event does match the listener name). However, this listener is *not* declared in the deployment descriptor web.xml. Instead, it's implemented by an object you intend to use as the "value" parameter in a call to `HttpSession.setAttribute(String name, Object value)`. So whereas any HttpSessionAttributeListener classes are funnels for any update to any attribute on any session, an HttpSessionBindingListener class has methods that are called only on the individual object being used as a session attribute. Let's first find out what the methods are and next see an example of HttpSessionBindingListener:

- `valueBound(HttpSessionBindingEvent hsbe)` is called whenever the object implementing the interface is the value object passed to an `HttpSession.setAttribute()` call.
- `valueUnbound(HttpSessionBindingEvent hsbe)` is called whenever the object implementing the interface is removed from the session as a result of an `HttpSession.removeAttribute()` call.

Now let's look at the full code for a class that implements HttpSessionBinding Listener. It's called SessionAttrObject: It has one private instance variable (a String called *data*) and prints this to the console when the `valueBound()` or `valueUnbound()` methods are called:

```
public class SessionAttrObject implements HttpSessionBindingListener {
  private String data;
  public SessionAttrObject(String value) {
      data = value;
  }
  public String getData() {return data;}
  public String toString() {return data;}
  public void setData(String data) {
      this.data = data;
  }
  public void valueBound(HttpSessionBindingEvent event) {
    System.out.println("valueBound() call on object " + getData());
  }
  public void valueUnbound(HttpSessionBindingEvent event) {
      System.out.println("valueUnbound() call on object " + getData());
  }
}
```

Let's now consider some servlet code that adds, replaces, and removes session attributes—some of whose values are of type SessionAttrObject:

```
11 SessionAttrObject boundObject1 = new SessionAttrObject("Prometheus1");
12 SessionAttrObject boundObject2 = new SessionAttrObject("Prometheus2");
13 HttpSession session = request.getSession();
14 session.setAttribute("bound", boundObject1);
15 session.setAttribute("bound2", boundObject2);
16 session.setAttribute("nonBound", "Icarus");
17 session.setAttribute("bound", boundObject2);
18 session.setAttribute("bound", null);
19 session.removeAttribute("bound2");
20 session.removeAttribute("nonBound");
```

The output when we execute this code (on the console) looks something like this (line numbers don't appear — they're for reference in the text):

```
01 >B>B> valueBound() called for object Prometheus1
02 >B>B> valueBound() called for object Prometheus2
03 >B>B> valueBound() called for object Prometheus2
04 >U>U> valueUnbound() called for object Prometheus1
05 >U>U> valueUnbound() called for object Prometheus2
06 >U>U> valueUnbound() called for object Prometheus2
```

How does this work? Let's consider what happens in the lines of code:

- In lines 11 and 12, we create two local variables—*boundObject1* and *boundObject2*— of our new HttpSessionBindingListener-implementing class, SessionAttrObject.
- At line 13, we obtain the session.
- At line 14, we set up a new attribute called *bound* and use *boundObject1* as the value for this. This triggers a call to the `valueBound()` method for *boundObject1* (first line of output).
- At line 15, we set up a new attribute called *bound2* and use *boundObject2* as the value for this. This triggers a call to the `valueBound()` method, this time for *boundObject2* (second line of output).
- At line 16, we set up another new attribute called *nonBound* and use a plain String literal as a value for this. There are no listener calls at this point; unsurprisingly, String doesn't implement HttpSessionBindingListener.
- At line 17, we change our first attribute. We replace the existing value (*boundObject1*) with a different value (*boundObject2*). This causes two lines of output. The `valueBound()` method is called for *boundObject2*—sensible enough, as it's being bound to another attribute (third line of output). And because *boundObject1* is displaced from this attribute, there's a call to valueUnbound for *boundObject1* (fourth line of output). So at this point, *boundObject1* isn't tied to any session attribute, but *boundObject2* is tied both to the *bound* and *bound2* attributes.
- At line 18, we remove session attribute *bound* by setting its associated value to **null**, which has the same effect as a `removeAttribute()` call. So *boundObject2* is no longer associated with attribute *bound*, and as a result its `valueUnbound()` method fires (fifth line of output).

- At line 19, we remove session attribute *bound2* with a straight (no chaser) call to `removeAttribute()`. Now *boundObject* is no longer associated with *bound2*, and its `valueUnbound()` method is called again (sixth and last line of output).
- At line 20, we remove the *nonBound* attribute with its plain String value; this has no effect in terms of calls on HttpSessionBindingListener-implementing classes.

**exam Watch**

***What happens if you have one or more objects implementing HttpSessionBindingListener, and have an HttpSessionAttributeListener defined in the deployment descriptor as well? Both have methods that are potentially called when session attributes are added, replaced, or removed—so which is called first? The answer is that the web container must call all appropriate HttpSessionBindingListener `valueBound()` and `valueUnbound()` methods first, and only then call HttpSessionAttributeListener methods `attributeAdded()`, `attributeReplaced()`, or `attributeRemoved()`.***

**exam Watch**

***All listener classes (and that's request and context ones as well as session) have to be declared in the deployment descriptor, web.xml. All, that is, except two—which the session attribute "value object" implements, not a class that is part of the container. These interfaces are HttpSessionBindingListener and HttpSessionActivationListener.***

### HttpSessionActivationListener

HttpSessionActivationListener is the second example of an interface that is *not* declared in the deployment descriptor. Like HttpSessionBindingListener, which we just examined, it's an interface that objects are welcome to implement if they are going to be attributes of a session. This time, however, the event methods that might be called have nothing to do with the addition, replacement, or removal of the attributes themselves. The methods are called in distributed environments, at the point where a session is moved from one JVM to another. In the source JVM, all objects bound to the session need to be serialized, and—of course—deserialized in the JVM that is the destination for the moved session. Armed with this information, we can make sense of the methods:

- `sessionWillPassivate(HttpSessionEvent hse)` is called on each implementing object bound to the session just prior to the serialization of the session (and all its attributes). In Star Trek terms, this is the point just before the characters in the transporter go fuzzy and dematerialize.
- `sessionDidActivate(HttpSessionEvent hse)` is called on each implementing object bound to the session just after deserialization of the session (and all its attributes). To press the Star Trek analogy further than it should boldly go, this is the point where the characters have lost their fuzziness and materialize on the planet's surface.

**exam Watch**

***HttpSessionActivationListener is yet another of those interfaces whose methods don't have a matching parameter name. HttpSessionActivationListener methods take an HttpSessionEvent object as a parameter—and there's no such thing as an HttpSessionActivationEvent.***

We've met HttpSessionEvent as a parameter before—it's used as the parameter for methods on the HttpSessionListener interface. So you'll no doubt recall that it has the one method—`getSession()`—which returns a handle to an HttpSession. In this case, it's the one that is either about to start or has finished migration.

There is a logical condition that any object implementing this interface must fulfill if the web container is to call the `sessionWillPassivate()` or `sessionDidActivate()` method. The object must be bound to the session as one of its current attributes. It's no good if the object has never been the subject of an `HttpSession.setAttribute()` call; equally, if it was once bound to a session but has now been removed, the HttpSessionActivationListener methods will never get called.

## EXERCISE 4-4


### Session Listeners and Order of Execution

In this exercise you'll write code to explore some of the session listeners. As in Exercise 4-3, where you explored the order of execution of different methods in listeners, you'll look in some detail at the more involved rules for session listeners. You'll start with code you've already seen in the "SessionBindingListener" section.

Create the usual web application directory structure under a directory called ex0404, and proceed with the steps for the exercise. There's a solution in the CD in the file sourcecode/ch04/ex0404.war—check there if you get stuck.

### Set Up the Deployment Descriptor

1. Declare a servlet named SetSessionAttributes, with a suitable servlet mapping. Ensure that it loads on startup of the web server. If needed, refer to Chapter 2 to refresh yourself on `<servlet>` element setup.
2. Declare two listeners, MySessionListener and MySessionAttributeListener.

### Write the SessionAttrObject Object

3. Create a Java source file SessionAttrObject.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file: It's a plain object (doesn't extend anything), but it should implement HttpSessionBindingListener.
4. You can steal the code for this object wholesale from the "HttpSessionBindingListener" section (see page 269). This contains implementations of the `valueBound()` and `valueUnbound()` methods (as required for HttpSessionBindingListener), as well as a few utility methods. The object is a simple wrapper for a String, with additional listener features.

### Write the SetSessionAttributes Servlet

5. Create a Java source file SetSessionAttributes.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
6. Write a `doGet()` method, which adds, replaces, and removes one or more SessionAttrObject instances as attributes of the session. Again, you can crib this code from the "HttpSessionBindingListener" section, but don't remove all the session attributes (so you can see what happens when the session is later destroyed).
7. Write additional code in the `doGet()` method that will terminate the session according to some trigger—a request parameter, perhaps.

### Write MySessionListener

8. Create a Java source file MySessionListener.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, implementing javax.servlet.http.HttpSessionListener.
9. Write a `sessionCreated()` method, which sends a message to the console and creates a session attribute.

10. Write a `sessionDestroyed()` method, which sends a message to the console and removes the session attribute created in the `sessionCreated()` method.

### Write MySessionAttributeListener

11. Create a Java source file MySessionAttributeListener.java in /WEB-INF/classes or an appropriate package directory. Write the class declaration in the source file, implementing javax.servlet.http.HttpSessionAttributeListener.
12. Write an `attributeAdded()` method, which displays the name and value of the added attribute on the server console.
13. Write an `attributeReplaced()` method, which displays the name, old value, and new value of the replaced attribute on the server console.
14. Write an `attributeRemoved()` method, which displays the name and value of the removed attribute on the server console.

### Deploy and Run

15. Deploy the WAR file, and call the SetSessionAttributes servlet. You are likely to use this URL:

    ```
    http://localhost:8080/ex0404/SetSessionAttributes
    ```

16. Check the console messages—in particular, the order of method calls for the HttpSessionBindingListener objects and the one HttpSessionAttribute Listener object. Do the methods get called in the order you expected?
17. Terminate the session, using whatever mechanism you put into the Set SessionAttributes servlet to do so (step 7). What further listener method calls do you get, and in what order?
18. The following screen shot shows the solution code console output after pointing a browser to the URL in step 15:

```
Session created method fired...
>Add>Add> attributeAdded() with name: sessionCreatedAttribute, value; Added in s
essionCreated() method
>B>B> valueBound() called for object Prometheus1
>Add>Add> attributeAdded() with name: bound, value; Prometheus1
>B>B> valueBound() called for object Prometheus2
>Add>Add> attributeAdded() with name: bound2, value; Prometheus2
>Add>Add> attributeAdded() with name: nonBound, value; Icarus
>B>B> valueBound() called for object Prometheus2
>U>U> valueUnbound() for object Prometheus1
>Rpl>Rpl> attributeReplaced() with name: bound, OLD value; Prometheus2, NEW valu
e: Prometheus2
>U>U> valueUnbound() for object Prometheus2
>Rmv>Rmv> attributeRemoved() with name: bound, value; Prometheus2
```

19. The following screen shot shows the solution code console output after invalidating the session, which is achieved with a URL of

```
http://localhost:8080/ex0404/SetSessionAttributes?invalidate=true
```

```
Session destroyed method fired...
>>>Session attribute name nonBound, value Icarus
>>>Session attribute name sessionCreatedAttribute, value Added in sessionCreated
() method
>>>Session attribute name bound2, value Prometheus2
>Rmv>Rmv> attributeRemoved() with name: sessionCreatedAttribute, value; Added in
 sessionCreated() method
>Rmv>Rmv> attributeRemoved() with name: nonBound, value; Icarus
>U>U> valueUnbound() for object Prometheus2
>Rmv>Rmv> attributeRemoved() with name: bound2, value; Prometheus2
```

# CERTIFICATION SUMMARY

In this chapter you started by learning about the session life cycle. You saw that a session is something that belongs to the HTTP world (java.servlet.http package) and doesn't have an equivalent in the "naked" servlet environment. Hence, the interface defining session behavior is called "HttpSession." You learned that sessions are obtained from the request (`HttpServletRequest.getSession()`)—perhaps not too surprisingly, for the function of a session is to tie a number of requests together.

You learned that a session is created—quite simply—when a request from a particular client (browser session) first asks for a session. At this point, you learned that the session is "new" and stays that way while the client doesn't know about the session, or if it's told about the session and refuses to join it. You saw that you test for newness of a session with the `HttpSession.isNew()` method. You then got to grips with different ways in which you can get hold of the session, dependent on whether one is there already or not. You saw that a `getSession()` call with no parameters is equivalent to a `getSession(true)` call (single boolean parameter)—and that this will create a session if one does not exist already. You then learned that a `getSession(false)` call can be used if you want only to get hold of a session if one exists already—perhaps because creation of a session can be done only in a servlet validating a user ID and password, for example.

Having seen how sessions live, you learned how they die: either through your servlet code invalidating them explicitly (`HttpSession.invalidate()`) or through a time-out mechanism. You learned about three possible time-out mechanisms. The

first is a global default provided by some web container-specific mechanism. The second is in a web application's deployment descriptor—setting a time in minutes in the deployment descriptor like this: `<session-config><session-timeout>30</session-timeout></session-config>`. The third sets a particular session's time-out in code, using the `HttpSession.setMaxInactiveInterval(int seconds)` method. You learned that a session—once invalidated—is pretty much useless: Nearly all methods, if used, will throw an IllegalStateException.

You learned that you can attach attributes to a session in very much the same way as you can for request and context scopes. You saw the boundaries of session scope: that a session is confined to a particular web application and that a RequestDispatcher call across web applications will meet a new session. You learned that session information is not strictly thread safe, for two requests can come concurrently from the same client—and that although this is unusual, you might want to take suitable precautions.

You looked then at the tracking information for sessions, and how this is passed between client and server. You saw that there is support for two principal mechanisms: cookies and URL rewriting—although you appreciated that other mechanisms are possible (such as SSL and hidden form fields). You found that cookies are the default mechanism for session support, with URL rewriting a moderately poor second. You learned that the cookie used in session support is called JSESSIONID and that the name part of the name/value pair passed in URLs is jsessionid. The uppercase/lowercase difference is significant. With cookies, you saw that there is no particular action you need to take in your code to provide support—the web container does it for you. By contrast, you learned that URL rewriting requires every link you encode in your servlets to be run through one of two methods—`HttpServletResponse.encodeURL` (for regular links) or `HttpServletResponse.endcodeRedirectURL` (for redirect links). You also saw that HttpSession has methods to determine if the session requested is supported by cookies or by URL rewriting—from the long-named `isRequestedSessionIdFromCookie()` and `isRequestedSessionIdFromURL()` methods.

From there, you left sessions for a while to explore the world of request and context listeners, followed up by session listeners. You saw that listeners are classes that implement interfaces known to the web container framework, with methods called under particular circumstances. You saw that (with a couple of exceptions we'll mention again later) listener classes are declared in the deployment descriptor—each class having a separate `<listener>` element, with a class declared like this: `<listener><listener-class>com.osborne.ListenerClassName</listener-class></listener>`. You met listeners that cover beginning and end of scope:

ServletRequestListener, HttpSessionListener, and ServletContextListener—with corresponding initialized (or created) and destroyed methods, called as the scopes begin and end. You also met listeners covering attributes attached to each scope: ServletRequestAttributeListener, HttpSessionAttributeListener, and ServletContext AttributeListener. You saw the same trio of methods on each of these listeners—for the addition, replacement, and removal of attributes.

You also learned that listener methods invariably accept a listener event class as a parameter. Each one of these listener event classes has methods that yield information about appropriate scope-level objects—such as ServletRequestEvent (with `getServletRequest()` and `getServletContext()` methods), HttpSessionEvent (with a `getSession() method`), and ServletContextEvent (with a `getServlet Context()` method). You saw how the corresponding attribute listener events inherit from the scope-level event classes: ServletRequestAttributeEvent from Servlet RequestEvent, HttpSession*Binding*Event from HttpSessionEvent, and Servlet ContextAttributeEvent from ServletContextEvent. You saw how all these attribute/binding event classes have `getName()` and `getValue()` methods—to obtain the name or value of the attribute added, removed, or replaced. It's obvious what the value is in the case of addition and removal—you saw that it is less obvious in the case of replacement, where `getValue()` gets the *old* value (the value that has been replaced).

You learned that session scope has a couple of additional listeners that are not declared in the deployment descriptor. Both are interfaces designed to be implemented by classes to be used as the values of session attributes. These are HttpSessionBindingListener, whose `valueBound()` and `valueUnbound()` methods are called on the value object as it is added or removed from use as a session attribute. You learned that calls to `HttpSession.setAttribute()` and `HttpSession .removeAttribute()` are typical triggers for calls to the methods both on HttpSessionBindingListener and HttpSessionAttributeListener—and in that case, the HttpSessionBindingListener `valueBound/Unbound()` methods are called first. You also saw that `valueBound/Unbound()` accepts the HttpSessionBinding Event as a parameter—again like the `attributeAdded/Replaced/Removed()` methods of HttpSessionAttributeListener.

Finally, you met the HttpSessionActivationListener. You found that the web container uses classes of this type for distributed applications and will call methods in this class when sessions migrate from one JVM to another. You saw that it is like the HttpSessionBindingListener, in that classes implementing this interface are intended for use as session attributes. You learned that when a session is about to be serialized (prior to migration), the web container calls the `sessionWillPassivate()`

method on each object implementing this interface that is currently attached to the session as an attribute. And you saw that when the session is deserialized in a different JVM, the web container ensures that the `sessionDidActivate()` method is called on the same set of objects. You finally learned that these methods receive an HttpSessionEvent (giving access to the session—the same parameter as received by methods on HttpSessionListener methods).

# TWO-MINUTE DRILL

## Session Life Cycle

- A session is begun when servlet (or filter) code invokes the `HttpServletRequest.getSession()` method.
- A session object is of type HttpSession, in the javax.servlet package. Sessions exist only in the HTTP servlet model; there is no non-HTTP equivalent such as javax.servlet.Session.
- `getSession()` can be called without parameters or with a single **boolean** parameter.
- A call to `getSession()` is equivalent to the call to `getSession(true)`. Both these calls will create a session if none exists already.
- The call `getSession(false)` will not create a session, but it will return a session if one exists already.
- A newly created session is deemed to be "new." This can be tested with the `HttpSession.isNew()` method, which returns a **boolean**—**true** for new, **false** for old.
- There are two possible conditions for "newness": Either the client doesn't know about the session yet or the client has refused to join the session.
- A session is normally confined to one web application and one JVM. However, if a web application is marked `<distributable />` in the deployment descriptor, a session may be cloned into a second running copy of the same web application, in a separate JVM.
- A session cannot cross web applications (unlike a request). A request that gets a session inside of one application, then dispatches to a different web application (different context), and then gets hold of the session in the separate application will get hold of a separate session object.
- Session death can come about through an explicit call (in servlet code) to `HttpSession.invalidate()`.
- Session death is more likely to come about through a time-out mechanism. If there is no activity on a session for a predefined length of time, the web container invalidates the session. There are three time-out mechanisms.
- Time-out mechanism 1: Most J2EE containers establish a "global default" for time-out. How this is specified and achieved is container-specific.

- Time-out mechanism 2: A web application can specify a time-out period—in minutes—in the deployment descriptor, using a `<session-timeout>` element nested inside a `<session-config>` element.
- A negative or zero value for `<session-timeout>` denotes that the session should not ever expire.
- Time-out mechanism 3: Servlet (or filter) code can override the time-out period for any individual session by calling the `HttpSession.setMaxInactiveInterval(int seconds)` method.
- A negative (but *not zero*) value as a parameter to `setMaxInactiveInterval()` denotes that the session should not expire.
- Note the difference in units: minutes in the deployment descriptor (mechanism (2)) and seconds in the HttpSession method (mechanism (3)).
- Apart from three methods, all HttpSession methods fail with an IllegalStateException if any attempt is made to use them after the session has been invalidated.
- You can attach attributes to a session (in the same way as you can for a request or context).
- Session attributes are not—strictly speaking—thread safe. It is possible to have two client windows open making concurrent requests, both sharing the same session.

## Session Management

- Sessions are maintained by passing tracking information between the client and the web application server. There is no alternative, for the connection between client and server is almost always broken after each HTTP request.
- When you obtain a session (using `HttpServletRequest.getSession()`), the web container manufactures a unique ID string for the session. This is passed as a token between the client and server.
- The servlet API recognizes two mechanisms for this token-passing session management—cookies and URL rewriting—but that is not to say that these are the only two.
- Other session mechanisms you might encounter include SSL (Secure Sockets Layer) and hidden form fields.

- ❏ With cookies, the unique ID generated by the web container is embedded as the value of a cookie whose name is JSESSIONID. This name is mandated by the servlet specification and must be spelled exactly as shown (all uppercase).
- ❏ Cookies are the preferred mechanism for J2EE web container session management. Where they can't be used (because the client doesn't support them or has switched off cookies because of privacy concerns), URL rewriting is used instead.
- ❏ With URL rewriting, every HTML link written by a servlet has the unique session ID embedded in the URL itself. The session information is in the form of a name/value pair: *jsessionid=1A2B3C4D* (etc.). The name is always *jsessionid*, spelled exactly as shown (all lowercase).
- ❏ URL rewriting looks very much like parameter passing in the query string, except that the *jsessionid=1A2B3C4D* part comes after a semicolon instead of the question mark denoting the query string. This example shows the session ID appearing before the query string: `http://localhost:8080/ex0402;jsessionid=1A2B3C4D?user=david`.
- ❏ As a servlet code developer, you put any URL for HTML links through the method `HttpServletResponse.encodeURL(String myURL)`. This returns your URL with *jsessionid* information inserted—but only when necessary (if cookies are used, there is no need).
- ❏ URLs that act as parameters for the `HttpServletResponse.sendRedirect()` method should use the `HttpServletResponse.encodeRedirectURL()` method to insert *jsessionid* information. The rules for when its appropriate to insert *jsessionid* may be different from those used in `HttpServletResponse.encodeURL(String myURL)`.
- ❏ The HttpServletRequest object can determine which standard session mechanism is in use through the methods `isRequestedSessionIdFromCookie()` and `isRequestedSessionIdFromURL()`. Both methods return a **boolean**.

## Request and Context Listeners

- ❏ Listeners are part of the web container model. They work in much the same way as listeners in the Swing user interface environment. For both frameworks (web containers and Swing), the listener methods are called in response to relevant events (examples: for Swing, a mouse movement; for the web container, an attribute added).

- There are listeners pertinent to every scope.
- Nearly all listeners (with a couple of exceptions we cover in the session section) should be declared in the deployment descriptor.
- The `<listener>` element has the root element `<web-app>` for its parent. Within the `<listener>` element, you include a `<listener-class>` element — the value for this is the fully qualified class name of a class implementing one or more listener interfaces.
- You need one `<listener>` (with embedded `<listener-class>`) for each listener class you wish to declare in the deployment descriptor.
- Request scope possesses two sorts of listener: ServletRequestListener and ServletRequestAttributeListener. Both of these are interfaces in the javax.servlet package.
- ServletRequestListener listens for the life and death of each request. The corresponding methods called on these events are `requestInitialized()` and `requestDestroyed()`.
- The call to `requestInitalized()` comes at the point where a client request is about to reach its target servlet's `service()` method (or alternatively, where the client request is about to reach the first filter's `doFilter()` method, if the request is intercepted by a filter chain).
- The call to `requestDestroyed()` comes at the point where the request's target servlet `service()` method ends (or alternatively, where the first filter in a chain reaches the end of its `doFilter()` method, with the first filter in the chain being the last to finish executing).
- These methods receive a ServletRequestEvent object as a parameter. From this you can obtain the ServletRequest itself (with `getServletRequest()`) or obtain the web application context (with `getServletContext()`).
- The ServletRequestAttributeListener listens — as you might expect — for any update to the attributes attached to a request.
- A call to `ServletRequest.setAttribute()` with a new name causes a call to the `attributeAdded()` method on ServletRequestAttributeListener.
- A call to `ServletRequest.setAttribute()` with an existing name causes a call to the `attributeReplaced()` method on ServletRequestAttribute Listener.

- A call to `ServletRequest.removeAttribute()` (or `ServletRequest.setAttribute()` with a **null** value) with an existing name causes a call to the `attributeRemoved()` method on ServletRequestAttributeListener.
- Each of the `attributeAdded/Replaced/Removed()` methods receives a ServletRequestAttributeEvent object as a parameter.
- ServletRequestAttributeEvent inherits from ServletRequestEvent (giving access to methods to get at the request and the context). The class also adds two methods of its own: `getName()` and `getValue()` (which return the String name or Object value for the attribute in question).
- `getValue()` returns the *old* value of the attribute in the `attributeReplaced()` method.
- Context (web application) scope possesses two sorts of listener: ServletContextListener and ServletContextAttributeListener. Both of these are interfaces in the javax.servlet package.
- ServletContextListener listens for the life and death of each request. The corresponding methods called on these events are `contextInitialized()` and `contextDestroyed()`.
- The call to `contextInitalized()` comes at the point where a web application starts up, before any request has been processed by a filter's `init()` method or a servlet's `service()` method.
- The call to `contextDestroyed()` comes at the point where a web application is taken out of service. This could be because of the controlled close-down of the server or because the server allows the application to be taken out of service. Every filter and servlet `destroy()` method must execute before this method is called.
- These methods receive a ServletContextEvent object as a parameter. From this you can obtain the ServletContext itself (with `getServletContext()`).
- The ServletContextAttributeListener listens for any update to the attributes attached to a context.
- This listener works in just the same way as ServletRequestAttributeListener: Review the rules above for this and substitute "Context" for "Request" as appropriate.
- The parameter to the listener methods is a ServletContextAttributeEvent object; this inherits from ServletContextEvent (giving access to the `get`

`ServletContext()` method). Following the request pattern, this class adds two methods of its own: `getName()` and `getValue()` (which return the String name or Object value for the attribute in question).

## Session Listeners

- There are four listeners related to sessions.
- HttpSessionListener: very like ServletContextListener and ServletRequest Listener.
- HttpSessionAttributeListener: very like ServletContextAttributeListener and ServletRequestAttributeListener.
- Also HttpSessionBindingListener and HttpSessionActivationListener, which have no counterpart in request and context scope.
- Classes implementing HttpSessionListener and HttpSessionAttributeListener should be set up in the deployment descriptor (like request and context scope listeners).
- Classes implementing HttpSessionBindingListener and HttpSession ActivationListener are intended for use by objects that are used as attribute values on a session, and are *not* declared in the deployment descriptor.
- HttpSessionListener has `sessionCreated()` and `sessionDestroyed()` methods, designed to be called at the beginning and end of a session's life.
- `sessionCreated()` is called when a request from a particular client first asks for a session.
- `sessionDestroyed()` is called when a session object is explicitly invalidated in servlet code (through the `HttpSession.invalidate()` method) or when the web container times the session out because there have been no requests for it for a predefined length of time.
- An HttpSessionEvent object is passed as a parameter to these two methods—this has a `getSession()` method to access the session.
- The HttpSession object is still accessible with all its attributes in the `sessionDestroyed()` method through the HttpSessionEvent object passed as a parameter. This is a change from past implementations of the servlet specification.
- HttpSessionAttributeListener works in the same way as ServletRequest AttributeListener and ServletContextAttributeListener. It has the same

three methods—`attributeAdded/Replaced/Removed()`—called in the same circumstances (but obviously for session attributes, not request or context).

- The parameter passed to these three methods is an HttpSessionBindingEvent object. Like ServletRequestAttributeEvent and ServletContextAttribute Event, this provides `getName()` and `getValue()` methods—which work in the same way.
- HttpSessionBindingEvent inherits from HttpSessionEvent, though which it has a `getSession()` method—to get at the session object whose attributes are affected.
- HttpSessionBindingListener is an interface defined by objects that are used as the values of attributes attached to a session.
- It has two methods: `valueBound()` and `valueUnbound()`.
- `valueBound()` is called when an object is attached as an attribute to a session.
- `valueUnbound()` is called when an object is removed as an attribute from a session.
- These methods on HttpSessionBindingListener are called before any methods on any HttpSessionAttributeListener (which are often triggered by the same events).
- These methods also receive an HttpSessionBindingEvent as a parameter—just as the HttpSessionAttributeListener methods do.
- The fourth and final listener interface for sessions is HttpSessionActivation Listener.
- This has two methods: `sessionWillPassivate()` and `sessionDid Activate()`.
- Both methods receive an HttpSessionEvent object as a parameter (discussed above—this type is a parameter for HttpSessionListener methods as well).
- `sessionWillPassivate()` is called just before a session is serialized to be cloned to another JVM.
- `sessionDidActivate()` is called just after a cloned session is deserialized in a target JVM.

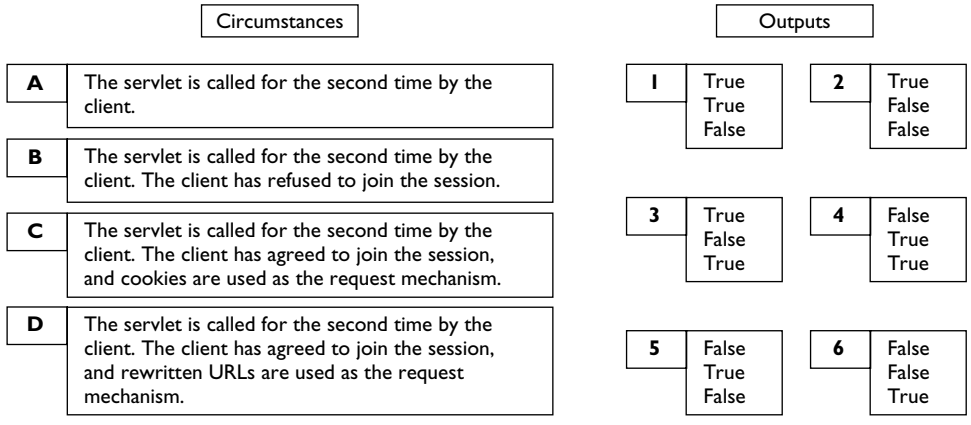
# SELF TEST

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.

## Session Life Cycle

1. (drag-and-drop question) A complete `doGet()` method for a servlet is listed next. Match the circumstances in which the servlet is called with the possible outputs (there are more possible outputs listed than are needed, and any of the possible outputs may be used more than once).

```
protected void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  PrintWriter out = response.getWriter();
  response.setContentType("text/html");
  out.write("<HTML><HEAD>");
  out.write("<TITLE>Session Aspects</TITLE>");
  out.write("</HEAD><BODY>");
  HttpSession session = request.getSession();
  out.write("<BR />" + session.isNew());
  out.write("<BR />" + request.isRequestedSessionIdFromURL());
  out.write("<BR />" + request.isRequestedSessionIdFromCookie());
  out.write("</BODY></HTML>");
}
```



2. What is the outcome of attempting to compile, deploy, and run the following servlet code? Line numbers are for reference only and should not be considered part of the code. (Choose one.)

```
10 import java.io.*;
11 import javax.servlet.*;
12 import javax.servlet.http.*;
13 public class Question2 extends HttpServlet {
14   protected void doGet(ServletRequest request,
15     ServletResponse response) throws ServletException, IOException {
16   HttpSession session = request.getSession(false);
17   session.invalidate();
18   session.setAttribute("illegal", "exception thrown"");
19 }
20 }
```

   A. Won't compile
   B. NullPointerException at line 16 on client's first call to servlet
   C. IllegalStateException at line 18
   D. Some other runtime exception
   E. None of the above

3. Identify the two equivalent method calls in the list below. (Choose two.)

   A. `HttpServletRequest.getSession()`
   B. `ServletRequest.getSession()`
   C. `ServletRequest.getSession(true)`
   D. `HttpServletRequest.getSession(false)`
   E. `ServletRequest.getSession(false)`
   F. `HttpServletRequest.getSession(true)`
   G. `HttpServletRequest.getSession("true")`
   H. `ServletRequest.getSession("false")`

4. Identify true statements about sessions from the list below. (Choose two.)

   A. Sessions can span web applications.
   B. Sessions can be cloned across JVMs.
   C. Sessions are destroyed only after a predefined period of inactivity.
   D. Sessions can be set to never time out.
   E. You can use the deployment descriptor to cause sessions to expire after a set number of requests.

5. Which of the following mechanisms will guarantee that every session in a web application will expire after 1 minute? You can assume that for each answer below, this is the only session timeout mechanism in force for the web application. (Choose two.)

   A. In the deployment descriptor:

   ```
   <session-config>
     <session-timeout>1</session-timeout>
   </session-config>
   ```

   B. In the deployment descriptor:

   ```
   <session-config>
   <session-timeout>60</session-timeout>
   </session-config>
   ```

   C. In the `doFilter()` method of a filter that has the following `<url-pattern>` mapping in the deployment descriptor: "/." *request* is an instance of HttpServletRequest, cast from the ServletRequest parameter passed to the method.

   ```
   HttpSession session = request.getSession();
   session.setMaxInactiveInterval(60);
   ```

   D. In the `doGet()` method of a servlet. *request* is an instance of HttpServletRequest, passed as a parameter to the method.

   ```
   HttpSession session = request.getSession();
   session.setMaxInactiveInterval(1);
   ```

   E. In the `init()` method of a servlet that loads on start up of the web application. *request* is an instance of HttpServletRequest, passed as a parameter to the method.

   ```
   HttpSession session = request.getSession();
   session.setMaxInactiveInterval(60);
   ```

## Session Management

6. Identify the default mechanism for session management from the list below. (Choose one.)

   A. URL rewriting

   B. Hidden Form Fields

   C. Cookies

D. SSL

E. jsessionId request parameter

7. Identify correct statements about session management from the list below. (Choose two.)

A. Session management is usually dependent on a hidden form field called *JSessionId*.

B. The unique identifier for a session may be passed back and forward through a name/value pair in the URL. The name is *jsessionid*.

C. If a cookie used for default session management, there is some flexibility with the name used for the cookie.

D. The cookie used for default session management must be added to the response using the `HttpServletResponse.addCookie(Cookie theCookie)` method.

E. The rules for rewriting URLs for links may be different from those for rewriting URLs for redirection.

8. (drag-and-drop question) In the following illustration, match the concealed parts of the code (lettered) with appropriate choices (numbered) on the right.

```
protected void doGet( A
 request, B response) throws
 ServletException, IOException {

 PrintWriter out = response.getWriter();
 response.setContentType("text/html");
 out.write("<HTML><HEAD>");
 out.write("<TITLE>Session Aspects</TITLE>");
 out.write("</HEAD><BODY>");
 C session = request. D ;
 out.write("\n<P>Session id is <B>" +
  session. E + "</B>.</P>");
 if (request.isRequestedSessionIdFromCookie())
 {
   out.write("\n<P>Session id comes from
    cookie F .</P>");
 }
 if (request. G )
 {
   out.write("\n<P>Session id comes from
    URL element H .</P>");
 }
 String URL = response.encodeURL("Q8");
 out.write("\n<P><A HREF=" + URL
 + ">Link to summon this servlet again.</A>");
 out.write("</BODY></HTML>");
}
```

| | |
|---|---|
| 1 | HttpSession |
| 2 | Session |
| 3 | ServletRequest |
| 4 | HttpServletRequest |
| 5 | getSession() |
| 6 | HttpServletRespons |
| 7 | getHttpSession() |
| 8 | JSESSIONID |
| 9 | isRequestedSessionIdFromURL() |
| 10 | jsessionid |
| 11 | isRequestedSessionIdFromUrl() |
| 12 | ServletResponse |
| 13 | getSessionID() |
| 14 | getId() |

9. Given the following servlet code called with this URL—`http://127.0.0.1:8080/examp0402/Q9`—and also given that URL rewriting is the session mechanism in force, identify the likely output from the servlet from the choices below. (Choose one.)

```
PrintWriter out = response.getWriter();
response.setContentType("text/html");
out.write("<HTML><HEAD>");
out.write("<TITLE>Encoding URLs</TITLE>");
out.write("</HEAD><BODY>");
HttpSession session = request.getSession();
out.write("\n<P>Session id is <B>"
  + session.getId() + "</B>.</P>");
String URL1 = response.encodeURL("Q9");
String URL2 = response.encodeURL
  ("http://127.0.0.1:8080/examp0401/Q1");
out.write("\n<P>URL1: " + URL1 + "</P>");
out.write("\n<P>URL2: " + URL2 + "</P>");
out.write("</BODY></HTML>");
```

A. Output:

Session ID is **4EDF861942E3539B1F3C101B71636C1A**.
URL1: Q9;JSESSIONID=4EDF861942E3539B1F3C101B71636C1A
URL2: http://127.0.0.1:8080/examp0401/Q1

B. Output:

Session ID is **4EDF861942E3539B1F3C101B71636C1A**.
URL1: Q9
URL2: http://127.0.0.1:8080/examp0401/Q1? jsessionid=4EDF861942E3539B1F3C101B71636C1A

C. Output:

Session ID is **4EDF861942E3539B1F3C101B71636C1A**.
URL1: Q9;jsessionid=4EDF861942E3539B1F3C101B71636C1A
URL2: http://127.0.0.1:8080/examp0401/Q1

D. Output:

Session ID is **4EDF861942E3539B1F3C101B71636C1A**.
URL1: Q9;jsessionid=4EDF861942E3539B1F3C101B71636C1A
URL2: http://127.0.0.1:8080/examp0401/Q1;jsessionid=4EDF861942E3539B1F3C101B71636C1A

E. Output:

Session ID is **4EDF861942E3539B1F3C101B71636C1A**.
URL1: Q9?JSESSIONID=4EDF861942E3539B1F3C101B71636C1A
URL2: http://127.0.0.1:8080/examp0401/Q1

10. Which of the following statements contain accurate advice for web developers? (Choose two.)
    A. Because the server determines the session mechanism, there is no need to rewrite URLs when cookies are switched on.
    B. Rewrite every URL embedded in your servlets and JSP code with the `HttpServletResponse.encodeURL()` method.
    C. Cookies are not necessarily supported by J2EE-compliant web containers, so always use URL rewriting as an additional precaution.
    D. Because the client determines whether cookies are permitted or not, it's a good idea always to encode URLs as a fallback session mechanism.
    E. Static pages in your web application can disrupt session management.

## Request and Context Listeners

11. Identify actions that won't fix a potential problem in the following ServletRequestListener code. (Choose two.)

```
01 public void requestDestroyed(ServletRequestEvent reqEvent) {
02 HttpServletRequest request = (HttpServletRequest)
03   reqEvent.getServletRequest();
04 HttpSession session = request.getSession();
05 session.setAttribute("name","value");
06 }
```

    A. Ensure that an HttpSession is created in the `requestInitialized()` method of the same ServletRequestListener.
    B. Ensure that any servlet in your web application obtains a session.
    C. Substitute the code below for lines 04 and 05:

```
HttpSession session = request.getSession(false);
if (session != null) session.setAttribute("name", "value");
```

    D. Take no action, for the code will work in all circumstances.
    E. Place lines 04 and 05 inside a try/catch block.

12. What is the outcome of attempting to compile and run the servlet code below? (Choose one.)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Question12 extends HttpServlet {
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    ServletContext context = getServletContext();
    context.addAttribute("mutable", "firstvalue");
    context.replaceAttribute("mutable", "secondvalue");
    context.removeAttribute("mutable");
    context.removeAttribute("mutable");
  }
}
```

A. There will be three method calls to any ServletContextAttributeListener classes registered to the web application.

B. An exception will be thrown on trying to remove the same attribute for a second time.

C. There will be four method calls to any ServletContextAttributeListener classes registered to the web application.

D. An exception will be thrown if the context cannot be obtained.

E. None of the above.

13. Identify true statements about listener interfaces and related classes from the list below. (Choose three.)

A. It is possible to add context attributes in the `contextDestroyed()` method.

B. During controlled closedown of a web application, the last listener whose methods are potentially called is the ServletContextListener.

C. You can access the current session from methods in classes implementing the Servlet RequestListener interface.

D. You can access the current session from methods in classes implementing the Servlet ContextAttributeListener interface.

E. The ServletContextAttributeEvent class extends java.util.EventObject.

F. It is unwise to change request attributes in the `attributeReplaced()` method of a class implementing the ServletRequestAttributeListener interface.

14. Identify the number and nature of the errors in the code below, which is taken from a class implementing the ServletRequestAttributeListener interface. (Choose one.)

```
01 public void attributeAdded(ServletRequestAttributeEvent event) {
02   HttpServletRequest request = event.getServletRequest();
03   Object o = event.getSource();
04   System.out.println("Source of event is: "
05     + o.getClass().getName());
06   String name = event.getName();
07   String value = event.getValue();
08   System.out.println("In ServletRequestAttributeListener."
09     + "attributeAdded() with name: "
10     + name + ", value; " + value);
11 }
```

A. No compilation errors, no runtime errors
B. No compilation errors, one runtime error
C. One compilation error
D. Two compilation errors
E. Three compilation errors

15. If a request attribute has been replaced, which of the following techniques will not obtain the current (new) value of the attribute? (Choose two.)

A. Use the `ServletRequest.getAttribute()` method anywhere in servlet code following the replacement.
B. Use the `ServletRequestAttributeEvent.getValue()` method anywhere in the `attributeReplaced()` method of a class implementing ServletRequestAttributeListener.
C. Use the `ServletRequest.getAttribute()` method anywhere in filter code following the replacement.
D. Use the following code in a class implementing ServletRequestAttributeListener:

```
01 public void attributeReplaced(ServletRequestAttributeEvent event) {
02   String name = event.getName();
03   Object newValue = event.getServletRequest().getAttribute(name);
04 }
```

E. Use the `ServletRequestAttributeEvent.getValue()` method anywhere in the `attributeUpdated()` method of a class implementing ServletRequestAttributeListener.

## Session Listeners

16. The code below is from a class implementing the HttpSessionListener interface (you can assume that the whole class compiles successfully). What will happen when the class is deployed in a web application and servlet code requests a session? (Choose one.)

```
public void sessionInitialized(HttpSessionEvent event) {
  System.out.println("Session Initialized...");
  HttpSession session = event.getSession();
  Boolean loginOK = (Boolean) session.getAttribute("login");
  if (loginOK == null || !loginOK.booleanValue()) {
      session.invalidate();
  }
}
```

A. A runtime exception.
B. Session will be invalidated dependent on the "login" attribute.
C. Session will always be invalidated.
D. Can't determine what will happen.

The code below shows code for the class MySessionAttribute (Listing A). An instance of this class is attached to an HttpSession (Listing B). From the list below, pick out the things that will happen when this session is migrated from a source JVM to a target JVM. (Choose four.)

**LISTING A**

```
import java.io.*;
import javax.servlet.http.*;
public class MySessionAttribute implements
  HttpSessionActivationListener, Serializable {
  private static String data;
  public String getData() { return data; }
  public void setData(String newData) {
      data = newData;
  }
  public void sessionWillPassivate(HttpSessionEvent arg0) {
      System.out.println(data);
  }
  public void sessionDidActivate(HttpSessionEvent arg0) {
      System.out.println(data);
  }
}
```

**LISTING B**

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

```
public class TestMySessionAttribute extends HttpServlet {
  protected void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
        HttpSession session = request.getSession();
        MySessionAttribute msa = new MySessionAttribute();
        msa.setData("My Data");
    }
}
```

A. `sessionWillPassivate()` method called in the source JVM
B. `sessionWillPassivate()` method called in the target JVM
C. `sessionDidActivate()` method called in the source JVM
D. `sessionDidActivate()` method called in the target JVM
E. "My data" written to the source JVM's web server console
F. "My data" not written to the source JVM's web server console
G. "My data" written to the target JVM's web server console
H. "My data" not written to the source JVM's web server console

18. (drag-and-drop question) Match the lettered missing pieces of code with choices from the numbered list. The choices may be used more than once.

```
import [      A      ].*;
public class SessionAttrListener implements
HttpSession[      B      ] {
public void
attributeAdded([      C      ] event)
{
    String name = [      D      ];
    String value = "" + [      E      ];
  }
public void
attributeRemoved([      F      ]
event) {
}
public void
attributeReplaced([      G      ]
event) {
  HttpSession session = event.[      H      ];
  String newValue = "" +
session.[      I      ](name);
  }
}
```

| | |
|---|---|
| 1 | event.getValue() |
| 2 | getAttribute |
| 3 | getValue |
| 4 | HttpSessionAttributeEvent |
| 5 | AttrListener |
| 6 | event.getName() |
| 7 | getSession() |
| 8 | javax.servlet.http |
| 9 | HttpSession |
| 10 | HttpSesssionBindingEvent |
| 11 | AttributeListener |
| 12 | javax.servlet |
| 13 | getSessionID() |
| 14 | Listener |
| 15 | getHttpSession() |

19. Pick out true statements from the list below. (Choose two.)

    A. Classes implementing HttpSessionBindingListener must be declared in the deployment descriptor.

    B. More than one session listener interface may take effect from the same deployment descriptor declaration.

    C. HttpSessionAttributeEvent is a parameter for methods on more than one of the session listener interfaces.

    D. A single class cannot implement both the interfaces ServletRequestAttributeListener and HttpSessionAttributeListener.

    E. `sessionDidPassivate()` is one of the methods of the HttpSessionActivationListener interface.

    F. An HttpSessionListener's `sessionDestroyed()` method will be called as a result of a client refusing to join a session.

20. A web application houses an HttpSessionAttributeListener and an object (SessionAttrObject) that implements HttpSessionBindingListener. Pick out the correct sequence of listener method calls that follows from executing l the servlet code below inside this web application. (Choose one.)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Question20 extends HttpServlet {
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    HttpSession session = request.getSession();
    session.invalidate();
    HttpSession newSession = request.getSession();
    SessionAttrObject boundObject = new SessionAttrObject("value");
    newSession.setAttribute("name", boundObject);
    newSession.setAttribute("name", "value");
    newSession.setAttribute("name", null);
  }
}
```

    A. `attributeAdded(), valueBound(), attributeRemoved(), valueUnbound(), attributeReplaced(), attributeRemoved()`

    B. `valueBound(), attributeAdded(), valueUnbound(), attributeRemoved(), attributeReplaced(), attributeRemoved()`

C. `valueBound(), attributeAdded(), valueUnbound(), attributeReplaced(), attributeRemoved()`

D. `attributeAdded(), valueBound(), attributeReplaced(), valueUnbound(), attributeReplaced()`

E. `valueBound(), attributeAdded(), valueUnbound(), attributeRemoved(), valueBound(), attributeAdded(), valueUnbound(), attributeRemoved()`

F. None of the above.

# LAB QUESTION

Let's use the skills you have learned in this chapter for a practical purpose—well, nearly practical! You're going to design a memory game. Someone setting up the game will use your application to type in ten words. These are presented back in the order in which they were entered.

The person playing the game signs on to a fresh browser session. She has an allotted number of seconds to remember the words and their order. She presses a button to begin the game. After she tries her best guess at the words, the application tells her how well she has done.

Some technical suggestions on achieving this: The words put in by the person setting up the game might be transferred from request parameters to session attributes—or a single session attribute whose value is a collection class, maintaining the order of the words entered. When the setup is done, transfer the session attribute(s) to context attribute(s). Make this happen as a consequence of invalidating the session, doing the transfer in an HttpSessionListener's `sessionDestroyed()` method. When playing the game, the request parameters (for the guesses) can again be transferred to session attributes. On conclusion of the game, compare the session attributes with the context attributes to come up with a mark.

Feel free to add as much needless complexity as you like. For example, you could pass each request parameter to a request attribute, listen for the removal of the request attribute, and transfer the word to a session attribute. This is just to explore the capacities of listener classes in preparation for the exam—in your real development life, keep it simple!

# SELF TEST ANSWERS

## Session Life Cycle

1. ☑ **A** matches to output **2**. The session is new, and because it's newly established by the web container, the `isRequestedSessionId*` methods both return false. **B** matches to output **2** also; if the session is refused by the client, it is still new—and the session ID can't come from a URL or cookie (it hasn't been returned by the client). **C** matches to output **6**; the session is not new (it's the second request). Because the ID has been returned with a cookie, the session is not from a URL and is from a cookie—so false, false, true. **D** matches to output **5**; the session is again not new (false output), and the session has come from a URL (true output) and not a cookie (false output).
   ☒ All other combinations are wrong, as shown by the reasoning for the correct answers.
2. ☑ **A** is the correct answer: The code will not compile. The parameters of the `doGet` method are defined as types ServletRequest and ServletResponse—not HttpServletRequest and HttpServletResponse. Hence, when at line 16 the code tries to get a session from the request, there's a compiler error saying that ServletRequest has no such method (all the session infrastructure is part of the javax.servlet.http package).
   ☒ **B** is incorrect but sounds plausible; however, even if the compiler error were corrected, the NullPointerException would occur at line 17 (when the code might try to use a null session reference). **C** is incorrect but would be true if the compiler error were corrected: An Illegal StateException would result from trying to use `setAttribute()` on an invalidated session. **D** is incorrect because we never get as far as any runtime exception. **E** is excluded because **A** fits the bill here.
3. ☑ **A** and **F** are the correct answers. A call to `HttpServletRequest.getSession()` with no parameters is equivalent to a call to `HttpServletRequest.getSession(true)`. Both calls will create a session if none exists already.
   ☒ **B**, **C**, **E**, and **H** are incorrect and can be dismissed straightaway, because the method for retrieving sessions is associated with HttpServletRequest, not ServletRequest. **D** is incorrect—a call to `HttpServletRequest.getSession(false)` is the odd one out—and under these circumstances, a session will not be created; the method obtains a session only if there is one already. **G** is incorrect because `HttpServletRequest.getSession()` does not accept a String parameter.
4. ☑ **B** and **D** are the correct answers. In distributed applications, session objects are cloned across JVMs. And sessions can be set to never time out, using `HttpSession.setMaxInactive`

`Interval()` with a negative parameter or using a zero or negative number as the value for the `<session-timeout>` element in the deployment descriptor.

☒ **A** is incorrect: Session objects are scoped to a single web application. **C** is incorrect because you can use code to explicitly invalidate a session as well as allowing its destruction after a predefined period of inactivity. **E** is incorrect: There is no deployment descriptor mechanism to specify that sessions will end after a set number of requests (you could, of course, write servlet code to achieve this end).

5. ☑ **A** and **C** are the correct answers. **A** is a correctly constructed part of the deployment descriptor file, and it correctly specifies a time-out of 1 minute. **C** is also correct. Because the filter has a mapping of "/," it will receive every request to the web application. The code correctly obtains the session from the request parameter and uses the right method—`setMaxInactive Interval`—using the correct number of seconds, 60.

☒ **B** is incorrect: The deployment descriptor is correctly constructed, but the value for `<session-timeout>` is specified in minutes (not seconds), and the value of 60 represents a whole hour. **D** is incorrect on two counts. Although the code is correct, the only sessions affected will be those invoking this servlet (nowhere are we told that it has a generic mapping). Also, a value of 1 passed to `setMaxInactiveInterval` represents just 1 second for time-out. **E** is incorrect because the `init()` method called on the startup of a web application does not receive an HttpServletRequest as a parameter. True, you could construct an overloaded `init()` method that did this—bizarre as this would be—but the answer would still be wrong, for only the sessions involved in calls to this servlet would be affected.

## Session Management

6. ☑ **C** is the correct answer. Cookies are used by default for session management.

☒ **A** is incorrect: URL rewriting is a substitute mechanism when cookies are disallowed. **B** is incorrect: You can do your own session management with hidden form fields, but it's not a standard mechanism supported by J2EE web containers (let alone the default). **D** is incorrect: SSL does have its own session management features, but it is used only for secure transactions. **E** is a complete red herring: jsessionId looks like something used in URL rewriting (yet it is in the wrong case), but is not strictly speaking a request parameter and is in any case only part of the mechanism.

7. ☑ **B** and **E** are the correct answers. *jsessionid* is the unique identifier for the session ID passed in URL rewriting—and it must be all lowercase, as shown. It's also true that you should use different methods to rewrite URLs for links versus URLs for redirection (`HttpServletResponse .encodeURL()` vs. `HttpServletResponse.encodeRedirectURL()`).

☒ **A** is incorrect: You could write a mechanism as described, but it would not be the usual way of managing sessions. **C** is incorrect: The cookie used for default session management must be called JSESSIONID (exactly that, all capitals). **D** is incorrect: If you are using the default session management mechanism, the web container adds the JSESSIONID cookie automatically to the response—you don't need to explicitly code for it.

8. ☑ **A** matches to **4**, and **B** matches to **6**: HttpServletRequest and HttpServletResponse are the appropriate parameter types for the `doGet()` method. **C** matches to **1**: It's an HttpSession type the code needs. **D** matches to **5**: the request method that obtains an HttpSession is `getSession()` (not any variant). **E** matches to **14**: The simple HttpSession method `getId()` returns the unique session identifier. **F** matches to **8**: JSESSIONID in capitals is the name of the cookie for session management. **G** matches to **9**: `isRequestedSessionIdFromURL()` is the correctly named method (not "Url" in mixed case). **H** matches to **10**: *jsessionid* is the URL element that names the unique session ID when URL rewriting is used for session management.
☒ All other matches are red herrings, based on the correct choices above.

9. ☑ **C** is correct. The session ID is encoded in URL1 (with correct syntax), but the session ID is not encoded in URL2. Because URL2 is clearly located in a different context from URL1, then it's not appropriate for the `encodeURL()` logic to attach the session ID. Sessions do not cross contexts (i.e., they don't span different web applications).
☒ **A** is incorrect because JSESSIONID is the name reserved for session management cookies; it should be jsessionid in URL rewriting, all lowercase. **B** is incorrect because the wrong URL has been rewritten (URL2 instead of URL1)—and also, *jsessionid* should be separated from the main part of the URL with a semicolon, not (as here) a question mark, which denotes the beginning of the query string. **D** is incorrect: Although syntactically OK, the session number is attached to URL2 (and the correct answer explains why this is wrong). Finally, **E** is incorrect because JSESSIONID is in capitals, and again a question mark is used where a semicolon should be.

10. ☑ **D** and **E** are the correct answers. **D** correctly states that it's the client that determines whether cookies are allowed or not: Because you may not have control over all the clients using your web application, it's always a good idea to rewrite URLs as a fallback. **E** is also correct: Static pages won't disrupt cookie management during sessions, but they will disrupt a URL-rewriting approach (static pages can't possibly contain a just-generated session ID in their links).
☒ **A** is incorrect: Although there isn't, strictly speaking, a need to rewrite URLs if cookies are used for session management, it's not true to say that this is determined by the server—it's the client's choice to accept or reject cookies. **B** is incorrect because there is a separate method (`HttpServletResponse.encodeRedirectURL()`) for URLs rewritten for redirection. **C** is in-

correct: Cookies are the default session support mechanism, and they must always be supported by J2EE-compliant web containers.

### Request and Context Listeners

11. ☑ **B** and **D** are the correct answers, for neither suggestion will fix the potential problem. The issue is that by the time the `requestDestroyed()` method has been reached, the response has been committed. At this point, it's illegal to attempt to create a new session (an IllegalState Exception is thrown)—but still OK to get hold of a session that exists. The method call at line 04—`request.getSession()`—will obtain a session if it already exists (no problem), but will also attempt to create a session if none exists already (which is the problem). Hence, **D** is an incorrect suggestion, for there will be a problem with the code in some circumstances. **B** might go a long way to solving the problem (ensuring that all servlets in your application obtain a session). But if the request is for some other type of resource (a static HTML page, for example), the request listener will still kick in, so the solution doesn't cover all circumstances.
☒ **A** is a correct suggestion, hence an incorrect answer. By creating a session in the corresponding `requestInitialized()` method, there will definitely be a session to obtain in the `requestDestroyed()` method. **C** will also fix the problem (hence is an incorrect answer) by explicitly passing **false** to the `getSession()` method: A session will be returned only if one exists already. The potential NullPointerException on `session.setAttribute()` is avoided by testing the session reference returned. **E** will also work, by trapping the potential IllegalState Exception.

12. ☑ **E** is the correct answer. In fact, there will be two compilation errors: The context methods `addAttribute()` and `replaceAttribute()` do not exist. You use the method `setAttribute()` for adding and replacing attributes.
☒ **A** is incorrect, for the code never runs (though if the compilation errors were corrected, this would be a true statement). **B** is incorrect—apart from the code not running, it's perfectly OK to remove the same attribute name as many times as you like. **C** is incorrect—were the code to be corrected and run, even, the second `removeAttribute()` call would not cause a method call to a listener (as the attribute has already gone). **D** is incorrect: It's inconceivable that you wouldn't get a context, anyway.

13. ☑ **A**, **C**, and **F** are the correct answers. **A** is counterintuitive, but you can indeed add context attributes (or replace or remove them) in the `ServletContextListener.contextDestroyed()` method—however pointless this may seem. **C** is correct—you have access to the current session via the current request, which is available from the event object passed as a parameter to ServletRequestListener interface methods. **F** is also correct—if you change a

request attribute in the `ServletRequestAttributeListener.attributeReplaced()` method, this will itself cause a call to that same method again—so you have the potential for a perpetual loop (or, more accurately, a StackOverflowError).

☒ **B** is incorrect: The last listener methods potentially called are those in those classes implementing ServletContextAttributeListener. The web container removes attributes from the context after the `ServletContextListener.contextDestroyed()` method has completed, which may cause calls to `ServletContextAttributeListener.attributeRemoved()`. **D** is incorrect: You can access the context only in a context listener (not the request or session). **E** is incorrect: ServletContextAttributeEvent extends ServletContextEvent—which in turn extends java.util.EventObject.

14. ☑ **D** is the correct answer: There are two compilation errors. Both have to do with casting. In line 02, the `getServletRequest()` method returns a ServletRequest object. In an HTTP environment (i.e., most of the time!), this is safe to cast to an HttpServletRequest—which is what's required here. In line 07, the `getValue()` method returns an Object, not a String. If you know the attribute value is a String, then it's safe to cast to String here.

☒ **A**, **B**, **C**, and **E** are incorrect because of the reasoning in the correct answer.

15. ☑ **B** and **E** are the correct answers, for neither approach will get the new value. **E** is an invented method (`attributeUpdated()`)—you can define such a method in a listener class, but the web container framework won't call it! **B** is a good approach—but use it for getting the *old* value of the attribute, not the new one.

☒ **A** is incorrect; it's a perfectly standard way to get hold of the current attribute value. **C** is incorrect for the same reason. **D** is incorrect; although it's a more convoluted way, you will get hold of the new value that has just been added.

## Session Listeners

16. ☑ **D** is the correct answer: You can't determine from this code what will happen. The nasty trick here is that the method shown—`sessionInitalized()`—is not one defined in the HttpSessionListener interface. Sure, you can define such a method in a class implementing the interface, but the method is never called by the web container. The method that IS called on the creation of a session is the `sessionCreated()` method.

☒ **A** is incorrect because the code never gets to run—at least not automatically on creation of a session. Also, there is nothing in the code likely to cause an exception. **B** and **C** are incorrect for the same reason—though had the method actually been the `sessionCreated()` method, **C** would have been the correct answer. Because the `sessionCreated()` method is called as soon as a session is first accessed, there can't have been an opportunity to add any session at-

tributes. This means that there won't be a session attribute named "login," and so the Boolean local variable called "loginOK" will be *null*. According to the logic, this will cause the session to invalidate itself.

17. ☑ **A**, **D**, **E**, and **H** are the correct answers. When the session is about to migrate from the source JVM, any session attribute objects implementing the HttpSessionActivationListener get a call to their `sessionWillPassivate()` method; hence, **A** is correct. When the session has materialized in the target JVM, the migrated session attribute objects have their `sessionDidActivate()` method called (answer **D**). As to why "My data" is written to the source JVM's console (**E**) but not the target JVM's console (**H**), that's because the data is a class variable. Only instance variables are serialized, and so reconstituted in the target JVM. Note that the code in MySessionAttribute employs the dubious practice of returning static data using instance methods.
☒ **B**, **C**, **F**, and **G** are incorrect because of the reasoning you see in the correct answer.

18. ☑ **A** matches to **8**; all the types declared come from the javax.servlet.http package. **B** matches to **11**; given the other information, this can only be an HttpSessionAttributeListener. **C** matches to **10**, as does **F** and **G**: An HttpSessionBindingEvent is passed as parameter to each of the three methods. **D** matches to **6**, and **E** matches to **1**: obvious method names for getting hold of the attribute name and value. **H** matches to **7**: another obvious method name for getting hold of the session. **I** matches to **2** (for the new value, simply get hold of the current attribute from the session).
☒ The remaining answers are eliminated because of the correct answers above. Look out for "faux" method names (such as answer **15**: `getHttpSession()`).

19. ☑ **B** and **F** are the correct answers. **B** is correct because there is no problem with a single class implementing more than one sort of session listener interface—and that single class will only require a single `<listener>` declaration in the deployment descriptor. **F** is correct because a client's refusal to join a session effectively "orphans" the session: The session will then time out according to the usual criteria, at which point `sessionDestroyed()` will be called. However, beware of any suggestions that the client's refusal to join immediately invalidates the session, for that is not necessarily true.
☒ **A** is incorrect because it's not appropriate to declare classes implementing HttpSession BindingListener in the deployment descriptor: Its methods will be called by virtue of having the class as a value object for a session attribute. **C** is incorrect because HttpSessionAttribute Event—while sounding like a logical enough name—is made up. HttpSessionBindingEvent is the type passed to the HttpSessionAttributeListener and HttpSessionBindingListener interface methods. **D** is incorrect: Although ServletRequestAttributeListener and HttpSessionAttribute

Listener have the same trio of methods (`attributeAdded/Replaced/Removed()`), these accept different types as parameters. So a single class can have methods for both request and session just by overloading the methods. **E** is incorrect: The correct name is `sessionWillPassivate()`—which better reflects the exact timing of the method call (i.e., just before the session is serialized).

20. ☑ **C** is the correct answer. First, `valueBound()` is called. This is because you're adding an attribute whose value implements the HttpSessionBindingListener interface. This takes precedence over the `HttpSessionAttributeListener.attributeAdded()` call, which follows immediately afterward. You then change the attribute, replacing the HttpSessionBinding Listener-implementing object; hence, the next call is `valueUnbound()`. Because you're replacing the value of the attribute, next comes an `attributeReplaced()` call. No more calls now to HttpSessionBindingListener methods, for you have added a plain String as the attribute value (and that doesn't implement that interface). However, by nullifying the attribute's value, you remove the attribute—hence the final call to `attributeRemoved()`.
☒ **A**, **B**, **D**, **E**, and **F** are incorrect, following the reasoning above.

# LAB ANSWER

Deploy the WAR file from the CD called lab04.war, in the /sourcecode/chapter04 directory. This contains a sample solution. You can call the initial servlet to start the test setup and taking process with a URL such as http://localhost:8080/lab04/Reset.