1 The Servlet Model

CERTIFICATION OBJECTIVES

- HTTP Methods
- Form Parameters
- Requests
- Responses
- Servlet Life Cycle

✓ Two-Minute Drill

Q&A Self Test

We go to the first topics in the Sun Certification for Web Component Developers. This chapter begins outside the formal world of Java and J2EE, for you're going to need to know something about the primary "inputs" to a web application: HTTP methods. And because the information carried from HTTP methods is mostly carried from web pages, the exam requires you to know a little about HTML syntax, which we'll cover here. Then we'll begin to open up the core exam topics of HTTP requests and responses: how these are decomposed and composed inside a J2EE web application.

CERTIFICATION OBJECTIVE

HTTP Methods (Exam Objective 1.1)

For each of the HTTP Methods (such as GET, POST, HEAD), describe the purpose of the method and the technical characteristics of the HTTP Method protocol, list triggers that might cause a Client (usually a Web browser) to use the method and identify the HttpServlet method that corresponds to the HTTP Method.

Because you are studying for Sun's Web Component exam, it will come as no surprise to you that you need to know something about the main protocol underlying web communication: HTTP. No huge expertise is required. You don't have to be any kind of networking expert or even know what TCP/IP stands for. However, you will need some grasp of the "big seven" HTTP methods—and, in particular, how these relate to J2EE and to the Web. The designers of the exam succeed in targeting just those areas that are also essential for becoming effective in your real-life web application developments.

HTTP

HTTP is a simple request/response protocol. A client—often (but not exclusively) a web browser—sends a request, which consists of an HTTP method and supplementary data. The HTTP server sends back a response—a status code indicating what happened with the request, and (typically) data targeted by the request. Once the request and response have happened, the conversation is over; the client and server don't remain connected. When you use a web browser to fill up your shopping cart, what's mostly happening is a series of lapses in a protracted conversation. This situation makes life much more efficient for the server (it doesn't have to keep its attention focused on you for much of the time); how it manages not to forget your identity is a subject covered in Chapter 4.

An HTTP Request and Response

Let's take a look at an HTTP request at work. There are strict rules about how the request is made up, defined by the World Wide Web consortium in a "Request for Comments" (RFC) document. Actually, the time for commenting on this version of the HTTP standard is long past, so you can regard RFC 2616 (defining HTTP/1.1) as an absolute yardstick.

A request consists of a request line, some request headers, and an (optional) message body. The request line contains three things:

- The HTTP method
- A pointer to the resource requested, in the form of a "URI"
- The version of the HTTP protocol employed in the request

Therefore, a typical request line might look like this:

```
GET http://www.osborne.com/index.html HTTP/1.1
```

A carriage return/line feed concludes the request line. After this come request headers in the form—name: value (the name of the request header, followed by a colon and space, followed by the value). Here are some examples:

```
Accept: image/*, application/vnd.ms-excel, */*
Accept-Language: en-gb
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Host: www.osborne.com
Connection: Keep-Alive
```

We'll explore the meaning of one or two of these headers a bit later. A blank line must follow the last request header and, after that, the request body—if there is one (there doesn't have to be). The request body can contain pretty much anything, from a set of parameters and values attaching to an HTML form, to an entire image file you intend to upload to the target URL.

exam watch

You won't be tested on the valid construction of HTTP messages, though you might find questions that expect you to know the effect of setting certain request headers. We'll meet these in the "Requests" section of this chapter.

Having sent the request, you can expect a response. The construction of the response is very similar to that of the request. Here's an example:

```
HTTP/1.0 200 OK
Connection: Close
Date: Fri, 02 May 2003 15:30:30 GMT
Set-Cookie: PREF=ID=1b4a0990016089fe:LD=en:TM=1051889430:
LM=1051889430:
S=JbQnlaabQb0I0KxZ; expires=Sun, 17-Jan-2038 19:14:07 GMT;
path=/; domain=.google.co.uk
Cache-control: private
Content-Type: text/html
Server: GWS/2.0
[BLANK LINE]
"<html><head><meta HTTP-EQUIV="content-type" CONTENT="text/html;
 charset=UTF-8"><title>Google Search: MIME </title>
 etc. etc. rest of web page
```

The response line also has three parts. First is the HTTP version actually used, reflected back to the client. Next is a response code (200 denotes success; you'll already know 404, page not found). After that is a brief description of the response code. A carriage return/line feed denotes the end of the response line.

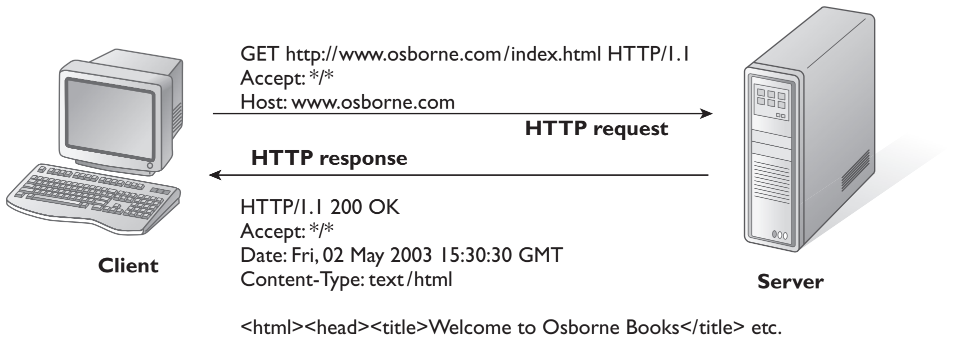
Response header lines follow, similar in format to request headers, although the actual headers themselves are likely to be a bit different. We'll explore some of these later in the chapter. Finally, separated from the headers by a blank line, is the response body. In the example, you can see the beginning of a web page being returned. Figure 1-1 shows the request/response interchange graphically.

The "Big Seven" HTTP Methods

The most fundamental request method is GET, which simply means "I want this resource from this server, so kindly return it to my browser." Whenever you type an address line in your browser and press GO, or click an underlined link on a web page, it's a GET method that is generated in the request behind the covers. However,

FIGURE 1-1

A Request/ Response Interchange


GET http://www.osborne.com/index.html HTTP/1.1
Accept: */*
Host: www.osborne.com
HTTP request
HTTP response
HTTP/1.1 200 OK
Accept: */*
Date: Fri, 02 May 2003 15:30:30 GMT
Content-Type: text/html
<html><head><title>Welcome to Osborne Books</title> etc.
Client
Server

GET is not the only HTTP method. Six others are sanctioned by Internet Standards Bodies: POST, HEAD, OPTIONS, TRACE, PUT, and DELETE—these, together with GET, constitute the "big seven" mentioned in the heading.

There are other pretenders to the ranks of HTTP methods beyond the big seven. CONNECT is officially listed in the RFC, but it is essentially reserved for future use and won't be further discussed in this book. Other methods have some limited support outside the official sanction of the RFC, and these may have full or limited support on some servers. Yet others are defined in other RFCs—for example, RFC2518, which defines HTTP extensions for web authoring (WebDAV). On servers that support this approach, you'll find methods such as PROPFIND, LOCK, and MOVE. However—to come back to basics—methods other than the "big seven" don't impinge on the standard servlet way of doing things and so don't figure on the SCWCD exam.

Let's examine each of the big seven methods in turn.

GET We've already said quite a bit about the GET method. It's meant to be a "read-only" request for information from a server. The server might respond with a static web page, an image, or a media file—or run a servlet and build up a dynamic web page, drawing on database and other information. The three standard triggers for a GET request (in the confines of a standard browser) are

- Typing into the address line of the browser and pressing GO
- Clicking on a link in a web page
- Pressing the submit button in an HTML `<form>` whose method is set to GET

The importance of an HTML `<form>` (this subject gets full treatment in the "Form Parameters" section of the chapter) is the ability to pass parameters along with the request. With a GET requested, these parameters are appended to the URL. Let's look at a true-life example, taken from a well-known dictionary web site (http://www.dictionary.com). On the home page, I type in the word I want to look up ("idempotent") in the one-field form provided, which uses the GET method. Pressing SUBMIT generates and executes the following request on the address line of my browser:

```
http://dictionary.reference.com/search?q=idempotent
```

Everything up to and including the word "search" is the standard URL—I am wanting to run the HTTP method GET for the "search" resource found on the computer hosting the dictionary.reference.com domain. The question mark introduces the query string (in other words, the parameter list). The name of the parameter is *q*, and—separated from the name by the equal sign—is the parameter's value, "idempotent."

POST The next most usual method is POST. This is again triggered from forms in browsers, this time when the form method is itself set to POST. When compared with GET, the main difference with POST is one of intent. Usually, a POSTed form is intended to change something on the target server—add a registration, make a booking, transfer some funds—an action that is likely to result in a database update (or something of equal seriousness). Whereas GET is intended as read-only, POST is for add/update/delete operations.

This discussion leads us to an important word, the one cited in the dictionary example: "idempotent." An idempotent request is meant to have the same result no matter how many times it is executed. So a request to look up a word in a dictionary changes nothing: The request can be executed again and again with the same result. That feature makes it idempotent. However, a request to transfer funds mustn't be repeated casually, for obvious reasons. Such a request can't be classified as idempotent. Each HTTP method is classified as being idempotent or not—and you're probably ahead of me in realizing that GET is supposed to be idempotent, whereas POST most definitely is not. Of course, that's how things are meant to be, but there is no absolute guarantee that any given GET request won't have irreversible side effects. However, the outcome depends on what the server program that receives the GET request actually does with it. Equally, a POST request may not result in an update—though that's generally less serious. However, by and large, GET and POST methods obey idempotency rules, and it goes without saying that your web applications should observe them.

Although as a web surfer you won't typically be aware when you're executing a GET and when a POST, there are some practical as well as philosophical differences.

When there are parameters with a POST request, they are put into the request body, not appended on to the query string of the URL, as for GET. This situation has two benefits:

- A URL is limited in length. The official line of RFC 2616 (the HTTP 1.1 specification) is not to impose a length limit: Servers should be able to handle anything. However, the authors of the specification are realistic enough to declare some caveats, especially with regard to browsers: The specification points out that some older models are limited to 255 characters for URL length. Even at the time of writing, Microsoft's Internet Explorer browser allows only 2,083 characters for the URL—much more generous than 255, but still not limitless. By contrast, the request body is as long as you want it to be (megabytes, if necessary) and is limited in practical terms only by the bandwidth of your Internet connection.
- Putting parameters into the URL is very visible and public and is usually recorded by the "history"-keeping nature of most browsers. It's much more private (though still not wholly secure) to pass parameters in the request body.

Indeed, POST isn't just for passing simple name/value parameter pairs. You can use the POST method to upload whole text or binary files in the request body (in which case, it is acting a lot like the PUT request, which we'll examine a bit later).

HEAD The HEAD method is identical to the GET method except for one important respect: It doesn't return a message body. However, all the response headers should be present just as if a GET had been executed, and these contain a great deal of information about the resource (content length, when the resource was last modified, and the MIME type of the file, to name three of the more straightforward pieces of information available). These data are often called "meta-information." Using the HEAD method is an economical way of checking that a resource is valid and accessible, or that it hasn't been recently updated—if all you're doing is some checking on the state of the resource, why bother to bring the whole thing back to your local machine?

OPTIONS The OPTIONS method is even more minimal than HEAD. Its sole purpose, pretty much, is to tell the requester what HTTP methods can be executed against the URL requested. So, for example, if I target the following URL on the McGraw-Hill web site:

```
http://www.mcgrawhill.com/about/about.html
```

I'm told I can target the GET and HEAD options for this web page. This seems fair—there's certainly no reason to permit the execution of more dangerous methods (such as PUT or DELETE) on this URL. As an aside—the information on allowed methods is squirreled away in the values of one of the *response header* fields, whose key is Allow. This subject will be covered in the "Responses" section of this chapter.

PUT The object of the PUT method is to take a client resource (typically, a file) and put it in a location on a server as specified in the URL of the request. If there's anything already on the server in that location, then tough luck—a PUT will obliterate it, overwriting the current contents of the URL with the file it is uploading. You can determine from the response codes you get back (and more on these later) whether the resource was replaced (typically, response code 200) or created for the first time (response code 201).

I mentioned earlier that a POST can do the same work of uploading a file, and many other things besides. What is the difference between POST and PUT in this capacity? It's the fact that PUT works directly on the URL given in the request. PUT is not supposed to do anything clever, such as put the uploaded file in the request somewhere different. However, that is the prerogative of POST. The object URL of a POST method is usually a clever program (in our case, probably a Java servlet), which can do whatever it pleases with the uploaded resource, including putting it in some other location. Indeed, the program is almost certain to put uploaded files in another location, for to put the file in its own URL slot would mean overwriting itself! So, in summary, a PUT does direct file replacement (and so is rather like using FTP), but a POST can be much more subtle.

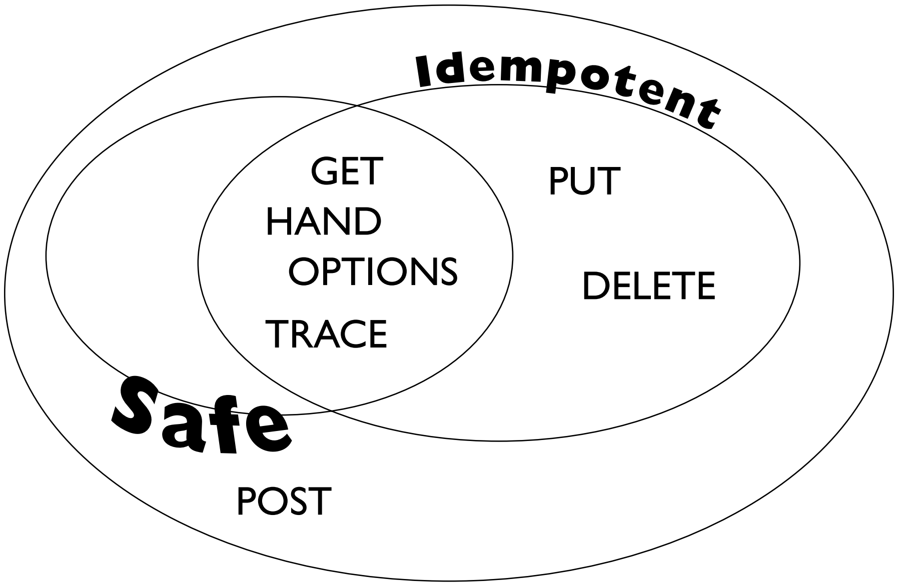
DELETE DELETE is the direct counterpart of PUT: It causes the server to delete the contents of the target URL—if not permanently, then by moving the resource there to an inaccessible location. A server has the right to delay its response to a DELETE method, as long as it responds later. A response code of 200 (OK) indicates that the deed has been done; 202 (accepted) means that the request has been accepted and will be acted on later.

TRACE Finally, the TRACE method exists because a request over HTTP is unlikely to go directly to the target host machine that actually holds the resource you require. The nature of the Internet is to pass a request from computer to computer in a long chain. Some request headers may be rewritten in transit along this chain. The purpose of TRACE is to return the request back to the requester in the state it was in at the point where it reached the last computer in the chain. As you might guess, the primary reason for doing so is to debug some problem that you attribute to request header change.

Idempotency and Safety

Let's return to the idea of idempotency, which we first came across on contrasting the scope of the GET and POST methods. We saw that GET requests should be idempotent and that POST methods are not. The other concept to introduce you to is safety. GET, TRACE, OPTIONS, and HEAD should leave nothing changed, and so are safe. Even if a GET request has irreversible side effects, a user should not be held responsible for them. Therefore, from a user perspective, the methods *are* safe. PUT, DELETE, and POST are inherently not safe: They do cause changes, and a user can be held accountable for executing these methods against the server.

What about idempotency? The safe methods are inherently idempotent because they don't (or shouldn't) change anything that would change the results when running the method again. Surprisingly, PUT is considered idempotent as well—because even if you run the same PUT request (uploading the same file to the same URL) repeatedly, the net result is always the same. The same reasoning applies to DELETE. The following illustration shows the methods grouped according to safety and idempotency.



Most web browsers don't have the capability of executing all of the "big seven" methods. Most limit their execution to GET and POST, as directed by the web page currently loaded. However, you'll meet a browser in the first exercise at the end of this section that—although cosmetically challenged—gives you the opportunity to try out all seven methods.

What Does This Discussion Have to Do with Java?

Now that you have a good grasp of HTTP methods, the question is "What does any of this have to do with Enterprise Java?" There is clearly a diverse world of web servers out there that understand HTTP methods and respond to them. For its part, Enterprise Java defines the concept of the *servlet*—a Java program written to a strict (but small) set of standards that can deal with HTTP (and other) requests and send back dynamic responses to client browsers.

Normal web servers are good at dealing with static document requests—web pages, images, and sundry other files that don't change. (Their authors may update the pages from time to time, but the point is that the pages don't change or get created in mid-request.) Web servers didn't take very long to develop dynamic features—capabilities to run programs or server-side scripts on receipt of an HTTP request—rather than simply returning a static document. These techniques frequently came under the heading of "common gateway interface" (CGI).

CGI has been very useful, but not without problems, the greatest of these being performance. The original CGI model demanded a heavyweight process on the web-serving host machine for each request made. Furthermore, this process would die once the request was completed. This was not a sustainable or scalable approach for web applications of any size, as the startup and shutdown of a process are heavy operations for most machines, often outweighing the "grunt" needed to get the real work done! Consequently, better models quickly followed. These included FastCGI, which has the benefit of keeping processes alive—usually, one for each different server-side program that your server supports. And there are many other alternatives—Active Server Pages (ASP), PHP, mod_Perl, and ColdFusion—to name some main players. All vary in their characteristics, such as which web servers support them and how efficient they are.

It's beyond the scope of this book to do any more than take this cursory glance at the differing technologies. After all, we're here to learn the Enterprise Java way of "doing CGI." So I will stop short of claiming that Java is better than the alternatives listed. I'll simply say that it offers a simple and elegant way of getting the job done and that it has other benefits. The structure of a Java web server is shown in

FIGURE 1-2

A Java Web Server

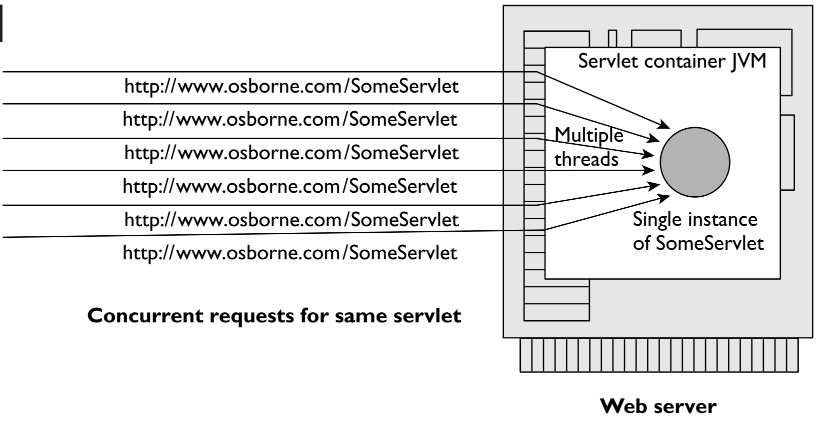

http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
Concurrent requests for same servlet
Servlet container JVM
Multiple threads
Single instance of SomeServlet
Web server

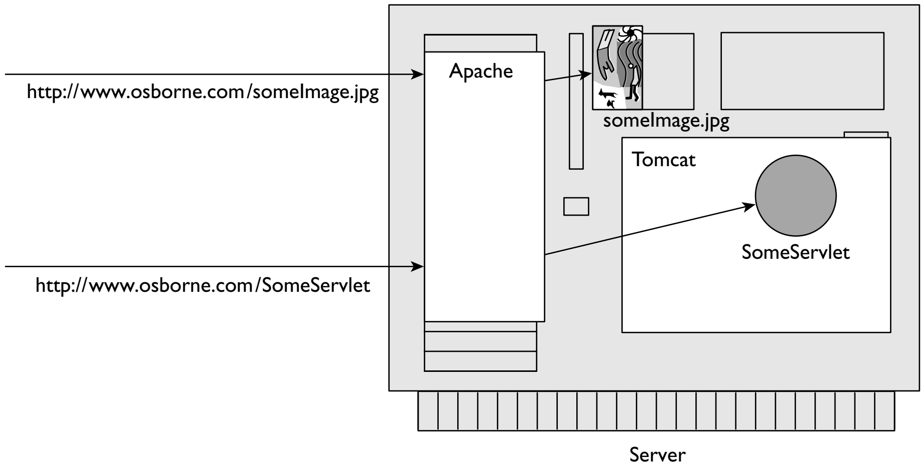
Figure 1-2. You can see that the web server contains a running JVM—a single running process on the host machine. Because Java is a multithreaded language, many threads can be set running within a single process. Many servlet-servicing threads run side by side in a pool. If the request is for a servlet, it doesn't matter which thread is selected—any thread can run any servlet (as well as doing normal web server stuff, such as returning static HTML files). It's not like FastCGI, in which a process is tied to one particular program.

Of course, a web server designed to support servlets is written to stringent specifications in order to provide support for the many things we describe in this book. Tomcat is one such server—an open source Java web server from the Apache Jakarta Project (http://jakarta.apache.org/tomcat). Tomcat is described as the official reference implementation for Java Servlet and JavaServer Page technologies, so it's the one I encourage you to use in the practical exercises and labs throughout this book—you can pretty much guarantee it will do exactly what the specifications say it should do. From the point of view of SCWCD exam readiness, this server is exactly what you require—it's easy to believe something about servlets as true that turns out to be a quirk of a server that doesn't quite support the specification as it should.

on the job

Tomcat is perfectly capable of working as a stand-alone web server that happens to run servlets. That's the mode we'll use it in throughout the practical exercises in this book. However, Tomcat doesn't really set itself up to be a "one-stop shop" for all your web serving needs. It describes itself as a* servlet and JSP container*—that is, a piece of software for running servlets (and

JavaServer pages). The most usual arrangement in a production environment is to have an industrial-strength web server separate from the servlet container. A typical open source combination is the Apache HTTP server (http://httpd.apache.org/), the most popular web server on the Internet, with Tomcat as the servlet container plug-in. The Apache HTTP server is great for serving up static content (such as images from image libraries), as well as being highly configurable for security and robustness. Whenever it encounters a request for a servlet that it cannot deal with, it hands this on to the Tomcat servlet container. There are also many commercial combinations of web servers and servlet containers, such as IBM's HTTP Server together with IBM's WebSphere Application Server. The following illustration shows how a web server and a servlet container cooperate.


http://www.osborne.com/someImage.jpg
http://www.osborne.com/SomeServlet
Apache
someImage.jpg
Tomcat
SomeServlet
Server

We return to servlets. What happens when a request for a servlet reaches the Tomcat servlet container? That's easy. If the HTTP method is GET, the servlet container will attempt to run a method called `doGet()` in the target servlet. If POST, then the method is `doPost()`. OPTIONS prompts `doOptions()`—it's an easy pattern to grasp.

We'll get to actually writing servlets a bit later in the chapter. Suffice to say two things for now:

1. You typically extend the existing javax.servlet.http.HttpServlet class.
2. You override at least one of the methods we've described: almost always `doGet()`, `doPost()`, or both, and sometimes `doPut()` or `doDelete()`.

If your servlet is hit by an OPTIONS, TRACE, or HEAD request, the existing implementation of the `doOptions()`, `doTrace()`, and `doHead()` methods in HttpServlet should suffice, so you rarely if ever override these methods.

All of these methods receive two parameters—Java objects that wrapper up the HTTP request and response. These are of type javax.servlet.HttpServletRequest and javax.servlet.HttpServletResponse, respectively. These make it easy (in Java terms) to extract information from the request and write content to the response. No low-level knowledge about formatting is required. After doing an exercise on the principles learned in this section of the book, we'll spend the next two sections learning about these two fundamental servlet classes.

EXERCISE 1-1

ON THE CD

Custom Browser for Learning HTTP Methods

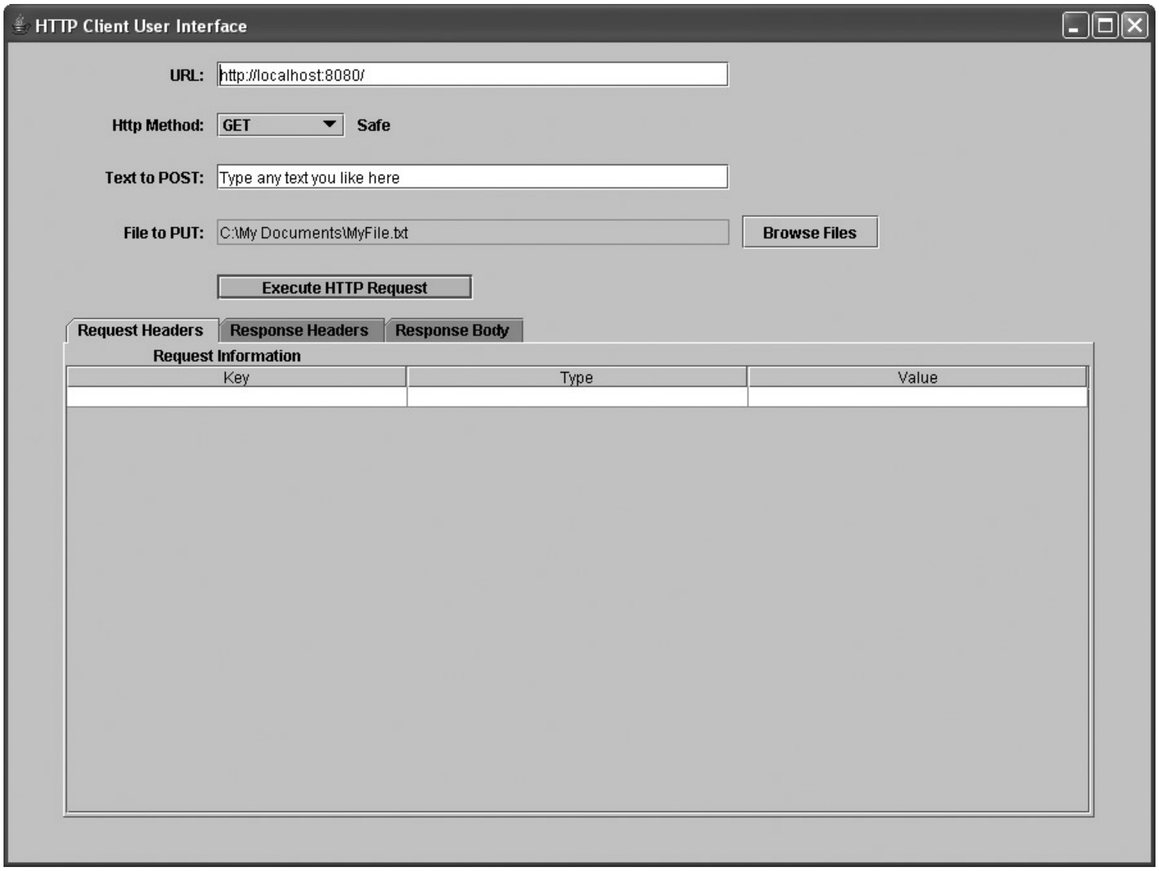
This exercise uses a browser specifically designed to show the use of the various HTTP methods. Unlike the common full-fledged models (Netscape, Mozilla, Internet Explorer), which render web pages prettily and shield you from so much, my browser exposes you to the dirty underbelly of HTTP.

You won't write any code in this exercise. However, you will deploy a web application (even though you won't officially learn about this until Chapter 2). You'll be doing this a lot through the various exercises—the instructions are in Appendix B. In this case, the web application file is called ex0101.war, and you'll find it on the accompanying CD in directory sourcecode/ch01. The instructions in the appendix also take you through starting up Tomcat, the Java-aware web server that underpins the exercises in this book.

Using the Custom Browser for Learning HTTP Methods

1. Start up a command prompt.
2. Change to directory <TOMCAT INSTALLATION DIRECTORY>/webapps/ex0101/WEB-INF/classes.
3. Execute the command: `java uk.co.jbridge.httpclient.User Interface.` If you get a rude message (such as "java" not recognized as a command), then you need to ensure that the J2SDK commands (in <J2SDK INSTALLATION DIRECTORY>/bin) are in your system path.

4. If successful, you'll see a graphical user interface as illustrated below. There's an area at the top where you can type in a URL—equivalent to the address line in popular browsers. Beneath this is a drop-down list where you can select the HTTP method to execute. Beneath this is a text field where you can type some text to associate with an HTTP POST request. Under that is a display area—by using the browser button to the right of this, you can select a file from your file system and then execute the HTTP PUT method to upload the file to the area of your web server's directory structure targeted by the URL. The button underneath this—Execute HTTP Request—will do exactly that when pressed, using the parameters as you have set them above. The area beneath the button is tabbed, and it displays the result of execut-



URL: http://localhost:8080/

Http Method: GET Safe

Text to POST: Type any text you like here

File to PUT: C:\My Documents\MyFile.txt

Request Information

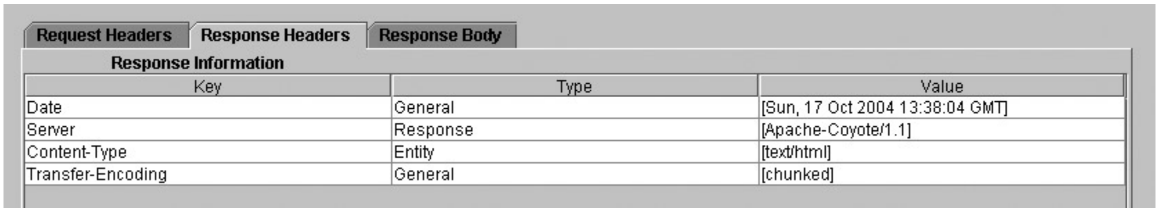
Key	Type	Value

ing the HTTP request. The first tab—Request Headers—shows information sent from the browser to the server (we learn about request headers in the next section). The second tab—Response Headers—shows information sent back from the server to the browser (we explore valid values for these later in the chapter). The third tab shows the response body—the resource retrieved from the server. If this is a web page, it is displayed as the underlying HTML.

5. Note that what you've done here is to start a plain old Java application—using the **java** command on a class with a `main()` method. As it happens, the code resides in the same place as the code for the web application it will communicate with (in a few steps), but that's just a coincidence. If you prefer, place all the classes from the package uk.co.jbridge.httpclient in a completely separate location on your machine and run the browser from there instead.

Executing Safe Methods (GET, HEAD, OPTIONS, TRACE)

6. In the browser, change the text in the URL field to say `http://localhost:8080/ex0101/index.jsp`. Leave the HTTP method as the default GET, and press the Execute HTTP Method button. This will invoke a very simple web page. Take a look at the request headers. The request headers are set up within the user interface program. "User-Agent" describes the type of client—usually one that derives from the Mozilla browser. "Accept-Language" indicates the (human) languages the client would prefer to see in any requested web pages. "Accept" indicates the file types the client can deal with, using standard MIME abbreviations (you learn more MIME in Chapter 2).
7. Now take a look at the response headers. These may vary depending on your server setup—the accompanying illustration shows you what I see.



Response Information

Key	Type	Value
Date	General	[Sun, 17 Oct 2004 13:38:04 GMT]
Server	Response	[Apache-Coyote/1.1]
Content-Type	Entity	[text/html]
Transfer-Encoding	General	[chunked]

`Date` reflects the date the requested resource was last changed. `Server` shows some version and type information about the web server software, in this case Tomcat. `Content-Type` shows the MIME type of the information returned—in this case, plain HTML text. Finally, `Transfer-Encoding` shows how the response message is parceled up—"chunked"—to help the browser interpret the contents.

8. Repeat the above exercise for the other safe HTTP methods. The browser gives a clear indication about which methods are safe and which are not. Try the safe methods on a range of other public URLs (e.g., http://www.osborne .com—you must type in the complete address). Note the different response headers you get back.

Executing Unsafe Methods (POST, PUT, DELETE)

9. The unsafe methods are POST, PUT, and DELETE. Nearly any browser will POST. This browser is set up to POST a line of text, which you can type into a field. The exercise comes with a corresponding servlet that will accumulate all the lines you type into this field and then display them on a web page.
10. Enter `http://localhost:8080/ex0101/PostServlet` in the browser's URL field. Change the HTTP method to POST, and type any text you like in the Text to POST field. Click the Execute HTTP Request button (which should have turned red to indicate an unsafe method). Click the Response Body tab—you should see some simple HTML with the text you typed in the middle.
11. Write some different text in the Text to POST field, and click the Execute HTTP Request button again. This time, in the Response Body tab you should see both the original text you typed, plus the latest text. (To avoid losing text between HTTP requests, the text is appended to a file called postData.txt—you can find this in the ex0101 context directory for the web application.)
12. Now we'll try using the browser to PUT a file in your web application directory structure. Change the URL field to say `http://localhost:8080/ex0101/myFile.txt`. Change the HTTP method to PUT. In the File to PUT field, use the Browse… button to select any text (.txt) file on your file system (preferably outside of any directory structure to do with your web applications). Now click the Execute HTTP Request button. Check in the web application subdirectory (ex0101)—you should find a file called myFile .txt, whose contents should be the same as the file you selected as the "file to PUT."
13. The above attempt to PUT a file may fail under Tomcat with a 403 response code: forbidden. By default, Tomcat blocks HTTP methods (such as PUT and DELETE) that alter the structure of a web application. Under these circum-

stances, you will need to find the file web.xml in <TOMCAT-INSTALL-DIRECTORY>/conf. Edit this with a text editor, and insert the lines in bold below:

```
<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.DefaultServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>readonly</param-name>
    <param-value>false</param-value>
  </init-param>
  <init-param>
    <param-name>listings</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

What you have done here is to add an initialization parameter to a servlet, which usefully anticipates the work we'll do in Chapter 2. In this case, you've added an initialization parameter to one of Tomcat's own servlets—the default servlet—which is used to process requests when the URL requested is not otherwise mapped to a resource in a web application. By setting *readonly* to *false*, you are saying that the PUT and DELETE methods will be permitted. You must *restart the Tomcat server* after making this change. If you're using another server to perform the exercises in this book, you're on your own, but be aware that many servers block PUT and DELETE methods by default, and you may have to do some digging in your server documentation to find out how to enable them.

14. Now try using the browser to delete the file you have just PUT in your web application. Leave the URL as is (`http://localhost:8080/ex0101/myFile.txt`), but change the HTTP method to DELETE. Click EXECUTE HTTP REQUEST. You may get a strange response code (204: No Content), but you should find if you look in the web application directory structure that the file has disappeared.

15. That concludes the exercise. Bask in the comfortable knowledge that you have exercised more HTTP methods than most people get to do in a career of working with web servers!

CERTIFICATION OBJECTIVE

Form Parameters (Exam Objective 1.2)

Using the HttpServletRequest interface, write code to retrieve HTML form parameters from the request.

Now that you have seen the connection working between a client and a servlet, we can turn our attention to the Java code. We're going to explore one of the most fundamental aspects: getting information from an HTTP request and using that within our servlet. It's possible to attach parameters to an HTTP request as a series of name/value pairs. These travel in the request body or the URL—it's even possible to type parameter names and values directly into a browser's address line (if you know what you're doing!). More usually, though, information destined for a request is collected in an HTML form embedded within a web page. The user types in data, and the browser sorts out how to format the HTTP request appropriately so that the parameters are included. Consequently, we'll need to explore HTML form construction—for the exam, you'll be expected to understand HTML well enough to predict the parameters that will arise from a given web page. Then we'll see the range of APIs for teasing out the parameters from the request on arrival in the servlet.

HTML Forms

HTML forms are very easy to construct and use. You use the `<form>` tag and place the screen controls you need before the closing `</form>` tag. That said, a range of graphical components (often called form controls, sometimes called "widgets") are supported within a web page, and you need to understand them all. These are mainly provided to allow for the user to input data and then to submit these data to the server in the form of parameter name/value pairs. Most form controls are defined using the `<input>` tag with a variety of attributes, but you'll need to understand the `<select>` and `<textarea>` tags as well.

The Form Itself (`<form>`)

A form is defined on a web page starting with the opening tag `<form>` and ending with the closing tag `</form>`. The opening tag has a number of attributes, but only two of them have real significance to web application operation. A typical form opening tag might look like this:

```
<form action="someServlet" method="POST">
```

Most browsers tolerate wide syntactic variety—you'll probably find that

```
<form action=someServlet method=POST>
```

or

```
<FORM ACTION = 'someServlet' METHOD = 'POST'>
```

works just as well. The essentials are that

- The tag begins with "<" and finishes with ">."
- The name of the tag is *form* (lowercase preferred), which comes immediately after the "<."
- At least one attribute is present in the tag—*action*—whose value contains some target resource in the web application.
- Each attribute is separated from its neighbor (or the tag name, *form*) by at least one space.
- The attribute name is followed (usually immediately) by the equal sign ("="), which is followed (again, usually immediately) by the attribute value. The correct form is to surround the value with double quotes.

Fortunately, you're not being tested on what constitutes well-formed HTML, which demands a book in its own right. What you really need to know is the interaction between a `<form>` definition and a web application, as determined by the *action* and *method* attributes.

- The value for *action*, as we noted above, denotes some target resource in the web application. This target is most often a dynamic resource, such as a servlet or JSP. However, no rule says that it has to be: Static resources can be the target of an action as well.
- The value for *method* denotes the HTTP method to execute. The default (if this attribute is missed out) is to execute an HTTP GET when the form is submitted. Most typically, you use this attribute to specify that the method is POST (this has advantages for parameter sending, which we will discuss soon).

The path for the *action* parameter obeys the rules that apply for any web page link. Suppose you call the web page containing the form with the following URL:

```
http://localhost:8080/ex0101/index.jsp
```

And the form tag within index.jsp looks like this:

```
<form action="someServlet">
```

Then when you submit the form, the browser will assume that someServlet resides in the same place as index.jsp. So the full URL for the request will be

```
http://localhost:8080/ex0101/someServlet
```

If—on the other hand—your path begins with a slash:

```
<form action="/someServlet">
```

Then the browser will assume the path begins at the root location for the specified host, in this case http://localhost:8080. Hence, the full URL would look like this:

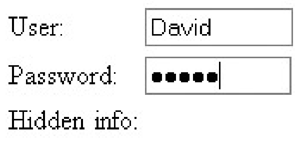
```
http://localhost:8080/someServlet
```

It's important to distinguish this behavior from paths constructed within servlet code, as we'll discover in Chapter 2.

on the job

There can be more than one form on the same web page. However, only one can be submitted at once. This technique can be useful if you are dealing with multiple rows in an HTML table. If your application allows you to operate on only one row at once, it can save a lot of data passing on the network to have a form associated with each row.

FIGURE 1-3



<input> Types for Data Input

Single Line Form Controls

Most form controls for data input are controlled through HTML's `<input>` tag. By varying the value for the *type* attribute, a wide range of field types are available. Figure 1-3 shows three of those—those that are dedicated to holding a single line of text. One—being a hidden field—is invisible!

Common to all `<input>` tags, whatever their type, is the *name* attribute. When form control data are passed to the server, the *name* attribute supplies the parameter name. The *value* attribute supplies the parameter value. This doesn't have to be included in the HTML for the form because whatever the user types in or selects will become the value for a control's *value* attribute. However, it can be used within HTML to supply a default value for a field. You'll see these attributes at work in the following definitions.

`<input type="text" />` This control allows a user to input a single line of text. The *size* attribute specifies the width of the text field in characters. The *maxlength* attribute controls the maximum number of characters that a user can type into the text field. The following HTML definition generated the field with the label "User" in Figure 1-3:

```
<input type="text" name="user" size="10" maxlength="5" />
```

This definition sets up a field whose name is "user." The width of the field as displayed is ten characters, but the user will be allowed to input only five characters. Suppose I type "David" into the field; when I press the submit button on my form, a parameter name/value pair will be sent in the form *user=David*.

`<input type="password" />` The *password* input type works just like *text*. The only difference is that a browser should mask the characters typed in by the user. The following HTML definition generated the field with the label "Password" in Figure 1-3:

```
<input type="password" name="password" size="10" maxlength="5" />
```

Assuming I type in "wells" as the password, the parameter passed to a server will be *password=wells*.

`<input type="hidden" />` The *hidden* input type is not available for user input. As its name implies, such a form control is hidden—invisible as rendered by the browser. You see only the contents of a hidden field if you view the HTML source of the web page. A hidden form control can be useful in two ways:

1. A servlet can write values to hidden fields. These may not be directly useful (and hence not visible) to the user of the web page that the servlet generates. However, when the user requests the next servlet in the chain (by clicking a button on the web page), these values may be useful as parameters to the next servlet. It's one approach to "session control" (answering such questions as "What's in my shopping cart?").
2. Script running within the web page can set the hidden values. This could be contingent on anything—a mouse movement, a keyboard press, an action taken in an applet or a Flash control, or even the selection of a value in some other control.

The following HTML definition generated the invisibly present "hiddenInfo" field in Figure 1-3:

```
<input type="hidden" name="hiddenInfo" value="Discrete Information" />
```

Parameter construction is no different: *hiddenInfo=Discrete Information* will be passed to the server.

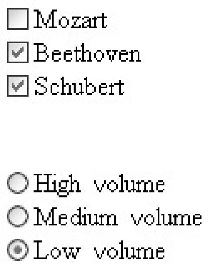
Multiple Choice Form Controls

Often, instead of allowing direct text input, it's better to have groups and lists of predefined choices in your user interface. You have several ways to achieve this—two still using the `<input>` tag and one using the entirely separate `<select>` tag. Figure 1-4 shows each of these in a browser screenshot.

`<input type="checkbox" />` Use *checkbox* to put one or more small boxes on screen. By clicking on these, the user ticks or "checks" the corresponding value to denote that it is selected—so in Figure 1-4, the musicians Beethoven and Schubert are selected. Let's look at the HTML used to set up the fields:

FIGURE 1-4

Multiple Choice Form Controls



```
<br /><input type="checkbox"
            name="musicians" value="MOZRT" />Mozart
<br /><input type="checkbox"
            name="musicians" value="BTHVN" />Beethoven
<br /><input type="checkbox"
            name="musicians" value="SCHBT" />Schubert
```

Note that all these checkboxes share a name in common: musicians. There is no technical necessity for this—every checkbox can have an independent name. However, it is the usual convention when the checkboxes are closely related.

This is our first example of a parameter with multiple values. If you make the choices shown in Figure 1-4 (and press the submit button on the form), the parameter string generated from this set of checkboxes will look like this:

```
musicians=BTHVN&musicians=SCHBT
```

There are two issues to note: Only the checked values make it to the parameter string (that's how you know which ones are chosen). Also, it's the value of the attribute *value* that is passed through on the right-hand side of the equal sign. That's no different from the way other `<input>` types work—it's just that for regular "text" fields, the *value* attribute isn't necessarily set up in the HTML.

We'll see a bit later in this section that servlet code has no issues when dealing with multiple values for the same parameter name.

Should you wish to set up some of your checkboxes as already selected, you can, using the HTML syntax (in bold) below:

```
<br /><input type="checkbox"
            name="musicians" value="BTHVN"
            checked="checked" />Beethoven
```

It's actually sufficient to write the attribute name (*checked*) alone, but I am giving you the benefit of full XHTML syntax, which you should of course strive for. There's nothing stopping the user unselecting a checkbox defined in this way, which prevents the value being passed through as a parameter.

`<input type="radio" />` The construction of radio buttons is very similar. This time, however, the choice made is mutually exclusive. Also, the name attribute is crucial to tying together a group of radio buttons. Here's the HTML that creates the radio button example shown in Figure 1-4:

```
<BR /><INPUT type="radio" name="volume" value="HIGH" />High Volume
<BR /><INPUT type="radio" name="volume"
             value="MED" checked="checked" />Medium Volume
<BR /><INPUT type="radio" name="volume"
             value="LOW" checked="checked" />Low Volume
```

The name attribute is set to "volume" for each of the three choices. The value attribute is set appropriately, and as for checkboxes, you will need to place some adjacent regular text (e.g., "High Volume") to act as a label; otherwise, the radio button appears without any choice description. As shown, there are two radio buttons "preselected" using the *checked="checked"* syntax. However, only one choice is possible. The browser resolves this by letting the last one marked as checked (in this case, "Low Volume") take precedence. So unless the user makes a choice here, the parameter string passed through will be *volume=LOW*.

The `<select>` tag Finally in this "multiple choice" array of form controls, we consider an entirely separate tag: `<select>`. This control allows you to set up a list of values to choose from in a web page—the style is either a pop-up menu or a scrollable list. There are two "modes" for this control: The user is either restricted to one choice from the list or has multiple choices from the list. Both sorts are illustrated in Figure 1-5. Let's consider the HTML for the single-choice, pop-up menu first:

```
<select name="Countries">
  <option value="FR">France</option>
  <option value="GB" selected>Great Britain</option>
  <option value="DK">Denmark</option>
</select>
```

This time, there is an outer tag beginning with `<select>` and closing with `</select>`. The opening tag has a name attribute, which will form the parameter name—in this case, "Countries." Within the `<select>` are nested the predefined list choices, each in an individual `<option>` tag. The user-visible text goes between the opening `<option>` and closing `</option>`; the value passed in the parameter is expressed as the *value* attribute of the opening `<option>` tag. In order to preselect an item in the list, the attribute *selected* is added to the opening `<option>` tag. So the parameter string looks like this: *Countries=GB.*

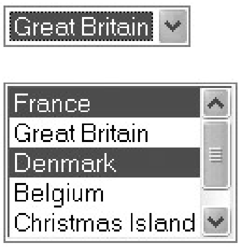
The alternative form of `<select>` is very similar, apart from the presence of the attribute name *multiple* in the opening `<select>` tag. It's illustrated in the lower part of Figure 1-5. Here's our list of countries again, but now you can choose more than one:

```
<select name="Countries" size="5" multiple>
  <option value="FR" selected>France</option>
  <option value="GB">Great Britain</option>
  <option value="DK" selected>Denmark</option>
  <option value="BE">Belgium</option>
  <option value="CX">Christmas Island</option>
  <option value="CO">Colombia</option>
</select>
```

This time, France and Denmark are preselected. It's a good idea to use the *size* attribute. At least with the browser I'm using, setting this to the same value as the number of `<option>` elements gives you a box with all the options shown. When the size value is less than the number of options, you get a scrolling region that displays as many rows as you specify in the size attribute. So in the example above, you need to scroll down to see Colombia. Leaving the size attribute out altogether allows the browser to impose its own rules, which may or may not give the effect you want. If the user leaves the above preselections unaltered, the parameter string passed through is *Countries=FR&Countries=DK.*

FIGURE 1-5

Select Tag



Multiple Lines of Text

Should one line of text be insufficient, and should you want your user to be able to type in an essay (or at least a multiline comment), you can resort to the `<textarea>` tag. This has an opening tag and an ending tag—you can put any text you like between the tags. This text will display in the middle of an editing box. The user can overtype or add to any text already there, and this constitutes the value of the parameter passed back to the server. The opening `<textarea>` tag has three attributes of consequence:

- *name*—as elsewhere, the parameter name
- *rows*—the number of visible lines (a scroll bar activates when access to further rows is needed)
- *cols*—the number of characters to displayed across the width of the area (based on an average-width character)

Another attribute—wrap—offers some flexibility in the treatment of carriage return and line feed characters, introduced either by the user pressing the enter key or the browser wrapping the text. There is no HTML standard for this attribute, and implementations vary slightly from browser to browser. All you need to know for practical purposes is that your servlet code should be able to deal with carriage return and line feed characters within the body of the text returned.

Figure 1-6 shows a text area, and below is the HTML that produced it:

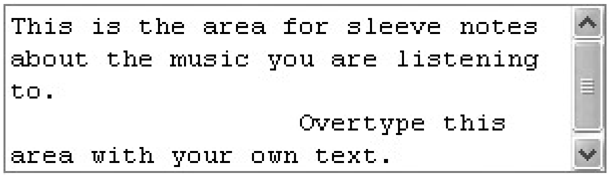
```
<TEXTAREA name="notes" rows="5" cols="35">
  This is the area for sleeve notes about the music
  you are listening to.
  Overtype with your own text.
</TEXTAREA>
```

The parameter string passed back from this text area (from Internet Explorer) bears some attention:

```
notes=This+is+the+area+for+sleeve+notes+about+the+music+you+are+listening+to.%0
D%0A%09%09++Overtype+this+area+with+your+own+text.
```

FIGURE 1-6

A Text Area



```
This is the area for sleeve notes
about the music you are listening
to.
                  Overtype this
area with your own text.
```

Most browsers have a horror of embedded white space in their parameter strings. Internet Explorer here substitutes a + for the space character (%20 is sometimes seen instead). Substitutes are also made for "non-typeable" characters. %0D and %0A are hex representations of the line feed and carriage return characters, and %09 is the horizontal tab. This reflects the fact that the full HTML source was set up in a text editor that inserted exactly those characters.

exam watch

You've seen how you can get multiple values associated with a related group of form controls—checkboxes all sharing the same name, for example. However, there is nothing illegal about entirely separate form controls sharing the same name. You can have, say, a checkbox, a set of radio buttons, and a text field that all share the same name—and so give rise to multiple values for the same parameter name when retrieved in the target servlet code.

Buttons

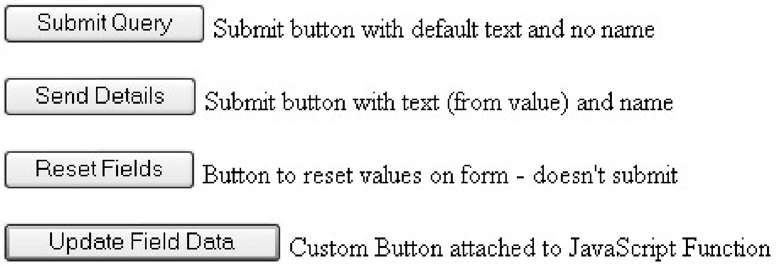
Now that we've surveyed all the form controls that allow data input or selection, let's consider other actions you can take with your form. Most especially, we need to know how to submit the data that our user has so generously supplied! For this and other actions triggered by buttons, we return to the `<input>` tag. There are three types: *submit*, *reset*, and *button*. They are no different in appearance; Figure 1-7 shows examples.

`<input type="submit" />` The purpose of an input tag whose *type* is submit is to send the form data to the URL designated by the *action* attribute of the opening `<form>` tag. To define it, no more is required than defining the *type*:

```
<input type="submit" />
```

FIGURE 1-7

Buttons within an HTML Form


Submit Query
Submit button with default text and no name
Send Details
Submit button with text (from value) and name
Reset Fields
Button to reset values on form - doesn't submit
Update Field Data
Custom Button attached to JavaScript Function

However, it is quite usual to supply the *value* attribute, which allows you to define your own text on the button. If you supply a *name* attribute as well, then the name/value data for the submit button are passed just like any other parameter as part of the submitted form data. The HTML below defines the submit button seen in Figure 1-7:

```
<input type="submit" name="formSubmission" value="Send Details" />
```

It gives rise to parameter data like this: *formSubmission=Send+Details*.

`<input type="reset" />` When an input tag is a *reset* type, the resulting button doesn't send form data. No request is made to the server. Instead, it's a request to the client browser to reset all the values within the form to the way they were when the page was first loaded—in other words, to scrap any user input since that point. You can regard it as an all-or-nothing "undo" facility. Again, you can supply a name attribute and have the button name/value passed as a parameter (though it's hard to find a good reason for doing this). The reset button in Figure 1-7 has the following HTML definition:

```
<INPUT type="reset" value="Reset Fields" />
```

`<input type="button" />` Slightly confusingly, there is a third type of button whose type is—well—*button*. Surely the other two types were buttons as well? It's best to think of type button as defining a "custom button," which is connected to some sort of script within your page. You typically harness this by defining an *onclick* attribute within the input tag and making the value correspond to some JavaScript function. The JavaScript function can do pretty much anything—up to and including submission of the form. More usually, functions define themselves to effects within the browser, such as changing which form control has focus or validating the contents of a field. Here's a fairly trivial example that uses a custom button to take the contents of a text-type input field and place the contents in a hidden field. It uses a simple message dialog to display the old and new values of the hidden field. Here's the script, contained in the `<head>` section of the enclosing HTML page:

```
<head>
  <title>Welcome</title>
  <script language="JavaScript">
  <!--
    function setAndDisplayHiddenField() {
      hiddenField = document.getElementById("hiddenInfo");
      alert("Current Value Of Hidden Field: " + hiddenField.value);
      hiddenField.value = document.getElementById("inputToHidden").value;
      alert("New Value Of Hidden Field: " + hiddenField.value);
    }
    -->
  </script>
</head>
```

The button that summons this script is illustrated in Figure 1-7 and has the following HTML definition:

```
<INPUT type="button" value="Update Field Data"
onclick="setAndDisplayHiddenField();" />
```

Note that if you're trying this script for yourself, you'll need a hidden field named "hiddenInfo" and a text field named "inputToHidden" to make it work. Type text into "inputToHidden," press the button, and the value will be transferred to "hiddenInfo"—the message dialogs will prove this to you.

When all is said and done, JavaScript is well "off topic" for the SCWCD but has a nasty habit of creeping into the day-to-day headaches of a web application developer.

Retrieving Parameters

Now that you have mastered HTML form controls, and understand the parameter data generated from them, we can move to the server side. Our first proper look at servlet code will examine the issue of retrieving parameter data. This is usually the first and most vital step in any web application—getting hold of what a user has supplied.

Servlet APIs for Parameters

Parameters are part of an HTTP request. However, they are regarded as so fundamental to servlet workings that the APIs to retrieve them are found on javax.servlet .ServletRequest, the parent interface for javax.servlet.http.HttpServletRequest interface (which you might consider a more likely home). Your servlet engine provides a class implementing the HttpServletRequest interface, and it passes an instance of this as a parameter to whichever of a servlet's `doXXX()` methods is targeted by an HTTP request. The HttpServletRequest object encapsulates information in the request, providing APIs to make it easy to access this information. Since HttpServlet Request inherits from ServletRequest, any implementing class must provide the ServletRequest methods as well, and these include methods for parameter access.

exam watch

In the heat of web application programming, you're unlikely to care very much whether a method originates from HttpServletRequest or the parent ServletRequest interface. Unfortunately, the exam does include questions that target knowledge of origin interfaces—there's no real option but to commit the methods on HttpServletRequest and ServletRequest to memory prior to taking the exam.

When you know the name of a parameter, and can guarantee a single value back from the form, then the easiest API to use is `ServletRequest.getParameter (String parmName)`. This returns a String representing the parameter value, or **null** if the parameter doesn't exist.

However, we've already seen that parameters can have the same name and multiple values. When you are after a specific parameter and know the name, your best bet is to use the `ServletRequest.getParameterValues(String parmName)` method. This returns a String array with all values present, or **null** if no value exists for the parameter name. If this array has a length of 1, this indicates that there was only one value. Can you predict anything about the order of the values? In practice, the order appears to reflect the left-to-right, top-to-bottom occurrence of the values within the form controls. However, the servlet specification is silent on this question. All you can guarantee is that parameters in the query string (associated with the URL) should be placed before parameters in a POST request body. This isn't greatly informative, for mostly you only get one or the other—parameters placed in the query string by a GET request, or parameters embedded in the request body by a POST request. However, you might encounter this sort of form declaration:

```
<form action="servlets/MyServlet?myparm=thisfirst" method="post">
```

This uses the POST method—which will put form parameters in the request body—but also specifies a parameter in a query string in addition.

The one other guarantee on offer is that the parameter value returned by `getParameter()` must be the first value in the array returned by `getParameterValues()`.

When you want to find out about all the parameters in a request, you have two options:

- `ServletRequest.getParameterNames()`, returning an Enumeration of String objects representing the names of all the parameters in the request. There's only one occurrence of any given name; however, many form controls share that name. Should no parameters be passed, the Enumeration is empty. To get hold of corresponding values, each parameter name retrieved from the Enumeration can be plugged into either of the `getParameter()` or `getParameterValues()` methods discussed above.
- `ServletRequest.getParameterMap()`, returning a java.util.Map object, where the keys in the map are of type String (and represent each unique parameter name) and the values in the map of type String array (representing the values for the parameter). Though the API documentation doesn't come clean on what should apply, my Tomcat implementation returns an empty Map if there are no parameters present—which makes sense and matches the behavior of `getParameterNames()`. The map returned is immutable. That means that if you try yourself to `put()` key/value pairs in the Map retrieved from this method, you will be told in no uncertain terms that this isn't appropriate behavior (Illegal StateException for Tomcat 5). This is right and proper—you shouldn't be adding to the list of genuine parameters. There are plenty of other mechanisms for storing data with the request, and other scopes that we meet in Chapter 2.

Getting at Parameters the Hard Way

Actually, there *is* an alternative way to get at POSTed parameters. You can read the request body directly by using `ServletRequest.getReader()` or `ServletRequest.getInputStream()`. It's then up to you to examine the resulting character or input stream for name/value pairs separated by ampersands. This process is not that hard, as you can see from the code below:

```
protected void doPost(HttpServletRequest request,
 HttpServletResponse response)
 throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.write("<html><head></head><body>");
  BufferedReader reader = new BufferedReader(new InputStreamReader(
                        request.getInputStream()));
  String line;
  while ((line = reader.readLine()) != null) {
    StringTokenizer st = new StringTokenizer(line, "&");
    while (st.hasMoreTokens()) {
       out.write("<br /" + st.nextToken());
    }
  }
  out.write("</body></html>");
}
```

This code uses a BufferedReader to work through the lines in the POSTed body. A StringTokenizer breaks down the input by splitting out the text between ampersands (which should delimit each name/value pair, internally separated by an equal sign). The servlet simply writes out each name/value pair to a new line on the web page. Note in this approach how multivalued attributes are written out many times. Suppose a web page submits a list of countries from a multiselect `<select>` form control named "Countries." The output from the servlet code above might look like this:

```
Countries=GB
Countries=DK
Countries=FR
```

Should you attempt to use `getInputStream()` after you have used `getReader()`, you are likely to get a message such as the one below:

```
java.lang.IllegalStateException: getReader() has already been
 called for this request
```

The reverse holds equally true—don't call `getReader()` after `getInputStream()`. These methods can blow up in other ways: with a straight IOException (if something goes wrong with the input/output process) or—for `getReader()` only—with an UnsupportedEncodingException if the character set encoding used is not supported on the platform, and the text therefore remains un-encodeable.

on the job

You won't often want to mess directly with the POST data; after all, it's much easier to use the getParameter* methods. However, there are occasions when you might want or even need to. This is when a POST request is being used to upload a file, usually in conjunction with an HTML `<input>` type we didn't explore in our form control review. The following tag—`<input type ="file" name="fileForUpload" />`—creates a button and a text field in your browser. The button can be used to launch a "choose file" type dialog allowing you to browse your local file system—the result of the choice is stored in the text field. When you submit the form, the chosen file is uploaded in the post body. The servlet code that receives the file (or files—a

single form can have several input elements where the type is "file") and does something with it that is not trivial—you can find a fabulous implementation (complete with source code) at http://www.servlets.com/cos.

exam watch

Conceptually, the data from a POSTed form are transferred to a "parameter set" and, once there, are available to convenience methods such as `getParameter()`. The conditions for the data to be present in the set are as follows: The request follows the HTTP or HTTPS protocol, using the POST method. The content type for the request (as declared in the Content-Type request header) must have a value of "application/x-www-form-urlencoded." Finally, a call must be made to one of the getParameter* convenience methods: The first such call causes the transfer. Once data have been transferred to the parameter set, you must not use the request's `getInputStream()` or `getReader()` to get hold of the parameters. Although you may not get any exception, the servlet specification promises unpredictable results, and this is borne out by practical experimentation. The reverse is also true; a call to a getParameter* method is unlikely to do you much good if the request body has been treated as a raw byte or character stream.

There's an alternative way to get at parameters that are sent in a GET request. That's with the `HttpServletRequest.getQueryString()` method. This returns the raw data following the question mark (if present) in a URL—you'll find a discussion of this in Chapter 3, in the "Forwarding" section.

EXERCISE 1-2

ON THE CD

Form Parameters

In this exercise, you'll construct an HTML form containing a number of controls. Then you'll write a servlet to receive the parameters from the form and display the choices made on a web page.

This exercise is the first of many that follow a similar pattern, described in Appendix B. Most exercises and labs exist as unique "web applications" in their own right. Full instructions are provided in Appendix B, and a solution (which you can deploy and look at independently) is included in the CD supplied with the book. The unique directory for this exercise is ex0102, and you'll need to create that (together with a directory structure underneath it, as described in Appendix B).

The solution is on the CD in the file sourcecode/ch01/ex0102.war—check there if you get stuck. Instructions for deploying the solution WAR file are also in Appendix B.

Write the HTML Form

1. Create an HTML file in directory ex0102 called weather.html.
2. Create a form on the page whose method is POST and whose action is "Weather." All the remaining instructions for the HTML form pertain to controls that should come between the opening and closing `<form>` tags.
3. Create an input field (with a *type* of "text") for the name of the person observing the weather.
4. Create an input field (with a *type* of "password") for the observer's password.
5. Create a hidden input field containing any information you like.
6. Create a series of checkboxes, all with the same name, to record different types of weather observed (rain, sun, snow, fog, and so on).
7. Create a series of radio buttons, all with the same name, for the observer to select a suitable temperature range to reflect the highest temperature achieved today.
8. Create a select box with three options for each of three possible weather stations.
9. Create a text area to hold comments on today's weather.
10. Finally, create a submit button (`<input>` of type *submit*) to send the form data to the servlet that you are about to write.

Write the WeatherParams Servlet

11. You're going to write a servlet that picks up all the parameters from the form and reflects these back on a web page to the user. You need to create a source file called WeatherParams.java, with a package of webcert.ch01.ex0102. Place this in an appropriate directory (webcert/ch01/ex0102) within the exercise subdirectory ex0102/WEB-INF/classes.
12. Following the package statement, your servlet code should import packages java.io, java.util, javax.servlet, and javax.servlet.http.
13. Your WeatherParams class should extend HttpServlet (and doesn't need to implement any interfaces).
14. You will override one method from the parent class, which is `doPost()`. The method signature is as follows:

```
protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
              throws ServletException, IOException
```

15. Here's some boilerplate code to place at the beginning of the method. This gets hold of a PrintWriter from the response, giving you a slate to write on to create your web page. The code also sets an appropriate MIME type to indicate you're generating HTML from your code, and starts off an HTML page in the right way. You'll learn more about all this in the next section.

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.write("<html>\n<head>\n<title>" +
  "Display Weather Parameters</title>\n</head>\n<body>");
```

16. Here's some boilerplate code to place at the end of the method. This closes off your HTML web page properly and closes the response's PrintWriter:

```
out.write("\n</body>\n</html>");
out.close();
```

17. All your remaining code goes between the two pieces you input above.
18. Your aim now is to write code to retrieve all the parameter names passed to the servlet and to write these as headings for the web page. The heading-writing code will look something like this:

```
out.write("<h4>" + paramName + "</h4>");
```

The variable *paramName* is a String holding each parameter name you've retrieved—in some kind of loop. For each name you retrieve, write out all the parameter values underneath. Here's the code to do the writing so that each value appears on a new line in the web page:

```
out.write("<br />" + paramValue);
```

As for the remaining code to do the parameter name and value retrieval, you're on your own for that. Well, not entirely. Refer back to and adapt the code examples in this section. If you get really stuck, refer to the source in the solution code.

19. Compile your servlet code in the same directory as your source file.

Running Your Code

20. You won’t learn about deployment and WAR files until Chapter 2, so for this chapter’s exercises we will cheat a bit.
21. Start the Tomcat Server.
22. Deploy the *solution* code WAR file, ex0102.war (follow the instructions for deploying WAR files in Appendix B).
23. Copy weather.html from your directory structure to the <Tomcat Installation Directory>/webapps/ex0102 directory (overwrite the solution version).
24. Copy WeatherParams.class from your directory structure to <Tomcat Installation Directory>/webapps/ex0102/WEB-INF/classes/webcert/ch01/ex0102 (overwrite the solution version).
25. Point your browser to an appropriate URL. For a default Tomcat installation, this will be

    ```
    http://localhost:8080/ex0102/weather.html
    ```

26. Test your code by filling in the parameters and pressing the submit button. If all goes well, you'll get a web page back that tells you the parameters you chose.

CERTIFICATION OBJECTIVE

Requests (Exam Objective 1.2)

Using the HttpServletRequest interface, . . . retrieve HTTP request header information or retrieve cookies from the request.

In this section you will work through the several APIs available on HttpServlet Request that break down the available information on an HTTP request. The APIs are straightforward. You will also encounter several header properties recognized by the HTTP protocol and learn the significance they have for servlet containers. We'll explore some of the more common header properties that sometimes grace the screens of the SCWCD exam.

We'll also look at the question of cookies — those small and useful text files that can be uploaded from browsers (we'll look at downloading to browsers in the next section, on the HttpServletResponse interface). You might regard cookies as a violation of your privacy, but there's no denying their usefulness — and you can't deny their place as a core topic on the exam syllabus!

Request Headers

The HTTP RFC lays out an extensive list of separate pieces of header information that can accompany a request (or a response). These are described as name/value pairs, very much like parameters. There are almost fifty valid named headers. And again very similar to parameters, one named header can have multiple values. Headers can be categorized into four types:

- Request Headers: pertaining strictly to the request—for example, communicating to the server what the client will find acceptable in terms of file formats and encodings.
- Response Headers: pertaining strictly to the response—for example, describing a specific aspect of how the server responded to a particular request.
- General Headers: can occur on either the request or the response. "Date" is a good example — this contains a timestamp for the HTTP message, so it is equally applicable whatever message is under consideration.
- Entity Headers: again, applicable to both request and response. These headers have information about the request or response body (how it's encoded or encrypted, for example).

Within an HTTP message, headers' names are separated from their values by a colon, multiple values are comma-separated, and each header is separated from the next by a carriage return. Here's how an HTTP request line looks with its headers immediately following:

```
GET /search?q=MIME&ie=UTF-8&oe=UTF-8&hl=en&btnG=Google+Search&meta= HTTP/1.0
Accept: image/*, application/vnd.ms-excel, */*
Accept-Language: en-gb
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Host: www.google.co.uk
Connection: Keep-Alive
 [BLANK LINE]
```

A blank line finishes the header section, separating this from any attached request body.

Our interaction with headers is made easier through the provision of five methods on HttpServletRequest. Here they are in summary:

HttpServletRequest Method	Description
String `getHeader(String name)`	Returns the (first or only) value of the specified request header as a String
Enumeration `getHeaders(String name)`	Returns all values of the specified request header as Strings within an Enumeration
Enumeration `getHeaderNames()`	Returns the names of all request headers as Strings within an Enumeration
long `getDateHeader(String name)`	Returns the value of the specified request header as a **long** primitive representing a date
int `getIntHeader(String name)`	Returns the value of the specified request header as an **int** primitive

exam watch

Method families that return Strings and Enumerations have a typical pattern—compare the getHeader* family with getParameter* (on the ServletRequest interface) and getAttribute* (on several interfaces). If there is no header (or parameter or attribute) of the given name, methods returning a String will return null. Methods returning Enumerations will return an instance of an Enumeration object, but it will be empty. (That said, look carefully at the API documentation for `getHeader(String name)` and `getHeaderNames()`).

The simplest method is `getHeader(String name)`, which returns a String—use this when you know the name of the request header you want and you know there will only ever be one value. If the name is not recognized, **null** is returned. Note that with this and the other header methods, you can specify the name in an entirely case-InSensitive manner.

- `getHeaders(String name)` returns all the values for the given request header name, in the form of an Enumeration containing Strings. The Enumeration object will be empty if the header name is not recognized, or simply not present.

- `getHeaderNames()` returns an Enumeration of Strings representing all available request header names, which you will typically feed into successive calls to `getHeader(String name)` or `getHeaders(String name)`. If there are no request headers, this Enumeration will be empty—although it's most unlikely you would ever receive an HTTP request (at least from a conventional browser) that contained no header information at all.

For `getHeaders(String Name)` and `getHeaderNames()`, it's possible for the servlet container to deny access to the request headers. In this case—and only this case—**null** may be returned instead of an Enumeration object.

You can do everything you need to with the methods above. However, there are a couple of convenience methods when you know that the value returned for a specific request header name will represent either a date or an integer. The two methods are `getDateHeader(String name)` and `getIntHeader(String name)`, which return a **long** (representing a date) and an **int**, respectively. These methods perform conversion from the original String representation of the header's value. Of course, there's the possibility of using these methods inappropriately— one way is to supply a header name that doesn't convert to a **long** date or an **int**. In that case, `getDateHeader()` will throw an IllegalArgumentException, while `getIntHeader()` will throw a NumberFormatException. Both methods return a value of −1 if the requested header is missing. There's an assumption there that `getIntHeader()` could never legitimately return a negative value, while any dates sought with `getDateHeader()` are after midnight on January 1, 1970, Greenwich Mean Time!

Here's a short code example that you might embed in a `doGet()` or `doPost()` method to obtain a date header, and write this to a resulting web page ("response" is the HttpServletResponse object passed as parameter to the `doGet()` or `doPost()` method and "request" is the HttpServletRequest object). Of course, the code runs satisfactorily only if an "If-Modified-Since" date header is supplied in the HTTP request.

```
PrintWriter out = response.getWriter();
long aDateHeader = request.getDateHeader("If-Modified-Since");
DateFormat df = DateFormat.getDateInstance();
String displayDate = df.format(new Date(aDateHeader));
out.write("<br>If-Modified-Since Request Header has value: " + displayDate);
```

What you next need to consider is some of the actual HTTP request headers you're likely to deal with. Table 1-1 summarizes these. The table shows the request header name, describes its purpose briefly, shows an example value, and has a column to indicate whether the value is a date or an integer, and therefore amenable to the use of the `getDateHeader()` or `getIntHeader()` convenience method.

TABLE 1-1 Common Request Headers

Request Header Name	Description	Example Value	int (I) or date as long (D)
Accept	MIME types acceptable to client (we'll explore what a MIME type is later: for now, think file formats).	text/html, text/plain, image/*	
Accept-Charset	ISO character sets acceptable to client. ISO-8859-1 is assumed as the default.	ISO-8859-6	
Accept-Encoding	The content encoding acceptable to client—usually associated with compression methods.	gzip, compress	
Accept-Language	The (human) language acceptable to client. ISO codes are used to denote which language.	en, us	
Authorization	Authentication information. Usually provided after the server has returned a 401 response code (indicating that the request requires user authentication).	(at simplest, user ID and password passed with minimum encoding)	
From	The e-mail address of the request sender. The server might use this to log request origins, or to notify the sender of unwanted requests.	zebedee@ magicroundabout.com	
Host	Internet host and port number from the request URL. Mandatory with HTTP 1.1 requests.	localhost:8080	
If-Modified-Since	If the requested resource has not been modified since the date given, it's not returned: A 304 status code is returned instead.	Thu, 07 May 2003 14:01:31 GMT	D
Max-Forwards	The maximum number of interim proxy servers a request can be forwarded to.	5	I
Referer	The URL of the resource that had the link to the resource now being requested. *Note the misspelling*—should be "Referrer," but "Referer" is kept for historical reasons.	http://www.osborne .com/mybookshelf	
User-Agent	Information about the client software (typically, browser) making the request—helpful to the server in tailoring responses to clients or in gathering statistics.	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)	

TABLE 1-2 Common General Headers

General Header Name	Description	Example Value	int (I) or date as long (D)
Cache-Contro	Directives that must be followed by all caches in the request/response chain—a complex area and not one you need to understand for the exam.	max-age=10000 (This means that the resource in the cache must not be returned if over 10,000 seconds old.)	
Connection	A directive about the connection (from server to client or vice versa). Typically used to denote that connections cannot be persistent.	close	
Date	Date and time when a message is originated. This is almost always supplied by the server (in a response), but seldom supplied by a client (in a request).	Tue, 15 Nov 1994 08:12:31 GMT (note the Internet standard date format)	D
Transfer-Encoding	The transfer encoding applied to the message (which gives options around the way the message is structured).	chunked	

At the beginning of this section, we noted that request headers are not the only kind of header. There are general headers, entity headers, and response headers as well. We'll defer discussion of response headers until we look in more detail at responses later in this chapter. However, let's devote a couple of tables to common general headers (Table 1-2) and entity headers (Table 1-3) that you might discover in the request (or place in the response).

Remember that the entity headers (listed in Table 1-3) describe things about the request (or response) body, if this is present in the HTTP message.

Cookies

Cookies are small text files passed between client browsers and web servers. They are used in part as an extended parameter mechanism. Although generally limited to a few thousand bytes in length, a cookie can store a great deal of essential information—often enough to identify the client and to store information about choices made. Nearly every sizable commercial web site uses cookies to "personalize" the user's experience. In this section, we're interested in one direction only—intercepting cookies sent from the browser. We'll look at the other direction (setting up cookies

TABLE 1-3 Common Entity Headers

Entity Header Name	Description	Example Value	int (I) or date as long (D)
Allow	HTTP options permitted for the requested resource. This is the value returned by the HTTP OPTIONS method.	GET, HEAD, OPTIONS, TRACE	
Content-Encoding	A modifier to the media type ("Content-Type"), indicating further encoding of the entity. Used mainly to allow a document to be compressed.	gzip	
Content-Language	The natural (human) language for the entity's intended audience.	En	
Content-Length	Length of the entity in bytes.	8124	I
Content-Location	The URI denoting where the requested resource is to be found on the server. Typically the same as the request URI—but can be different (see the discussion of welcome-file-list in Chapter 2 for one reason that this comes about).	http://www.osborne.com/index.html (This might be returned when the request URI was simply http://www.osborne.com.)	
Content-Type	The MIME type of the entitytext/html.		
Expires	Date/Time by which the response—returned from a cache—is considered stale.	Thu, 17 Nov 1994 09:13:32 GMT	D
Last-Modified	Date/Time on which the entity was last modified, as best the server can determine. This is easy for static resources (by using the date of a file on the file system, for example), but is less obvious for dynamic resources.	Thu, 17 Nov 1994 09:13:32 GMT	D

at the server end to send to the browser) when we get around to the HttpServlet Response interface later in the chapter.

Getting Cookies from the Request

Getting cookies passed with the request is easy—you simply use the `HttpServletRequest.getCookies()` method. This returns an array of javax.servlet.http.Cookie objects, or **null** if none are sent with the request.

You'll need to have at least a passing familiarity with the things you can do with a Cookie. It's a very simple class, consisting entirely of data with some getters and setters. The data attributes are shown in Table 1-4.

TABLE 1-4 Cookie Attributes

Attribute	Type	Mandatory?	Description
Name	String	Yes	May contain only ASCII alphanumeric characters; cannot contain commas, semicolons, or whitespace; must not begin with a $ character. The name can't be changed once passed into the constructor—hence, there is no `setName()` method.
Value	String	Yes	A String value. To be compliant with version 0 cookies (see "Version," below), a value can't have any embedded white space, and most punctuation signs are banned: (){}[]=@,:;?" \/ are explicitly outlawed. That does leave a few options for delimiters—* and—and _, for example.
Domain	String	No	The domain to which the cookie is applicable—if visiting that domain, the browser should send the cookie (e.g., google.com). Although not a mandatory attribute, it's hard to imagine a very functional cookie without this attribute set.
Path	String	No	A path for the client to return the cookie to—specifically, this is meant to match the path of the servlet that set the cookie (e.g., ex0101/CookieMakerServlet). This meaning of the path is that this cookie should be visible to resources invoked with this directory path, or to resources held in any subdirectories of it (e.g., ex0101/CookieMakerServlet/chocchip).
Comment	String	No	A comment meant to explain the purpose of a cookie to a user (if the browser is designed to present such comments)—supported at Version 1 only.
MaxAge	int	No	The maximum age of the cookie in seconds. There are two special values: ■ any negative value: denotes that the cookie should be deleted on exiting the browser (a transient cookie) ■ a value of zero: denotes that the cookie should be deleted
Secure	boolean	No	When set to **true**, indicates that the cookie should be passed over a secure transport layer (HTTPS, SSL).
Version	int	No	0 indicates the original cookie specification defined by Netscape. 1 indicates the standard defined by RFC 2109. It's less widely supported—the default is that cookies are created at version 0 for maximum compatibility (so you have to explicitly setVersion (1) if that's what you really want).

A Cookie has a two-argument constructor passing in the mandatory name and value—thereafter, you can set any of its attributes except the name, which is treated with the same kind of sanctity reserved for unique keys on database tables. Cookies are transmitted to servers within HTTP request header fields. You can `clone()` a cookie to make a copy of it.

EXERCISE 1-3

ON THE CD

Reading HttpServletRequest Headers and Cookies

In this exercise, you'll write and deploy servlet code to display request headers and the details of any cookies passed with the request. This activity follows the same pattern as the previous exercise. The unique directory is ex0103, and the solution is on the CD in file sourcecode/ch01/ex0103.war.

Write the RequestHeaders Servlet

1. You need to create a source file called RequestHeaders.java, with a package of webcert.ch01.ex0103. Place this in an appropriate package directory (webcert/ch01/ex0103) within the exercise subdirectory ex0103/WEB-INF/classes.
2. Following the package statement, your servlet code should import packages java.io, java.util, javax.servlet, and javax.servlet.http.
3. Your RequestHeaders class should extend HttpServlet (and doesn't need to implement any interfaces).
4. You will override one method from the parent class, which is `doGet()`. The method signature is as follows:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
          throws ServletException, IOException
```

5. You need code to set the response type to HTML and to start and finish the web page with appropriate HTML syntax. Copy and adapt the boilerplate code you used for this purpose in the Exercise 1-2.
6. Using the content of this section as a guide, find the appropriate method to discover all the request header names. This returns an Enumeration—set

up a loop to process each element in turn. Use `out.write()` to output each header name as an HTML heading.

7. Within the loop, insert the method that pulls back all values for a given header name, again as an Enumeration. Feed each header name in turn as a parameter to this method. Set up an inner loop to process all the elements representing header values. Use `out.write()` to output each value in turn to a fresh line on the web page.
8. After the above code, include the following line of code, which ensures that an HttpSession object is associated with your use of this servlet. You don't formally learn about HttpSession objects until Chapter 4. I'm including it here only because it practically guarantees the creation of a cookie to be passed from client to server.
9. Retrieve the array of cookies from the HttpServletRequest passed as parameter to the `doGet()` method. Assuming this is not **null**, process each cookie in turn, and display each attribute of the cookie as a line of text on the web page. Refer to Table 1-4 for a list of attributes of cookies.

Running Your Code

10. Start the Tomcat Server (if not started already).
11. Deploy the *solution* code WAR file, ex0103.war (follow the instructions for deploying WAR files in Appendix B).
12. Copy RequestHeaders.class from your directory structure to <Tomcat Installation Directory>/webapps/ex0103/WEB-INF/classes/webcert/ch01/ex0103 (overwrite the solution version).
13. Point your browser to the appropriate URL. For a default Tomcat installation, this will be

```
http://localhost:8080/ex0103/RequestHeaders
```

If you don't see any cookie information at first, then refresh the browser page. The second and subsequent accesses to the RequestHeaders servlet in the same session should at least guarantee that you see a cookie whose name is JSESSONID.

CERTIFICATION OBJECTIVE

Responses (Exam Objective 1.3)

Using the HttpServletResponse interface, write code to set an HTTP response header, set the content type of the response, acquire a text stream for the response, acquire a binary stream for the response, redirect an HTTP request to another URL, or add cookies to the response.

We've seen some of the fundamental things we can do with HttpServletRequest. Now we're going to deal with its counterpart, HttpServletResponse, a class that conveniently wrappers up the HTTP message sent back to the requester. It's available as the second parameter in the set of `doXXX()` methods within a servlet.

HTTPServletResponse

We'll explore how to set header fields in the response. Some of these header fields we met already when we looked at how to read them from HttpServletRequest. Now we'll discover how to write header fields, some of which have the same name, and others that are new and unique to the response. Earlier, we were able to read cookies from the request; now we'll see how it's possible to write cookies to the response—and at the same time discover that a cookie is really just a specialized kind of response header. Maybe the servlet we invoke isn't quite what the client needs, so we'll explore how easy it is to instruct the client to redirect itself to a different URL.

There are many possibilities for the material you transmit in the response. You'll see how you can help the recipient client by at least hinting at what type of content is in the HTTP response message. You'll also see how to deal with the two fundamental divisions of content—textual and binary—the response provides both Writer- and Stream-based approaches.

Setting Response Header Fields

For the request, we were interested in reading the values of header fields coming into our servlet. For the response, we are interested in setting the values of header fields to send back to the client. We'll explore in a moment the range of methods available on the HttpServletResponse, which complement the getHeader* methods on HttpServletRequest.

In the previous section, on HttpServletRequest, we looked at most common possibilities for header fields. You'll want to look back at the tables in that section. One table listed header fields that applied only to the request, but Tables 1-2 and 1-3 described general and entity header fields that are just as applicable to the response. Table 1-5 shows the most commonly used header fields that you would expect to find only in a response.

There are two approaches to setting headers on the response field. I'd suggest that you most want to use `addHeader(String name, String value)`. This will keep adding values, even where the name is the same (which is just the effect you want if there are multiple values to add—perhaps on the Accept header, for example).

TABLE 1-5 Response Header Fields

Response Header Name	Description	Example Value	int (I) or date as long (D)
Age	If the server has returned the response from a cache, this figure determines the age (in seconds) of the cached resource—giving an indication of its "freshness."	814487	I
Location	Used to redirect the requester to a URI other than the requested URI. Used "under the covers" by the `sendRedirect()` method.	http://www.osborne.com/altlocation.html	
Retry-After	If a service is unavailable, the time after which a client should try again. Could be a time in seconds *or* an absolute date/time.	180 Thu, 07 May 2003 11:59:59 GMT	I D
Server	Information about the web server providing the response—for example, the product and version number.	Apache 2.0 45-dev (Unix)	
Set-Cookie	Used by a server to ask a client to create a cookie according to the details set in the value. Used "under the covers" by the `addCookie()` method.	(A String—usually fairly long—containing cookie fields. Format varies according to cookie version.)	
WWW-Authenticate	Accompanies a response status code of 401 (unauthorized). This field describes the authentication method required and expected parameters (authentication methods are discussed in Chapter 5).	BASIC	

There's also a `setHeader(String name, String value)` that overwrites existing values.

on the job

You may well wonder what happens if you use `addHeader()` to add multiple values for the same header name, and then invoke `setHeader()` on the same header name. Do all the values you added get obliterated in favor of the single value you have just added with `setHeader()`? The API documentation is silent on this point, and the answer appears to be no: `setHeader()` just replaces one of the existing values with its own setting—from my testing, the first one added with `addHeader()`. My advice would be don't use `setHeader()` for multiple value headers; use it only to replace (or add) a single value header field. Otherwise, you will confuse those maintaining your code. You can always use `HttpServletResponse.containsHeader(String headerName)` to determine if a value has been set for a given header name already—as you've probably guessed, this returns a** boolean **primitive.

Most of the art of writing headers has to do with, first, knowing the header name for your purpose and, second, providing appropriate and well-formatted values for the header name. Tables 1-2, 1-3, and 1-5 help you get started with that and give you more than sufficient guidance for the exam. There are two pairs of convenience methods when you know the value you are dealing with is either an integer or a date. Take the case of the header Retry-After—this can accept either an integer (representing a number of seconds) or a date. So you can use

```
response.addIntHeader("Retry-After", 180);
```

to add a header to tell the client to try again in 180 seconds, or

```
Date retryAfter = new Date();
long retryAfterMillis = retryAfter.getTime() + 180000;
response.addDateHeader("Retry-After", retryAfterMillis);
```

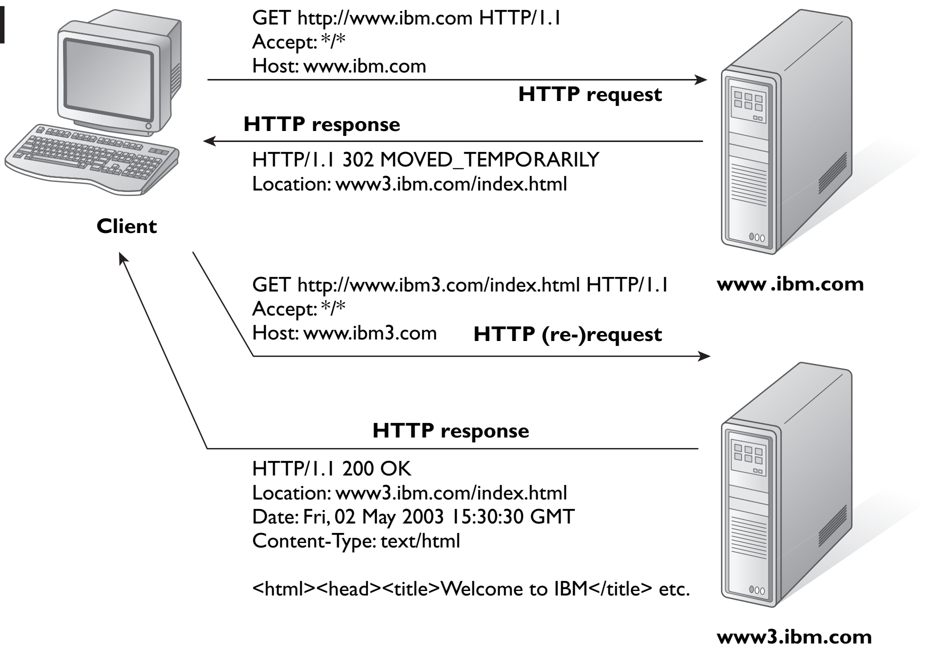
to achieve the same effect with dates. There are "set" equivalents of both the above methods to replace values instead of adding to what's already there.

Redirection

HttpServletResponse also provides other methods that don't set headers directly, but instead do this "under the covers." These may set appropriate headers and sometimes do additional work in order to get the job done. A good example is the `sendRedirect()` method. When a client makes a call to a servlet, there is the op-

FIGURE 1-8

Redirection


GET http://www.ibm.com HTTP/1.1
Accept: */*
Host: www.ibm.com
HTTP request
HTTP response
HTTP/1.1 302 MOVED_TEMPORARILY
Location: www3.ibm.com/index.html
Client
GET http://www.ibm3.com/index.html HTTP/1.1
Accept: */*
Host: www.ibm3.com
HTTP (re-)request
www .ibm.com
HTTP response
HTTP/1.1 200 OK
Location: www3.ibm.com/index.html
Date: Fri, 02 May 2003 15:30:30 GMT
Content-Type: text/html
<html><head><title>Welcome to IBM</title> etc.
www3.ibm.com

tion to send an alternative URL back—and the client will understand on receiving this URL to look there instead. It works as shown in Figure 1-8.

The redirection is achieved through the `HttpServletResponse.sendRedirect (String pathname)` method. The String parameter in this method is a path capable of conversion to a URL. This can be either relative or absolute. Let's consider a request for a servlet that has a call to this method, which looks like this:

```
http://localhost:8080/ex0104/servlet/RedirectorServlet
```

In this example, `/servlet/RedirectorServlet` is a path to a specific servlet inside the web application `/ex0104`. The servlet container's root is `http://localhost:8080/` (if running under Tomcat, you'll find the main help page at this location).

The RedirectorServlet code might redirect to a completely different domain, with code like this:

```
response.sendRedirect("http://www.otherdomain.com/otherResource");
```

If the parameter begins with a forward slash, this is interpreted as relative to the root of the servlet container. The code might look like this:

```
response.sendRedirect("/otherResource");
```

The servlet container would translate this to the following URL:

```
http://localhost:8080/otherResource
```

If you don't lead with a forward slash, the parameter is interpreted as relative to the path in which the original resource is found. So if you asked for the following address:

```
http://localhost:8080/ex0104/servlet/RedirectorServlet
```

and the code of RedirectorServlet had the following redirection request:

```
response.sendRedirect("subservlet/OtherPlace");
```

then the server would assume that the search for this resource should begin with the same path that found RedirectorServlet and tell the client to redirect to

```
http://localhost:8080/ex0104/servlet/subservlet/OtherPlace
```

exam watch

Redirection achieves roughly the same effect as forwarding, which we meet later. Both effectively divert the request and cause a different resource to be served back to the client. The big difference is that `sendRedirect()` actually sends a message back to the client, so the client makes a re-request for the actual resource required. With forwarding, all the action stays on the server.

What has all this to do with response headers? Well, the call to **`sendRedirect()`** causes the servlet container to fill out a "Location" header with the (full) alternative URL, according to the rules given above. But this isn't quite enough by itself. The servlet container must also tell the client that the resource has been moved temporarily. In technical terms, this is achieved by setting a response status code with the **`setStatus()`** method on response. If you wanted to achieve the same effect as a **`sendRedirect()`**, you could execute code such as the following:

```
response.setStatus(HttpServletResponse.SC_TEMPORARY_REDIRECT);
response.setHeader("Location", "http://www.osborne.com/index.jsp");
```

HttpServletResponse comes loaded with a set of constant values (public static final int variables) to represent different possible values for the response status code.

Setting Cookies on the Response As we learned earlier, cookies are small pieces of textual information sent between server and client. In this case, we're interested in attaching cookies to a response for the browser to receive them—and if not refused, store them. A browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each. The rules on cookie creation and attributes are covered in the earlier section on getting cookies from the request. To attach a cookie to the response, simply invoke the `addCookie(Cookie cookie)` method as many times as you need to add cookies. There is no "remove cookie" method—though `addCookie()` is a convenience method to add a header with the name "Set-Cookie" and a correctly formatted cookie content. In theory, you could use `setHeader("Set-Cookie", newValue)` to change the value of an existing cookie, but then you would have to do the formatting of the value yourself as a String.

Sending Back Content in the Response As you may well ask at this point, "Setting headers and cookies is all very well, but how do I simply send back stuff in the HTTP response message?" We couldn't avoid introducing part of this in the exercises, but now it's time to look at the topic in a little more detail.

The first decision you have to (well, should) make is this: What kind of information am I sending back to the client? This means setting the correct content type with the `ServletResponse.setContentType(String mimeType)` method. Although there's no validation as such on the parameter passed in (beyond the fact that it has to be a String), you should provide information that a browser will understand, in the form of a registered MIME type. MIME stands for Multipurpose Internet Mail Extension, but despite the "mail" component in the acronym, it has become accepted as the universal standard for describing formats (primarily, file formats) for transmission through any Internet protocol (not just e-mail). You can easily find a list of allowed MIME types on the Internet: A good one is at ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types. One of the most common is "text/html," which indicates text using HTML markup. The MIME type "text/plain" denotes text with no markup at all. Generally, a MIME type consists of a top-level classification (text, image, application), followed by a slash, followed by a subclassification (often represented as the typical file extension).

Content type is one thing. The exact encoding you are using for your files is another. This could range from simple ASCII through to full Unicode, with plenty of variants in between. You can, in fact, set the encoding along with the content type as part of the String passed to `setContentType()`. The API documentation tells you how. You can set the encoding separately through the `setCharacterEncoding()`. You don't necessarily have to do this: Often, you can get away with whatever default encoding is supplied on your server (probably ISO-8859-1).

It’s all very well defining the type of your content, but at some point you have to produce the content. Your first decision is whether the content is character-based or byte-based, for you have both a PrintWriter and an OutputStream associated with the response, but you can’t use both at once. To get hold of the PrintWriter, all you have to do is execute

```
PrintWriter out = response.getWriter();
```

You know this already, as you have had to use this code from the first servlet you wrote in Exercise 1-2. You then have access to all the methods that java.io.Print Writer allows, though the most convenient is undoubtedly the overloaded `out.write()` method, to which you typically pass a String. This means that if your MIME type is “text/html,” you can start writing (preferably well-formed) HTML syntax directly to the Writer—as we have been doing throughout. Writing HTML in Java code seems bizarre when first encountered, but you quickly get used to it, and it’s a fine approach for simple, dynamic web pages. You have the option of flushing the PrintWriter (`out.flush()`), which is a pretty good idea if you want to make sure that all the output written has been committed to the response. You can even close the PrintWriter (`out.close()`), though this may not be a good idea unless you know that your servlet is the last thing that will contribute to the composition of the response (as you’ll learn later, a servlet may be one small part in a chain of other servlets and/or filters).

If it’s a binary file format you want to send in your response, then

```
OutputStream out = response.getOutputStream()
```

is the right choice for you. You’ll be expected to know the range of methods for this class and the techniques you learned for the SCJP in mastering the java.io package. At its most basic, you can write each byte individually with out.write (int byteToWrite).

EXERCISE 1-4

ON THE CD

Using HttpServletResponse

You’ll write two servlets in this exercise: one to return a binary file to the requester, and another to simply redirect to this image-loading servlet. The unique directory for this exercise is ex0104, and the solution is on the CD in file sourcecode/ch01/ex0104.war.

Write the ImageLoader Servlet

1. You need to create a source file called ImageLoader.java, with a package of webcert.ch01.ex0104. Place this in an appropriate package directory (webcert/ch01/ex0104) within the exercise subdirectory ex0104/WEB-INF/classes.
2. You'll need a gif image—any image file will do—placed directly in directory ex0104.
3. Following the package statement, your servlet code should import packages java.io, javax.servlet, and javax.servlet.http.
4. Your ImageLoader class should extend HttpServlet (and doesn't need to implement any interfaces).
5. Override the `doGet()` method. (Refer back to previous exercises if the signature isn't yet familiar.)
6. Set the response content type to "image/gif."
7. Obtain the full path to the example image file with the following line of code (we'll learn more about ServletContext APIs later in the book):

```
String path = getServletContext().getRealPath("exampleimage.gif");
```

8. Use the path so obtained to create a File object.
9. Set the response content length to the size of the File.
10. Wrapper the File object in a FileInputStream, and wrapper that in a Buffered InputStream.
11. Obtain the OutputStream from the response.
12. Write the contents of the BufferedInputStream to the response's Output Stream.
13. Compile the servlet to the same directory as the source.

Write the Redirector Servlet

14. You need to create a source file called Redirector.java, with a package of webcert.ch01.ex0104. Place this in an appropriate package directory (webcert/ch01/ex0104) within the exercise subdirectory ex0104/WEB-INF/classes.
15. Following the package statement, your servlet code should import packages java.io, javax.servlet, and javax.servlet.http.

16. Your Redirector class should extend HttpServlet (and doesn't need to implement any interfaces).
17. Override the `doGet()` method. (Refer back to previous exercises if the signature isn't yet familiar.)
18. Have the servlet accept a parameter called "location"—store the value in a String.
19. Pass the location String as a parameter into the response's redirection method.
20. Compile the servlet in the same directory as the source.

Write redirect.html

21. Create an HTML file called redirect.html directly in directory ex0104.
22. Create a form within the file, whose action is "Redirector" and whose method is GET (you can leave this out—as then the default method for the form will be GET).
23. Create an input text field within the form named "location." You'll use this to type the URL to redirect to (so make it a reasonable size).
24. Create a submit button within the form.

Run the Code

25. Start the Tomcat Server (if not started already).
26. Deploy the *solution* code WAR file, ex0104.war (follow the instructions for deploying WAR files in Appendix B).
27. Copy redirect.html and your image file from your directory structure to the <Tomcat Installation Directory>/webapps/ex0104 directory (overwrite solution versions).
28. Copy ImageLoader.class and Redirector.class from your directory structure to <Tomcat Installation Directory>/webapps/ex0104/WEB-INF/classes/webcert/ch01/ex0104 (overwrite the solution versions).
29. Point your browser to the appropriate URL. For a default Tomcat installation, this will be

```
http://localhost:8080/ex0104/redirect.html
```

30. Type in ImageLoader into the text field, for this is the servlet you want to redirect to. Your image should load into the browser.

CERTIFICATION OBJECTIVE

Servlet Life Cycle (Exam Objective 1.4)

Describe the purpose and event sequence of the servlet life cycle: (1) servlet class loading, (2) servlet instantiation, (3) call the init method, (4) call the service method, and (5) call destroy method.

Now that we've seen some of the practicalities of servlets — responding to requests and supplying responses — we'll throw the net a bit wider in the next examination objective. At the end of this chapter, we'll look at the entire lifespan of a servlet and see what support the servlet container is bound to provide and the rules it has to follow.

Life Cycle

So far, we've focused more or less exclusively on overridden `doXXX()` methods in our servlets. We have buried the fact that all these methods are called from the `service()` method in the parent class we override, HttpServlet. There's no need to override `service()` itself, as it already does a splendid job of converting HTTP requests into appropriate `doXXX()` method calls. However, what we do need to take note of is that the `service()` method comes at the center of the servlet life cycle — and that's true not just for HTTP, but in the plain world of GenericServlet as well.

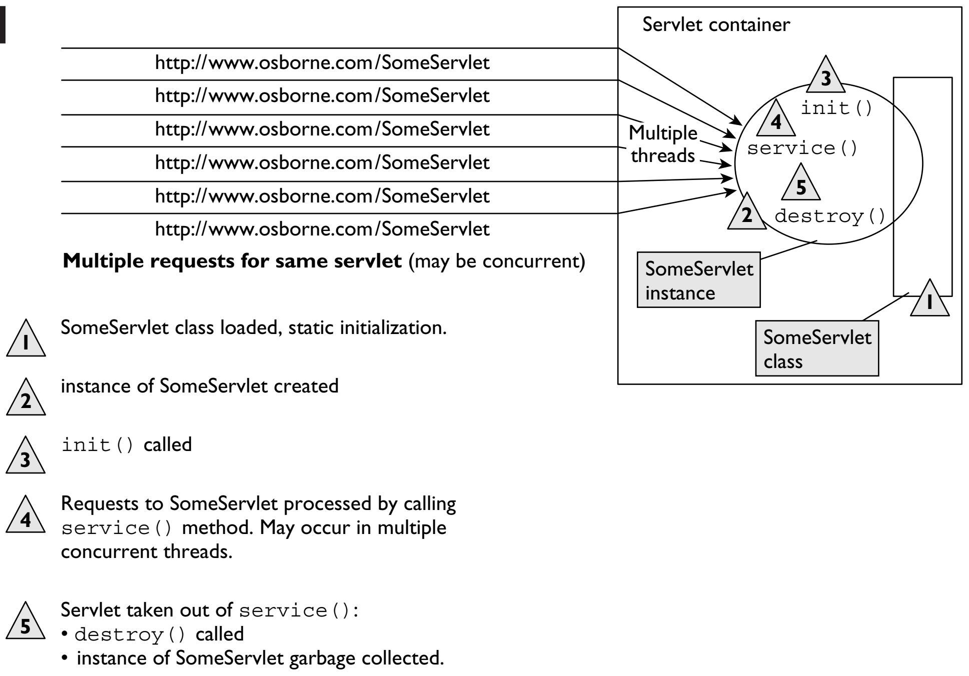
The certification objective asks you to consider the following stages of the servlet life cycle, and it is kind enough to list them in order:

1. Servlet class loading — the point where static data (if any) in your servlet are initialized
2. Servlet instantiation — the point where instance data (if any) in your servlet are initialized, and the no-argument constructor (if present) is called
3. `init()` — the initialization method called when a servlet instance is created
4. `service()` — the method called to perform the work
5. `destroy()` — the method called when a servlet is taken out of service

We need to consider in a bit more detail when each of these milestones occurs, and what they are good for. To set the stage, Figure 1-9 shows the five stages in pictures.

FIGURE 1-9

The Servlet Life Cycle


Servlet container
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
Multiple requests for same servlet (may be concurrent)
Multiple threads
3 init()
4 service()
5 destroy()
2
SomeServlet instance
1
SomeServlet class
1 SomeServlet class loaded, static initialization.
2 instance of SomeServlet created
3 init() called
4 Requests to SomeServlet processed by calling service() method. May occur in multiple concurrent threads.
5 Servlet taken out of service():
• destroy() called
• instance of SomeServlet garbage collected.

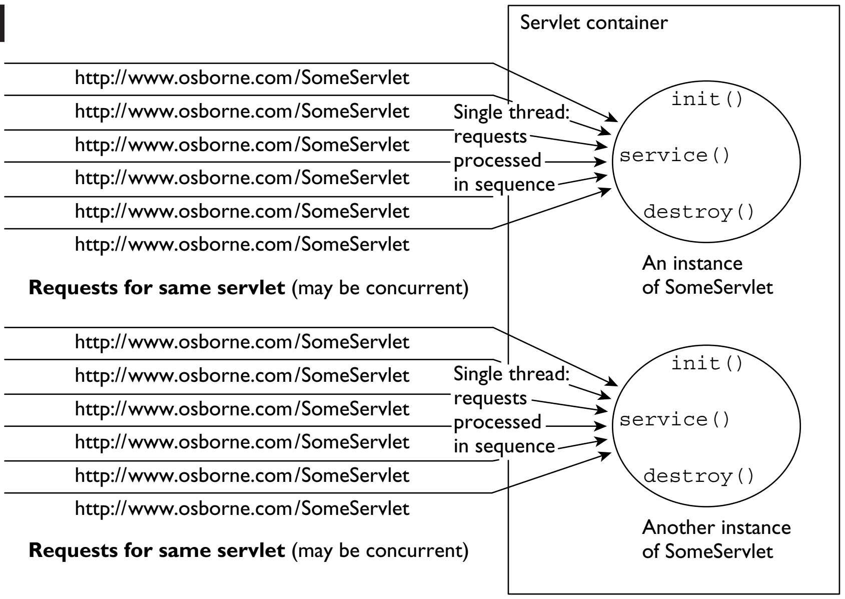
Servlet Class Loading and Servlet Instantiation

You are bound to be asked about when servlets are loaded and instantiated; it's a classic exam topic. There'll be questions that lead you astray by talking about events that are always linked to servlet creation. Actually, there is only one rule to remember: The servlet must be loaded and instantiated before the first request for it is processed. It's pretty obvious that the servlet must be there to service the request! The implication of the rule, though, is that it doesn't matter when instantiation happens. It might be "just in time"—the servlet is loaded at the point where a request comes through for it (which might lead to a performance penalty for the first user to access the servlet). Or it might happen as soon as the servlet container is started up. Or it might happen at any point in between, according to the servlet container's whim.

For any given declaration of a servlet (and what that means we learn in Chapter 2), there will—normally—be one instance. That's not to say the servlet can deal with only one request at a time. Again—normally—the same servlet instance can be used by multiple Java threads to maximize throughput. It's a very efficient model. However, it does mean that servlet instance data can be accessed by any of those threads at any time, so the best approach is to avoid using servlet instance data al-

FIGURE 1-10

The Deprecated SingleThread Model


Servlet container
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
Single thread: requests processed in sequence
init()
service()
destroy()
An instance of SomeServlet
Requests for same servlet (may be concurrent)
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
http://www.osborne.com/SomeServlet
Single thread: requests processed in sequence
init()
service()
destroy()
Another instance of SomeServlet
Requests for same servlet (may be concurrent)

together. You can get around the problem by having your servlet implement the (now deprecated) SingleThreadModel interface. It's a marker interface—no methods to implement—but it's a sign to the servlet container to ensure that any one instance of the servlet has only one request accessing it at a time. To avoid a single-instance bottleneck, servlet containers can instantiate multiple instances of a servlet (as a less efficient but bearable alternative to having multiple threads going through a single instance). Figure 1-10 shows how it looks.

But the good news (for exam purposes, especially) is that you should need to know about this interface only for historical reasons. Its use is heavily discouraged, and it's deprecated: Servlet containers probably still support it for backward compatibility, but its time has come and gone. And any mention of it has been removed from the exam syllabus.

In the next chapter, you'll learn that a servlet can be set to "load on startup." The only way in which this affects the process is that the servlet is loaded and instantiated on startup of the web application.

The `init()` Method

Initialization code for a servlet could go in the constructor. There's nothing wrong in having a zero-argument constructor for a servlet, but it's more usual to override the `public void init()` method and place initialization code there. The servlet container is guaranteed to call this method once—and only once—on instantiation of the servlet. Here are the sorts of things you might do in this method:

- Set up expensive resources (database connections, object pools, etc.)
- Do one-off initialization (such as reading configuration files into Java objects in memory)

The `init()` method must complete successfully before the servlet container will allow any requests to be processed by the servlet. Two things might go wrong. The method might throw a ServletException, or it might run out of time. Dealing with out of time first: A server should provide its own means of setting a default time beyond which `init()` is deemed to have failed. The ServletException is a little more complex. A straight ServletException is a failure, and the servlet container can abandon this instance and try to construct another. However, there is a subclass of Servlet Exception called UnavailableException, which can be constructed in two ways:

- With a message—this denotes permanent unavailability. The servlet container should log the fault and not necessarily try to create another instance. The nature of the error is likely to require some operator intervention—changing some configuration information, perhaps.
- With a message *and* a time limit (in seconds)—this denotes temporary unavailability. There's no absolute definition of what the servlet container should do under these circumstances. A sensible resolution would be to block requests to the

exam watch

You might get questioned about the parentage of the `init()` method. The `init()` method is a convenience method in the GenericServlet class. The servlet container actually calls `Servlet.init(ServletConfig config)`, passing in a servlet configuration object (we'll learn more about that in the following chapters). This method does some essential initialization of its own, including loading any initialization parameters associated with the servlet. The no-parameter `init()` saves you the bother in your servlets of overriding `init(config)` and then having to remember to call `super(config)` before adding your own initialization. The servlet container actually calls `init(config)`—which is found in GenericServlet—and this calls `init()` after completing its own work. If you have overridden the no-parameter `init()`, your version of the method will be found polymorphically.

servlet until the time limit has expired, then allow them through if the `init()` method has successfully completed.

The `service()` Method

Let's review the key points about this method, some of which we've met already:

- The method is called by the servlet container in response to a client request.
- The method is defined in the Servlet interface.
- The method is implemented in the GenericServlet and HttpServlet classes.
- The method accepts a ServletRequest and ServletResponse as parameters.
- There is an overloaded version of the method in HttpServlet that dispatches to the appropriate `doXXX()` methods. The overloaded version accepts an Http ServletRequest and HttpServletResponse as parameters.
- If there are multiple requests, these may come through on multiple threads. So multiple threads may simultaneously access the `service()` method for a single servlet instance.
- There may be no client requests at all, and that means the `service()` method may never get executed.

The `destroy()` Method

When a servlet instance is taken out of service, the `destroy()` method is called—only once for that instance. It's an opportunity to reclaim all the expensive resources that may have been set up in `init()`.

What prompts a servlet container to take a servlet out of service is, like startup, somewhat arbitrary. Clearly, this method should be called if the web application or entire web server is closed down. However, there may be more transient reasons for taking a servlet out of service: to conserve memory, for example (you get the feeling that a web server would be have to be in a pretty bad way to undertake this sort of reclamation, but it's possible).

There are some conditions that dictate whether or not `destroy()` can be called:

1. All threads executing a `service()` method on this instance must have ended—or if not ended, gone beyond a server-defined time limit.
2. `destroy()` is never called if `init()` failed. It's inappropriate to call a method to tear down what was never set up in the first place.

EXERCISE 1-5

ON THE CD

Exploring the Servlet Life Cycle

In this exercise, you'll write a servlet that tracks its own life cycle. You'll be able to tell when the class loads, when an instance is made, and when the life cycle methods are called. The unique directory for this exercise is ex0105, and the solution is on the CD in file sourcecode/ch01/ex0105.war.

Write the LifeCycle Servlet

1. Create a source file called LifeCycle.java, with a package of webcert.ch01 .ex0105. Place this in an appropriate package directory (webcert/ch01/ ex0105) within the exercise subdirectory ex0105/WEB-INF/classes.
2. Following the package statement, your servlet code should import packages java.io, javax.servlet, and javax.servlet.http. (Note: Future exercises won't mention a list of imports—you're ready to handle this yourself!)
3. Your LifeCycle class should extend HttpServlet (and doesn't need to implement any interfaces). Again, future exercises won't repeat this information.
4. You will override the `init()`, `destroy()`, and `doGet()` methods in your servlet. Also, you will supply a zero-argument public constructor.
5. In each of the above methods, output some text to indicate what method is being executed. Don't try to write this to the response (which in any case is only available in the `doGet()` method)—go for something simple, such as `System.out.println()`, to put text on the server's console. For a more advanced approach, use Java's native logging facilities.
6. Also include "static initializer" code. This is code between curly braces within the class but outside any method. It will be called at the point where the class is loaded. This code should also do a `System.out.println()` to indicate that the class is being loaded.

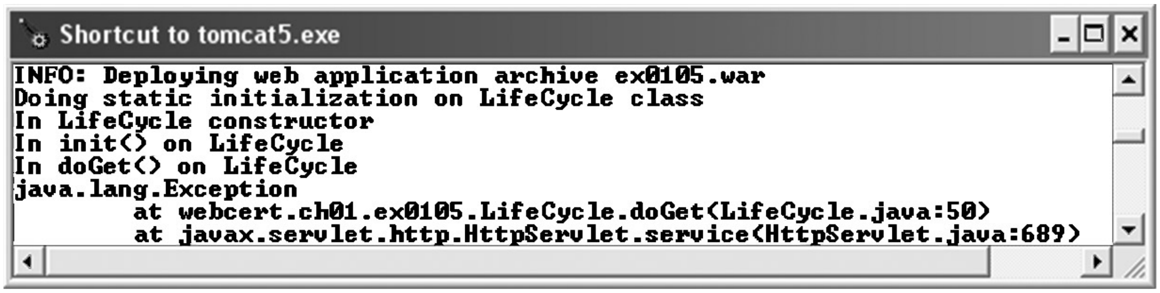
Run the Solution Code and Your Code

7. Start the Tomcat Server (if not started already).
8. Deploy the *solution* code WAR file, ex0105.war (follow the instructions for deploying WAR files in Appendix B).

9. Copy LifeCycle.class from your directory structure to <Tomcat Installation Directory>/webapps/ex0105/WEB-INF/classes/webcert/ch01/ex0105 (overwrite the solution version).
10. This time, restart (or stop and start) the Tomcat Server.
11. Check to see if there is any output from the LifeCycle servlet already on the console (is the class loaded on server startup?).
12. Point your browser to the appropriate URL for the servlet. This is likely to be

    ```
    http://localhost:8080/ex0105/LifeCycle
    ```

13. Note any additional messages on the console. Refresh the browser page a few times to ensure that you enter the `doGet()` method without going back through `init()`. Remember that the `doGet()` method is called from `service()`, in case you were wondering where `service()` fitted into the picture. My Tomcat console is shown below, from my run of the exercise (don't be alarmed by the Exception: I print a stack trace to the console specifically to show that `doGet()` is called from `service()`).



```
INFO: Deploying web application archive ex0105.war
Doing static initialization on LifeCycle class
In LifeCycle constructor
In init() on LifeCycle
In doGet() on LifeCycle
java.lang.Exception
        at webcert.ch01.ex0105.LifeCycle.doGet(LifeCycle.java:50)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:689)
```

14. Finally, recompile your version of LifeCycle.java, such that your compiled class file replaces the existing servlet class solution file LifeCycle.class. See if Tomcat picks up the change to the class file—you should see a call to the `destroy()` method as the old class is unloaded. Call your own version of LifeCycle by pointing to the appropriate URL as at step 10 above, to check that it works OK.

CERTIFICATION SUMMARY

This chapter has discussed the certification objectives that are really fundamental to the way servlets work. You started off by learning about the seven HTTP methods — GET, POST, HEAD, OPTIONS, TRACE, PUT, and DELETE. You saw that HTTP (hypertext transfer protocol) is a high-level protocol built on top of TCP/IP and used for the transfer of messages across the Internet between client and server machines.

You looked at HTTP messages in some detail and saw that they contain three components: an initial line, some headers, and an (optional) body. You learned that this structure holds true for both requests and responses, though clearly the exact content of requests and response differs a little. You learned that a request line consists of the method itself (one of the seven), the target URI, and an HTTP version. You saw a number of request headers following this request line, some typical ones being Accept (to indicate what file formats the client can cope with), User-Agent (to describe the client software), and Host (to state which server the client is targeting). You learned that the request body might contain parameters (name/value pairs) when the HTTP method is POST.

Regarding the seven methods, you saw that GET generally obtains a resource, while POST may get a resource but is more intended for sending data to the server. You learned that HEAD does everything that GET does, short of retrieving the resource itself. You saw that OPTIONS told you what you could do with a target resource. You found out that PUT and DELETE are much more rarely used, but serve to place resources at or remove resources from a given URL. You learned that POST, PUT, and DELETE are "unsafe" methods in that they can cause a permanent change to server state. Also, all the methods except POST are idempotent — that is, you can repeat them over and over again, yet expect no difference to occur just because you repeat the method call to the same URL.

Most importantly, you learned how the servlet world relates to HTTP. You saw that a J2EE servlet container will take the raw HTTP request and translate this to a method call within a servlet. A GET method is translated to a `doGet()` call, a POST to a `doPost()`, and so on. Each of these methods receives two parameters — an HttpServletRequest object (representing the HTTP request) and an HttpServlet Response object (representing the HTTP response).

From there, you went on to look at HTML forms because you need to learn how parameters generated from such forms could be made available to your servlets. You saw that the form itself began with a `<form>` opening tag, whose `action` attribute targets the servlet you want to execute. You also learned that `<form>` has an optional method attribute, which you often want to set to POST, and when left out defaults to GET. Within the opening and closing form tags, you saw that you could create several sorts of form controls. You learned that most of these are controlled by the `<input>` tag, which has numerous values for the attribute `type`. A `type` of `text`, `password` or `hidden`, creates a straight line of text—visible, concealed when typed, or hidden altogether. You saw that you are not restricted to straight text—a `type` of `checkbox` or `radio` creates tick boxes and mutually exclusive radio buttons, respectively. You learned that for form controls to be passed as parameters, they must have their `name` attribute set—and where the value isn't directly input by the user, the `value` attribute also.

You went on to learn about the `<select>` tag, which allows the user to define from a predefined list of options defined within individual `<option>` subelements. You saw that this could work in two modes—for selection of a single value or of multiple values (achieved by including the "multiple" attribute within the opening `<select>` tag). You also met the `<textarea>` element, for input of multiple lines of text in a single control.

Finally, on form controls, you saw how to create different sorts of buttons—again through use of the `<input>` element. You met the `types` of `submit`, `reset`, and `button`. As you found, pressing on a submit type actually sends the form data to the target of the action, along with the named parameters. You saw that the reset type keeps the action on the client browser and simply restores the values in form controls as they were before any user input took place. You finally saw that the button type is meant for connection with custom scripting (for example, with JavaScript) and that, again, all the action stays within the client browser.

You saw that when parameters are submitted, they are passed as name/value pairs separated by ampersands—for example, *user=David&password=revealed*. You learned that multiple parameters with the same name could be passed, simply by repeating the name with a fresh value—for example, *country=DK&country=GB*. You saw parameters passed in the URL when the HTTP method is GET and didn't see parameters passed when the HTTP method is POST, for they are then concealed in the request body.

You then met the servlet APIs for intercepting parameters. You saw that the ServletRequest object had useful methods for this—`getParameter()` to get hold of the first value from a named parameter, `getParameterValues()` to get hold of all values for a named parameter as a String array, and `getParameterNames()` to get an Enumeration of Strings with all parameter names present in the request. You also met `getParameterMap()`, which returns an Enumeration of all keys and their values. You also learned that you could get at parameters the hard way through a request's InputStream—and also never to mix and match the two approaches (APIs vs. InputStream).

You went on to look at more methods on the HttpServletRequest object, passed as the first parameter into `doXXX()` methods. This first of these concerned request headers—each a piece of information within the request consisting of a name with single or multiple values. You met APIs for getting hold of the headers within the request, such as `getHeader()` returning a single value for the supplied name. You saw that you could get all values as an Enumeration for a given header name using `getHeaders()`—and that if you needed to find out the names themselves, you could use `getHeaderNames()`, also returning an Enumeration. You saw that there were also convenience methods—the `getDateHeader()` and `getIntHeader()` methods—specifically for headers whose values are dates or integers. You then met the `getCookies()` method, which returns an array of javax.servlet.http.Cookie objects. You saw that cookies come in two versions—0 or 1—and that the only mandatory attributes in both cases are a name and a value. Other attributes include a domain, a path, a maximum age, and a comment. You learned that you could create a cookie with a two-parameter constructor, accepting the mandatory attributes of `name` and `value`, and attach this to the HttpServletResponse object with the `addCookie()` method.

There were several more things you learned about the HttpServletResponse object. You saw that just as you can read headers from HttpServletRequest, so you can write headers to HttpServletResponse. You saw that there are add* and set* methods to do this, with each call to an add* method for the same-named header placing an additional value against the header, and with each call to a set* method replacing a value already there (if any). You were warned against the use of set* methods for multiple value headers. Like the request methods, there are convenience methods for integers and dates: an `addIntHeader()`, `setIntHeader()`, `addDateHeader()`, and `setDateHeader()`.

You also learned how to achieve redirection using the `HttpServletResponse.sendRedirect()` method. You saw that this can accept a String representing a complete or partial URL—partial URLs being interpreted as relative to the web server's root (when beginning with a forward slash) and relative to the location of the redirecting resource (when not beginning with a forward slash). You learned that the server converts a partial URL to a full URL, then transmits this back to the client with an appropriate response code, so it's up to the client to re-request the suggested URL. You also learned that the mechanics of this (setting a Location header and an appropriate response code) are concealed within the `sendRedirect()` method.

Finally, you met the methods you are likely to use most often in HttpServletResponse: those to do with setting a content type and with actually writing content. You saw that you should use the `setContentType()` method to supply a suitable MIME type for the content, represented by a String such as "text/html." You learned that you can supply an optional character encoding at the same time, or set this separately. You then saw that you can obtain a PrintWriter or OutputStream from the response, using `getWriter()` and `getOutputStream()` respectively—but that you should never mix and match the two within the same response. You saw that you can use the regular java.io methods on these classes (most typically, `write()`) to place content in the response.

In the final section of the chapter, you learned about the servlet life cycle. You learned that the web container generally makes only one instance of a servlet class (though there was a mysterious sentence about "if the class is declared only once," which will make sense when we look at the deployment descriptor in Chapter 2). You learned that the web container does any static class initialization, then calls the no-argument servlet constructor (if there is a bespoke one in your servlet), and then calls the `init(ServletConfig config)` method—and you are guaranteed that the `init(ServletConfig config)` method will only ever be called once for any given instance of a servlet. You learned that this whole initialization process can take place any time before the first request to the servlet is processed. You saw then that if the servlet received any requests (and it might not), the web container calls the `service()` method—which in the case of HTTP servlets, dispatches to the appropriate `doXXX()` method. Finally, you learned that when a web container takes a servlet out of service, it calls the servlet's `destroy()` method (just once) before that instance of the servlet is garbage collected. You learned that for `destroy()` to be called, certain conditions have to be true: (1) initialization must have completed successfully, and (2) all requests against the servlet must be complete *or* some server-defined time limit must have expired.

You briefly met the SingleThreadModel, which keeps requests unique to particular instances of servlets. However, you learned that knowledge of this is required only for legacy code you maintain and that the interface has been deprecated and dropped from the exam syllabus.

✓ TWO-MINUTE DRILL

HTTP Methods

- ❏ HTTP is a simple request/response protocol underpinning most web applications on the Internet, regardless of whether they are written in Java.
- ❏ J2EE servlet containers provide a Java "superstructure" built around the HTTP protocol.
- ❏ HTTP works through seven methods, supplied with the request. These are GET, POST, HEAD, OPTIONS, TRACE, PUT, and DELETE.
- ❏ GET is used to obtain a web resource, usually in a "read only" fashion.
- ❏ POST is used typically to send data to a web server, but is also frequently used to return web resources in addition.
- ❏ HEAD is equivalent to GET except that it doesn't return the web resource—only meta-information about the resource.
- ❏ OPTIONS lists which of the seven methods can be executed against a target resource.
- ❏ TRACE is for debug purposes and reflects a client request back from the server to the client (to check how it might have changed en route).
- ❏ PUT places a resource at the URL that is the target of the HTTP request.
- ❏ DELETE does the opposite of PUT—it removes a resource from the URL that is the target of the HTTP request.
- ❏ PUT and DELETE are disallowed on most web servers. They are "unsafe" methods. POST is also deemed an "unsafe" method. "Unsafe" means that the client may be held accountable for the action.
- ❏ By contrast, GET, OPTIONS, HEAD, and TRACE are "safe" methods that should never execute anything for which the client can be held to account.
- ❏ The HTTP specification defines most methods as "idempotent," which means that if you execute them more than once, the result is the same as executing them only once (in terms of the state the web server is left in).
- ❏ Of the seven methods, POST is the only one that is not considered "idempotent."
- ❏ HTTP requests consist of a request line, some headers, and an (optional) message body.

- HTTP responses are similar: response line, some headers, and message body.
- The seven HTTP methods map on to servlet methods of the same name with a "do" in front (e.g., POST maps on to `doPost()`).
- The `doXXX()` methods receive two objects as parameters—the first representing the HTTP request and the second the HTTP response.

Form Parameters

- The primary means by which user input is made available to a web application is through an HTML form.
- An HTML `<form>` element consists of an opening and closing form tag, containing other elements representing user interface elements on the form (form controls).
- The opening `<form>` tag has two crucial attributes: *action* (used to target a resource in the web application—typically a servlet or JSP) and `method` (to denote the HTTP method).
- If the *method* attribute is left out, the default method invoked is an HTTP GET.
- It's more usual to set method="POST."
- Many form controls are created using the `<input>` element.
- The `<input>` element has three crucial attributes: *type*, *name*, and *value*.
- The type attribute determines what sort of user interface component should be drawn by the browser within the web page (e.g., text field, checkbox, radio button).
- If `<input>` elements have a `name` and `value` set, these are passed in the HTTP request as parameters, separated by an equal sign. Each name/value pair is separated from the next by an ampersand. Example*: user=david&password=indiscrete*.
- Parameters are attached to the URL when the HTTP method is GET, separated from the rest of the URL by a question mark (e.g., `http://localhost:8080/login?user=david&password=indiscrete`).
- Parameters are passed in the request body when the HTTP method is POST.
- An input element doesn't necessarily have the value attribute set when this is supplied by direct user input. A text field (`type="text"`) is a good example—by typing into the field, the browser knows to associate the value with the named form control.

- ❑ An input element of `type="text"` creates a single-line text field. Attributes can be set to limit the displayed size and maximum length of input.
- ❑ An input element of `type="password"` is exactly like a text field, except that user input is masked when typed in.
- ❑ An input element of `type="hidden"` creates a field invisible to the user but present in the HTML source. These fields are often used to hold session-dependent data in an ongoing client-server interaction. A scripting language (such as JavaScript) might also be used entirely on the client side to populate these field values.
- ❑ An input element of `type="checkbox"` creates a selectable box.
- ❑ If the user checks the box, the name/value pair defined for the checkbox is sent as a parameter.
- ❑ An input element of `type="radio"` creates a radio button. Two or more radio buttons sharing the same name will be mutually exclusive (only one can be selected from the group).
- ❑ The chosen radio button has its name/value pair sent as a parameter.
- ❑ Descriptive text for checkboxes and radio buttons is supplied separately, usually adjacent to the input element (but not an intrinsic part—there is no `description` attribute).
- ❑ An input element of `type="submit"` creates a button that triggers the action on the form, sending all parameter data to the server in an HTTP request.
- ❑ An input element of `type="reset"` creates a button that returns the HTML form to the state before any user input took place. Nothing is sent to the server; it's a client-side action.
- ❑ An input element of `type="button"` creates a button that can trigger client-side script.
- ❑ Other form controls include the `<select>` and `<textarea>` elements.
- ❑ The `<select>` element allows definition of a predefined list of options.
- ❑ By default, the user can choose one of the options, but the inclusion of the `multiple` attribute allows selection of more than one.
- ❑ The `<select>` element should include a name attribute (the name of the parameter—or parameters for multiple selections—passed to the web server).

- ❑ The `<select>` element can restrict the number of visible rows using a *size* attribute (e.g., `size="3"`). Remaining rows are generally accessible with a scroll bar.
- ❑ The `<select>` element contains `<option>` elements, which should have a *value* attribute included (for the value of the parameter passed back to the server—if this item in the list is selected).
- ❑ The `<textarea>` element has a `name` attribute, whose function is the same as for other form controls.
- ❑ Generally, you should define `rows` and `cols` attributes, which define the number of visible rows and columns (imposing—usually—scrollbars for rows and wrap-around for columns).
- ❑ Text typed into the text area is passed back as the *value* parameter. This may contain special characters to denote white space, tabs, line feeds, and carriage returns.
- ❑ On the receiving end, a servlet can use a number of methods through the interface ServletRequest to get at request parameters.
- ❑ The simplest is `getParameter(String parmName)`, which returns the (first) value for a given parameter name.
- ❑ `getParameterValues(String parmName)` returns all values for a given parameter name—as a String array (**null** if no values present).
- ❑ `getParameterNames()` returns an Enumeration of String objects containing all parameter names. The Enumeration will never be **null**, but there may be no Strings within it.
- ❑ Finally, `getParameterMap()` returns a java.util.Map object with all the parameter names (the keys of the Map, of type String) and all the parameter values (the values in the Map, of type String array).
- ❑ You can get at parameters directly through the Reader associated with the request (`ServletRequest.getReader()`).However, this approach should not be used in conjunction with the APIs already described—unpredictable results occur.

Requests

- ❑ Request header information can be obtained using APIs available in the HttpServletRequest interface.

- ❑ `getHeader(String headerName)` returns the (first or only) value for the specified header name. This can return **null** if the named header is not present.
- ❑ `getHeaders(String headerName)` returns an Enumeration of all values for the specified header.
- ❑ `getHeaderNames()` returns an Enumeration of all header names present in the request.
- ❑ The Enumerations returned by the above two methods can be empty but are—in general—never **null**, unless the web container imposes security restrictions on the availability of some or all request headers.
- ❑ `getIntHeader(String headerName)` returns a primitive **int** for the specified header name.
- ❑ If the requested header isn't present, this method returns −1.
- ❑ If the request header is present but isn't numeric, this method throws a NumberFormatException.
- ❑ `getDateHeader(String headerName)` returns a **long** representing a date for the specified header name.
- ❑ If the requested header isn't present, this method returns −1.
- ❑ If the requested header is present but can't be interpreted as a date, this method throws an IllegalArgumentException.
- ❑ Common request headers are Accept, Accept-Language, Host (compulsory in HTTP version 1.1), and User-Agent.
- ❑ There are other headers used in the request that aren't tied to request messages. They may be generally used in HTTP messages or describe the entity attached to either a request or a response.
- ❑ Cookies are small text files attached as request or response headers.
- ❑ When present in the request, the HttpServletRequest method `getCookies()` can be used to return a javax.servlet.http.Cookie array.
- ❑ Cookies have two compulsory attributes: *name* and *value*.
- ❑ Optional attributes of cookies are *domain*, *path*, *comment*, *maximum age*, *version*, and a flag denoting whether the cookie has been passed over a secure protocol or not.

Responses

- ❑ HTTP responses can be manipulated using APIs in the HttpServletResponse interface.
- ❑ `addHeader(String name, String value)` adds a header of a given name and value to the response.
- ❑ If a header of that name already exists, `addHeader()` simply interprets this as an additional value to add to the existing response header.
- ❑ `setHeader(String name, String value)` also adds a header of a given name and value to the response; however, it replaces a value already given if the header is already present.
- ❑ `addIntHeader(String name, int value)` can be used to add a header whose value is known to be an integer.
- ❑ `addDateHeader(String name, long value)` can be used to add a header whose value is known to be a date.
- ❑ There are `setIntHeader()` and `setDateHeader()` counterpart methods.
- ❑ Common response headers include Date (the date the message was returned), and most usually have to do with the entity returned, such as Content-Type, Content-Encoding, and Content-Language.
- ❑ HttpServletResponse has some convenience methods that mask the underlying setting of response headers. These include `sendRedirect(String path)`, which sets the Location header with an alternative URL and sets a response code to tell the client to make a request to the alternative URL.
- ❑ `addCookie(Cookie aCookie)` is used to attach a Cookie to the response.
- ❑ Under the covers, a Set-Cookie response header is written.
- ❑ `setContentType(String MIMEtype)` sets the Content-Type header to an appropriate value (which should be chosen from the list of defined MIME types—such as "text/html").
- ❑ Content itself is written to the response through a PrintWriter (for characters) or OutputStream (for binary data).
- ❑ Either (but not both) can be obtained using the HttpServletResponse methods `getWriter()` or `getOutputStream()`.

Servlet Life Cycle

- ❑ Servlets can be instantiated at any point before processing their first request (server startup, at the point where the first request is received, or somewhere in between).
- ❑ In general, there is only one instance of a servlet per servlet class.
- ❑ The same servlet class can have more than one declaration in the deployment descriptor; in that case, the servlet may have one instance per declaration.
- ❑ Multiple request threads may access this one instance (hence, servlets are not thread-safe).
- ❑ Servlets have their own creation and instantiation rules over and above obeying standard rules for Java class loading and object instantiation.
- ❑ First, the class is loaded, and any static initialization is done (obviously not repeated for any subsequent servlet instance—if there is one).
- ❑ Second, the no-argument constructor on the servlet is called (you can supply one in your own servlets).
- ❑ Third, the `init(ServletConfig config)` method is called—once only for the given instance.
- ❑ In general, you should place initialization code in the `init(ServletConfig config)` method rather than in the constructor.
- ❑ Now the servlet is instantiated, its `service()` method is called for every request made—probably in multiple threads if there are many concurrent requests.
- ❑ `service()` dispatches to `doXXX()` methods in HTTP servlet implementations—there is no need to override `service()` in HttpServlet.
- ❑ `service()` may never be called; there may be no requests.
- ❑ Servlets can be taken out of service at any point the web server sees fit (closed down, running out of memory, not used for a long time, . . .).
- ❑ The servlet container must wait for all threads running `service()` methods on a servlet to complete before taking the servlet out of service.
- ❑ The servlet container can impose a time limit on this waiting period.
- ❑ Before the servlet container takes a servlet out of service, it must call the servlet's `destroy()` method.

- ❑ The `destroy()` method can be used to cleanly close down expensive resources probably initialized in the servlet's `init()` method.
- ❑ `destroy()` is never called if the servlet fails to initialize.
- ❑ Failure to initialize is denoted by `init()` not completing properly.
- ❑ `init()` may throw a ServletException to denote not completing properly.
- ❑ `init()` may throw a subclass of ServletException called Unavailable Exception.
- ❑ UnavailableExceptions can be temporary or permanent; the servlet container has the right to treat these differently or treat everything as a permanent error.
- ❑ `init()` may simply run out of time (as determined by a web server specific configuration parameter).
- ❑ With most of the preceding failures of `init()` the servlet container dereferences the failing servlet instance and allows it to be garbage collected.
- ❑ If the failure is not permanent (i.e., a temporary UnavailableException), the servlet container may try again to make another instance of the servlet.
- ❑ If the failure is permanent, the servlet container must return an HTTP 404 (page not found) error.

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.

HTTP Methods

1. Which of the HTTP methods below is not considered to be "idempotent"? (Choose one.)

   A. GET
   B. TRACE
   C. POST
   D. HEAD
   E. OPTIONS
   F. SERVICE

2. Which of the HTTP methods below are likely to change state on the web server? (Choose three.)

   A. DELETE
   B. TRACE
   C. OPTIONS
   D. POST
   E. PUT
   F. CONNECT
   G. HEAD
   H. SERVICE

3. Which of the following are valid servlet methods that match up with HTTP methods? (Choose four.)

   A. `doGet()`
   B. `doPost()`
   C. `doConnect()`
   D. `doOptions()`
   E. `doHead()`
   F. `doRequest()`
   G. `doService()`

4. What is the likely effect of calling a servlet with the POST HTTP method if that servlet does not have a `doPost()` method? (Choose one.)

   A. If the servlet has a `doGet()` method, it executes that instead.

   B. 404 response code: SC_NOT_FOUND.

   C. 405 response code: SC_METHOD_NOT_ALLOWED.

   D. 500 response code: SC_INTERNAL_SERVER_ERROR.

   E. 501 response code: SC_NOT_IMPLEMENTED.

5. What is the likely effect of calling a servlet with the HEAD HTTP method if that servlet does not have a `doHead()` method? (Choose one.)

   A. 200 response code: SC_OK

   B. 404 response code: SC_NOT_FOUND

   C. 405 response code: SC_METHOD_NOT_ALLOWED

   D. 500 response code: SC_INTERNAL_SERVER_ERROR

   E. 501 response code: SC_NOT_IMPLEMENTED

Form Parameters

6. What will be the result of pressing the submit button in the following HTML form? (Choose two.)

```
<form action="/servlet/Register">
  <input type="text" name="fullName" value="Type name here" />
  <input type="submit" name="sbmButton" value="OK" />
</form>
```

   A. A request is sent with the HTTP method HEAD.

   B. A request is sent with the HTTP method POST.

   C. A request is sent with the HTTP method GET.

   D. The parameter *fullName* is the only parameter passed to the web server in the request URL.

   E. The parameter *fullName* is the only parameter passed to the web server as part of the request body.

   F. The parameters *fullName* and *sbmButton* are passed to the web server in the request URL.

   G. The parameters *fullName* and *sbmButton* are passed to the web server as part of the request body.

   H. No parameters are passed to the web server.

7. Consider the following form and servlet code. Assuming the user changes none of the default settings and presses SUBMIT, what will the servlet output in the response? (Choose one.)

```
<form action="PrintParams?param1=First" method="post">
    <input type="hidden" name="param1" value="First" />
    <input type="text" name="param1" value="Second" />
    <input type="radio" name="param1" value="Third" />
    <input type="submit" />
</form>
protected void doPost
  HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.write("<html>\n<head>\n<title>Print
    Parameters</title>\n</head>\n<body>");
  String[] param1 = request.getParameterValues("param1");
  for (int i = 0; i < param1.length; i++) {
    out.write(param1[i] + ":");
  }
  out.write("\n</body>\n</html>");
  out.close();
}
```

A. First:Second:Third

B. First:Second:Second

C. First:Third:Third

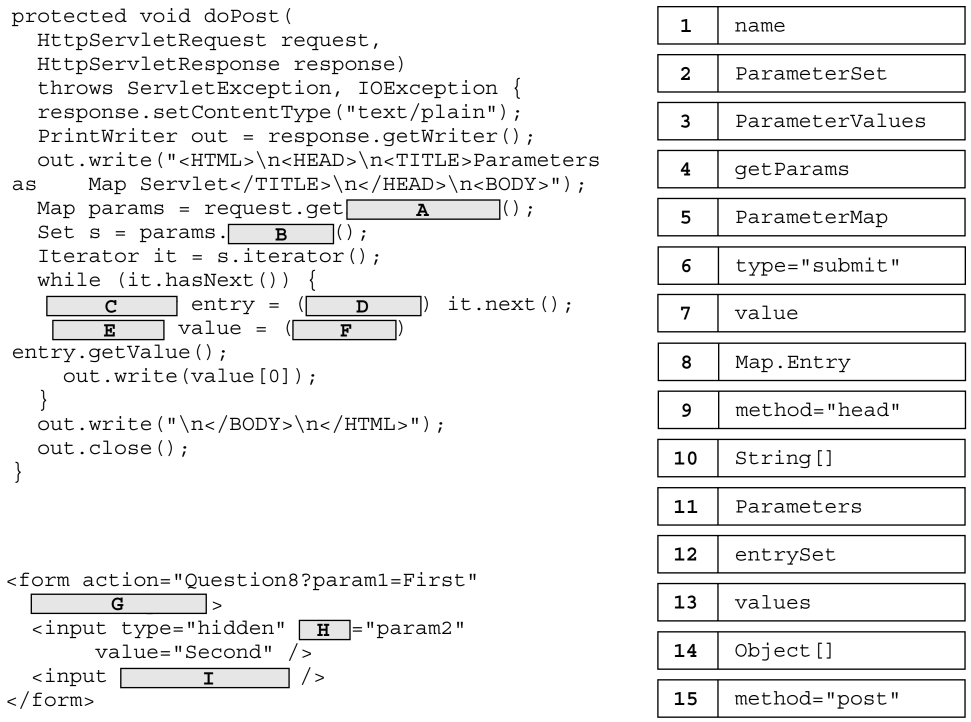
D. Second:Third:First

E. First:First:Second

F. No response—servlet will not compile.

G. No response—ServletException occurs.

8. (drag-and-drop question) The following illustration shows a form in an HTML page and also the `doPost()` method of the servlet that is the target of the form's action attribute. Match the hidden lettered values from the HTML form and the servlet code with numbers from the list on the right.



```
protected void doPost(
  HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  response.setContentType("text/plain");
  PrintWriter out = response.getWriter();
  out.write("<HTML>\n<HEAD>\n<TITLE>Parameters
as    Map Servlet</TITLE>\n</HEAD>\n<BODY>");
  Map params = request.get[     A     ]();
  Set s = params.[   B   ]();
  Iterator it = s.iterator();
  while (it.hasNext()) {
   [    C    ] entry = ([    D    ]) it.next();
   [    E    ] value = ([    F    ])
entry.getValue();
    out.write(value[0]);
  }
  out.write("\n</BODY>\n</HTML>");
  out.close();
}
```

```
<form action="Question8?param1=First"
  [      G      ]>
  <input type="hidden" [ H ]="param2"
       value="Second" />
  <input [      I      ] />
</form>
```

1	name
2	ParameterSet
3	ParameterValues
4	getParams
5	ParameterMap
6	type="submit"
7	value
8	Map.Entry
9	method="head"
10	String[]
11	Parameters
12	entrySet
13	values
14	Object[]
15	method="post"

9. What is the maximum number of parameter values that can be forwarded to the servlet from the following HTML form? (Choose one.)

```
<html>
  <body>
    <h1>Chapter 1 Question 9</h1>
    <form action="ParamsServlet" method="get">
      <select name="Languages" size="3" multiple>
        <option value="JAVA" selected>Java</option>
        <option value="CSHARP">C#</option>
        <option value="C" selected>C</option>
        <option value="CPLUSPLUS">C++</option>
        <option value="PASCAL">Pascal</option>
        <option value="ADA">Ada</option>
      </select>
      <input type="submit" name="button" />
    </form>
  </body>
</html>
```

A. 0
B. 1
C. 2
D. 3
E. 4
F. 5
G. 6
H. 7

10. What request header must be set for parameter data to be decoded from a form? (Choose one.)

A. EncType: application/x-www-urlencoded
B. Content-Type: application/x-www-form-urlencoded
C. Content-Type: multipart/form-data
D. Encoding-Type: multipart/form-data
E. Accept-Encoding: application/www-form-encoded
F. Encoding-Type: multipart/www-form-data

Requests

11. Which of the following are likely to found as request header fields? (Choose three.)

    A. Accept
    B. WWW-Authenticate
    C. Accept-Language
    D. From
    E. Client-Agent
    F. Retry-After

12. What is the most likely outcome of running the following servlet code? (Choose one.)

```
long date = request.getDateHeader("Host");
response.setContentType("text/plain");
response.getWriter().write("" + date);
```

    A. A formatted date is written to the response.
    B. −1 is written to the response.
    C. Won't run because won't compile.
    D. IllegalArgumentException
    E. NumberFormatException
    F. DateFormatException

13. What is the likely outcome of attempting to run the following servlet code?

```
String[] values = request.getHeaders("BogusHeader");
response.setContentType("text/plain");
response.getWriter().write(values[0]);
```

    A. IllegalArgumentException
    B. NumberFormatException
    C. Won't run: 1 compilation error.
    D. Won't run: 2 compilation errors.
    E. Nothing written to the response.
    F. *null* written to the response.

14. What is the likely outcome of attempting to compile and run the following servlet code, assuming there is one cookie attached to the incoming request?

```
11 Cookie[] cookies = request.getCookies();
12 Cookie cookie1 = cookies[0];
13 response.setContentType("text/plain");
14 String attributes = cookie1.getName();
15 attributes += cookie1.getValue();
16 attributes += cookie1.getDomain();
17 attributes += cookie1.getPath();
18 response.getWriter().write(attributes);
```

A. Compilation error at line 11
B. Compilation error at line 16
C. Output to the response including at least the name and domain
D. Output to the response including at least the name and value
E. Output to the response including at least the name, value, and domain
F. Output to the response including all of name, value, domain, and path

15. Under what circumstances can the `HttpServletRequest.getHeaders(String name)` method return *null*? (Choose one.)

A. If there are no request headers present in the request for the given header name.
B. If there are multiple headers for the given header name.
C. If the container disallows access to the header information.
D. If there are multiple values for the given header name.
E. If there is only a single value for the given header name.
F. There is no such method on HttpServletRequest.

Responses

16. Which of the following methods can be used to add cookies to a servlet response? (Choose two.)

A. HttpServletResponse.addCookie(Cookie cookie)
B. ServletResponse.addCookie(Cookie cookie)
C. HttpServletResponse.addCookie(String contents)
D. ServletResponse.addCookie(String contents)
E. HttpServletResponse.addHeader(String name, String value)
F. ServletResponse.addHeader(String name, String value)

17. What is the outcome of running the following servlet code? (Choose two.)

```
public void doGet(
  HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  response.setContentType("text/plain;charset-UTF-8");
  PrintWriter out = response.getWriter();
  out.flush();
  out.close();
  System.out.println(response.isCommitted());
  response.setContentType("illegal/value");
}
```

A. An IllegalArgumentException is thrown.
B. A blank page is returned to the client.
C. A 500 error is reported to the client.
D. “true” is output on the server’s console.
E. “false” is output on the server’s console.

18. What will be the outcome of executing the following code? (Choose one.)

```
public void doGet(
  HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  response.setContentType("text/plain");
  response.setContentLength(4);
  PrintWriter out = response.getWriter();
  out.write("What will be the response? ");
  out.write("" + response.isCommitted());
}
```

A. Won’t execute because of a compilation error.
B. An IllegalArgumentException is thrown.
C. An IllegalStateException is thrown.
D. A blank page is returned to the client.
E. “What” is returned to the client.
F. “What will be the response?” is returned to the client.
G. “What will be the response? true” is returned to the client.
H. “What will be the response? false” is returned to the client.

19. (drag-and-drop question) Consider the following servlet code, which downloads a binary file to the client. Match the concealed (lettered) parts of the code with the (numbered) possibilities. You may need to use some possibilities more than once.

```
public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  response.setContentType([    A    ]);
  String path =
     getServletContext().
     getRealPath("loading-msg.gif");
  File imageFile = new File(path);
  [ B ] length = imageFile.length();
  response.set[      C      ](([ D ]) length);
  OutputStream os = response.getOutputStream();
  BufferedInputStream bis = new
     BufferedInputStream(new FileInputStream(
                imageFile));
  [ E ] info;
  while ((info = bis.read()) > -1) {
    os.[ F ](info);
  }
  os.flush();
}
```

1	`byte`
2	`ContentType`
3	`short`
4	`ContentLength`
5	`"image/gif"`
6	`long`
7	`int`
8	`StreamLength`
9	`write`
10	`"image/mime"`
11	`print`
12	`ImageLength`
13	`println`
14	`Output`
15	`"text/plain"`

20. Which of the approaches below will correctly cause a client to redirect to an alternative URL? (In the code fragments below, consider that "response" is an instance of HttpServletResponse.) (Choose two.)

A. `response.sendRedirect("index.jsp");`

B. `response.setLocation("index.jsp");`

C. `RequestDispatcher rd = response.getRequestDispatcher("index.jsp");`
`rd.sendRedirect();`

D. `response.redirect("index.jsp");`

E. `response.setHeader("Location", "index.jsp");`

F. `response.setStatus(HttpServletResponse.SC_TEMPORARY_REDIRECT);`
   `response.setHeader("Location", "index.jsp");`

Servlet Life Cycle

21. Identify statements that are always true about threads running through the `service()` method of a servlet with the following class declaration. (Choose two.)

```
public class MyServlet extends HttpServlet { // servlet code }
```

A. The `destroy()` method never cuts short threads running through the `service()` method.

B. Threads running through the `service()` method must run one at a time.

C. There could be anything from zero to many threads running through the `service()` method during the time the servlet is loaded.

D. If the `init()` method for the servlet hasn't run, no threads have yet been able to run through the `service()` method.

E. At least one thread will run through the `service()` method if `init()` has been executed.

22. Under which of the following circumstances are servlets most likely to be instantiated? (Choose four.)

A. During web application startup

B. If there are insufficient instances of the servlet to service incoming requests

C. On a client first requesting the servlet

D. At the same time as a different servlet is instantiated, when that different servlet makes use of the servlet in question

E. After the servlet's `destroy()` method is called, dependent on the server's keep-alive setting

F. At some arbitrary point in the web application or application server lifetime

G. After the time specified on an UnavailableException has expired

23. Which of the following are true statements about servlet availability? (Choose two.)

A. If a servlet is removed from service, then any requests to the servlet should result in an HTTP 404 (SC_NOT_FOUND) error.

B. The `init()` method must not throw an UnavailableException.

C. If permanent unavailability is indicated via an UnavailableException, a servlet's `destroy()` method must be called.

D. Servlet containers must distinguish between periods of temporary and permanent unavailability.

E. If a servlet is deemed temporarily unavailable, a container may return an HTTP 503 (SC_SERVICE_UNAVAILABLE) message on receiving requests to the servlet during its time of unavailability.

24. Under what circumstances will a servlet instance's `destroy()` method never be called? (Choose two.)

    A. As a result of a web application closedown request

    B. When `init()` has not run to completion successfully

    C. If no thread has ever executed the `service()` method

    D. After `destroy()` has already been called

    E. During servlet replacement

25. Given the following servlet code, identify the outputs that could not or should not occur during the lifetime of the web application housing the servlet.

    A. init:destroy:

    B. init:destroy:init:destroy:

    C. init:init:init:

    D. destroy:service:

    E. init:service:service:service:service:service:

    F. init:service:init:service:

```
public class Question25 extends HttpServlet {
  public void init() {
    System.out.print("init:");
  }
  public void destroy() {
    System.out.print("destroy:");
  }
  protected void service(HttpServletRequest arg0,
     HttpServletResponse arg1)
     throws ServletException, IOException {
     super.service(arg0, arg1);
     System.out.print("service:");
  }
}
```

LAB QUESTION

Here is your turn to put together the skills you've learned in this first chapter. You'll write the servlet that you used at the start of this chapter in Exercise 1-1. Your servlet is going to do several things: (1) accept posted data from a request parameter, then (2) append this data to the end of a text file, and finally (3) read the entire text file and return this in the web page response.

Use PostServlet as the name and webcert.lab01 as the package. Use the web.xml file from the solution file, lab01.war—place this in your lab01/WEB-INF directory. To operate the servlet, write a basic HTML page with a form and an appropriate action and text field— or series of input fields, depending on how adventurous you are feeling.

This and future labs have solution code on the CD—you'll find the references to this in the Lab Answer after the Self Test Answers in each chapter.

SELF TEST ANSWERS

HTTP Methods

1. ☑ **C** is correct. Idempotent methods should behave the same however many times they are executed against a particular resource. The POST method doesn't offer that guarantee.
   ☒ **A**, **B**, **D**, and **E** are incorrect because all these methods (GET, TRACE, HEAD, and OPTIONS) are the personification of idempotency. **F** is incorrect because SERVICE is not an HTTP method at all.

2. ☑ **A**, **D**, and **E** are correct. The methods DELETE, POST, and PUT are all liable to change state on the web server—DELETE by removing a resource at the specified URL, PUT by placing one there, and POST by doing whatever it pleases.
   ☒ **B**, **C**, and **G** are incorrect because all these methods (TRACE, OPTIONS, and HEAD) are enquiry methods that should change nothing on the server. **F** and **H** are incorrect because these are not HTTP methods at all.

3. ☑ **A**, **B**, **D**, and **E** are the correct answers. `doGet()` matches HTTP GET, `doPost()` HTTP POST, `doOptions()` HTTP OPTIONS, and `doHead()` HTTP HEAD.
   ☒ **C**, **F**, and **G** are incorrect. CONNECT is a method reserved for future use in the HTTP RFC and has no servlet method counterpart. Although the concepts of request and service play a part in the servlet response to HTTP messages (you receive a request as a parameter into the `doXXX()` method family, and a servlet's `service()` method dispatches to the correct `doXXX()` method), there are no REQUEST or SERVICE HTTP methods.

4. ☑ **C** is the correct answer: 405, SC_METHOD_NOT_ALLOWED. If your servlet doesn't provide a `doPost()` method (overriding the one in HttpServlet), the default behavior is to reject the POST method request in this way.
   ☒ **A** is incorrect—there's no automatic substitution of a `doPost()` with the next-best method. **B** is incorrect, as the resource is actually there. **D** is incorrect—a 500 error is reserved for the servlet failing with some kind of exception. **E** is plausible because the method is indeed not implemented—but this isn't the response code returned. Try it!

5. ☑ **A** is correct. Your own servlet should rarely or never override the `doHead()` method—the default implementation for this is fine. The default `doHead()` method will return all response headers, but no body—and the likelihood is a normal (200) response code as well.
   ☒ **B** is incorrect because the resource is found in the scenario described. **C** is incorrect because the method is allowed through a servlet container's default implementation. **D** is incorrect in that you could get a 500 error if the servlet goes wrong—but that will be for some other

reason, not the absence of a `doHead()` method (which is what the question is driving at). Finally, **E** is incorrect — a "not implemented" error is very unlikely.

Form Parameters

6. ☑ **C** and **F**. In the absence of specifying a method parameter on the `<form>` opening tag, an HTTP GET is the default. This means that parameters from the form are passed within the query string of the URL.
☒ **A** and **B** are incorrect because an HTML's default method is GET—you are likely to explicitly specify POST instead with method="post," but this happens only if you are explicit. You would never use HEAD in a form — GET and POST are the only valid methods. **D** is incorrect because *sbmButton* is passed as a parameter as well as *fullName:* As long as a field in a form has a name, its value will be passed as a parameter (even if there may seem no point in passing the submit button as a parameter). **E** is incorrect for the same reason as **D**, and because form parameters get passed in the request body only when the method is POST. **G** is incorrect — right parameters, wrong place for them when the method is GET. Finally, **H** is incorrect because parameters are passed to the server — there's nothing in the HTML that would indicate this wouldn't happen.

7. ☑ **E** is correct. The servlet writes out the value of param1 in the query string of the form's action, then the value of param1 in the hidden field, and then the value of param1 in the text field. It doesn't print out the value of param1 in the radio field: The user doesn't select it, nor is it preselected with the "checked" attribute in the HTML.
☒ **A**, **B**, **C**, and **D** are incorrect according to the reasoning for the correct answer. The servlet compiles fine, so **F** is incorrect — and there's no reason for it to throw a ServletException; hence, **G** is incorrect.

8. ☑ **A**, **5**; **B**, **12**; **C**, **8**; **D**, **8**; **E**, **10**; **F**, **10**; **G**, **15**; **H**, **1**; and **I**, **6**. There are a few things thrown in here that rely on your knowledge of the map interface — and that's fair game for the exam. The key thing is that each entry in the map is a Map.Entry object. Beyond that, you have to know that the values part of the Map.Entry will be held as a String array, even though the parameters concerned have only a single value.
☒ All other combinations are incorrect, as dictated by the right answer above.

9. ☑ **H** is the correct answer. Seven parameter values may be returned by the form by selecting all six of the language options in the select (it's a multiple selector), and you get an additional parameter value for free with the named `<input>` submit button.
☒ **A** through **G** are incorrect, according to the reasoning in the correct answer. Issues you might have thought that limited the number of parameters: Does the size attribute for the

`<select>` limit the number of choices? No, only the number of visible rows. Does the fact that two of the options are already selected make a difference? No, the user can select all the others in addition to the ones already selected. Does the button have an associated parameter value, as it lacks a value attribute? Yes it does—because the `<input>` button has a name attribute, a default value will be passed (something like "Submit Query"—matching the default text on the button).

10. ☑ **B** is correct. You just have to know this one.
☒ **A** is incorrect—EncType is not a proper request header name, and the value is mangled. **C** is incorrect: Content-Type is the correct request header names, and *multipart/form-data* is a valid value—but you use it when posting complete data files from an HTML form rather than simple parameters. **D**, **E**, and **F** are incorrect—Encoding-Type and Accept-Encoding are correct request header names, but not applicable to this situation—and the values are mangled in different ways.

Requests

11. ☑ **A**, **C**, and **D**. Accept describes the MIME types acceptable to the client, and AcceptLanguage describes the human language preferred for the response. From has the e-mail address of the client.
☒ **B** is incorrect because WWW-Authenticate is returned as a response header, to indicate that authentication details are required from the client. **E** is incorrect—Client-Agent is made up. (User-Agent is a real request header, however, and describes the type of client making the request.) **F** is incorrect—Retry-After is returned as a response header, not a request header, to indicate that a service is unavailable and the client should retry after the suggested time or time interval.

12. ☑ **D** is the correct answer—the most likely outcome is an IllegalArgumentException. This occurs because the request header "Host" is almost certainly present in the request (it's mandatory with HTTP 1.1) but patently doesn't contain a date (it holds the domain that is the target of the request). Since the header value can't be formatted as a date, the exception results.
☒ **A** is incorrect—even if a date was returned from `getDateHeader()`, the code makes no attempt to format it. **B** is incorrect, though close—if the header requested *didn't exist*, then `getDateHeader()` would indeed return −1. The code compiles just fine, ruling out **C**. **E** is incorrect: A NumberFormatException is, however, a possible outcome from `getIntHeader()`. Finally, **F** is incorrect: There's no such thing as a DateFormatException.

13. ☑ **D** is the correct answer—there are two compilation errors, both relating to the first line of code. For one thing, `HttpServletRequest.getHeaders()` returns all request header names,

so you don't specify any parameter to the method to narrow down the range. Second, the method returns an Enumeration, not a String array.

☒ **A** and **B** are incorrect—`getHeaders()` doesn't give rise either to an IllegalArgument Exception or a NumberFormatException. **C** is wrong because there are two compilation errors, not one. **E** and **F**—to do with writing to the response—are incorrect because you'd never reach that point. Furthermore, you would need code to extract the Strings from the Enumeration returned to actually write header names to the response.

14. ☑ **D** is the correct answer—you will get output to the response including at least the name and value.

☒ **A** is incorrect—there is no compilation error in the call to `getCookies()`. You may have thought this returned an Enumeration, as do many other get* methods in the servlet API (especially when getting a plural number of things). However, a Cookie array is indeed what's returned. **B** is incorrect—you may have thought that `getDomain()` did not exist in the Cookie class, but it does. **C** is incorrect—as a cookie's value is a mandatory attribute, that should be present in the list. **E** and **F** are incorrect—both answers are perfectly possible (as domain and path, though optional attributes, are regularly set), but as statements they are not as accurate as the correct answer.

15. ☑ **C** is the correct answer—the one circumstance where `HttpServletRequest.getHeaders (String name)` would return *null* is if the servlet container disallows return of values.

☒ **A** is incorrect—if there are no request headers for the given name, this method returns an empty Enumeration. **B** is incorrect—multiple headers for the given header name don't make sense. **D** is incorrect—the whole point of the method is to return multiple values for the given header name. **E** is incorrect—if there is only a single value for the given header name, you still get an Enumeration back containing one String value. **F** is incorrect because there is such a method on HttpServletRequest—it returns all the values for a given request header name.

Responses

16. ☑ **A** and **E** are the correct answers. **A** is correct, for the `addCookie()` method is part of HttpServletResponse, not ServletResponse—and accepts a Cookie object as a parameter. You can also add cookies the hard way, using the `addHeader()` or `setHeader()` methods also on HttpServletResponse—making **E** correct as well. You pass in a header name of Set-Cookie, and a formatted String as the value, with the cookie fields formatted according to the version of the cookie standard used.

☒ **B**, **D**, and **F** are incorrect, for there are no methods on ServletResponse (the generic servlet response interface) that have to do with cookies—it's most definitely an HTTP thing. **C** is incorrect—there is no overloading on the `addCookie()` method to allow passing in of a String directly.

17. ☑ **B** and **D** are the correct answers. A blank page is written to the client (nothing is written to the PrintWriter, although it's obtained from the response); "true" is output to the console: once the PrintWriter is flushed and closed, the response is committed.
☒ **A** is incorrect, for there is no IllegalArgumentException. You can put any rubbish into the parameter of setContentType, but nothing will go wrong in the server code. (However, you may confuse the client—though again, most browsers are built to be fairly robust faced with incorrect content types. Some even ignore what's set and do their own interpretation of the response.) **C** is incorrect. You might have thought that the call to `setContentType()` after the response was committed would cause an exception resulting in a 500 error code to the client, but this call is simply ignored. Finally, **E** is incorrect—"false" isn't shown on the server console because the response has been committed at this point.

18. ☑ **E** is the correct answer: "What" is returned to the client. The response is committed once the content length (of 4 bytes) is reached.
☒ **A** is incorrect—there's nothing wrong with the code that will prevent compilation. **B** and **C** are incorrect—there's nothing in the code to cause IllegalArgumentException or Illegal StateException (you might have thought that exceeding the content length would do this, but additional content is simply ignored). **D** and **F** are wrong because the output is as described in the correct answer. **G** and **H** are wrong for the same reason, but as a point of interest, the output of `response.isCommitted()` in the output would be "true" at this point.

19. ☑ **A** matches to **5**: The content type here should be "image/gif." **B** matches to **6**: This is java.io revision—the size of a file is returned as a **long**. **C** matches to **4:** `HttpServletResponse.setContentLength()` is the method for setting the length of the response, here to match the size of the file. As this method accepts an **int**, **D** matches to **7**: We have to convert the **long** returned from inspecting the file length to an **int**. **E** also matches to **7**: A Reader reads an **int** representing a byte from the file. Finally, **F** matches to **9**: The OutputStream method is `write()` for writing an **int** (representing a byte) to a file.
☒ Other matches are wrong, according to the correct answer. It's worth mentioning that some other contenders for content type (such as "image/mime") are made up. "text/plain" exists—but we're clearly dealing with a binary file here.

20. ☑ **A** and **F** are the correct answers. **A** shows the easy way to do it: simply invoke the `sendRedirect()` method on the response. The code in answer **F** shows what `sendRedirect()` does under the covers—sets an appropriate status code, then sets the Location header with the URL to redirect to (the container must translate this to a complete URL).

☒ **B** is incorrect — there is no `setLocation()` method on HttpServletResponse. **C** is incorrect — we will encounter the RequestDispatcher class later in the book, but it has no `sendRedirect()` method, and doesn't perform redirection (as such). **D** is incorrect — the method name on HttpServletResponse is not `redirect()`; it's `sendRedirect()`. **E** is incorrect — when redirecting the manual way, it's insufficient to set the Location header only: You have to set an appropriate response status as well, as per the correct answer.

Servlet Life Cycle

21. ☑ **C** and **D** are the correct answers. There can be any number of threads active at one time in a `service()` method — it just all depends on the number of client requests and how the servlet container manages those requests.
    ☒ **A** is incorrect, though there's room for some discussion. `destroy()` should not be run until all the threads in the `service()` method end. However, there is a let-out clause that allows a servlet container to impose a time-out for threads in `service()` to end. If they are not complete by the end of this time, then `destroy()` will be called. So it's fair to say that `destroy()` should not cut short `service()` threads if at all possible, but not fair to say that it will never cut them short. **B** is incorrect — only if the servlet implements the deprecated SingleThread Model interface should threads be single-queued through an instance. We can see from the declaration that it doesn't implement this interface (nor should it, being as it's deprecated). E is incorrect — even if the servlet is initialized, there is no guarantee that any thread will run through the `service()` method. Of course, it's likely — especially if servlets are lazily initialized at the point of a user first requesting them — but it's not always true.

22. ☑ **A**, **C**, **F**, and **G** are the correct answers. A servlet is very likely to be instantiated on web application startup (especially if `<load-on-startup>` is specified — we learn about this in Chapter 2). If servlet instantiation doesn't happen, then it must happen when a client first requests a servlet. The server may also start and stop servlets on a whim (maybe for memory management reasons), so a servlet may start at a seemingly arbitrary point. Finally, some servlet containers will attempt to reinitialize a servlet that first failed with an UnavailableException (if this is of a temporary nature).
    ☒ **B** is incorrect. A servlet container just starts up more threads on the same instance. Only if the servlet container was supporting servlets implementing the deprecated SingleThreadModel interface could it possibly work as described. **D** is incorrect — there is no check on servlet dependencies at instantiation stage, and consequent loading of "chains" of servlets. **E** is incorrect — there's no such thing as a server's keep-alive setting for servlet instances.

23. ☑ **A** and **E** are the correct answers, being true statements. A servlet container should return an HTTP 404 error when a servlet is not in service, and can return an HTTP 503 error during a period of unavailability.
☒ **B** is incorrect, for an error can occur during initialization and cause an Unavailable Exception (subtype of ServletException). **C** is almost correct—if an UnavailableException occurs during a run of a servlet's `service()` method, then the `destroy()` method must be called, true enough—but not if the exception occurs in the `init()` method. Finally, **D** is incorrect: The servlet specification allows containers to treat temporary and permanent unavailability in the same way (removing a servlet from `service()`; returning HTTP 404 errors).

24. ☑ **B** and **D** are the correct answers. When `init()` does not complete successfully, the container deems this an initialization error, so it's not appropriate to call `destroy()`. Also, once `destroy()` has been called on a servlet instance, the servlet should be made available for garbage collection; hence, `destroy()` cannot be called again on the same instance.
☒ **A** is incorrect, for `destroy()` is very likely to be called on all servlet instances as a result of web application closedown. **C** is incorrect—the number of threads that have executed the `service()` method (including zero) has no bearing on whether `destroy()` is called or not. Finally, **E** is incorrect—"hot" replacement of servlet classes very often results in `destroy()` being called on the running instance, where the servlet container supports this.

25. ☑ **D** and **F** are the correct answers, for the methods should not be executed in the immediate sequence shown. In **D**, if a servlet is destroyed, then the next instance of the same servlet should not show output from the `service()` method before `init()` is called. In **F**, a "destroy:" is missing. `init()` cannot be called again on the same instance, so the implication is that the same servlet must have been taken out of service; hence, there should be a "destroy:" between the first "service:" and the second "init:."
☒ **A**, **B**, **C**, and **E** are incorrect, for they are all perfectly feasible outputs. **A** ("init:destroy:") occurs if a servlet is put into service, then taken out again at some later stage—though no requests are directed to it. **B** ("init:destroy:init:destroy") is an extension of the same idea—just that after the servlet was taken out of service, it was put back into service again. **C** ("init:init: init:") might occur if a servlet stalls on initialization and the servlet container tries to start it again. **E** ("init:service:service:service:service:service:") is a snapshot of business as usual for a servlet (prior to destruction)—initialization followed by five requests.

LAB ANSWER

You'll find the solution file lab01.war on the CD, in the /sourcecode/chapter01 directory. Deploy this according to the instructions in Appendix B. The initial web page to call is postData.html, so for me, running the Tomcat server at port 8080 on my local machine, a URL of `http://localhost:8080/`

`lab01/postData.html` works well. Enter some data in the text field, and press the “Submit Query” button. You should see what you just typed displayed on a web page. Press the “back” button on your browser, and repeat the exercise. Now you should see what you typed before as well as what you just typed on the web page. If you inspect the postData.txt file (for me, at <Tomcat Install Directory>/webapps/lab01), then all the text you typed in should also be saved there.