

## Chapter 2

# Java Servlets

In this chapter the concept of Servlets, not the entire Servlet specification, is explained; consider this an introduction to the Servlet specification starting strictly with Servlets. At times the content of this chapter may seem dry, even reminiscent of the actual specification. While an attempt is always made to liven the material up, however, there are several relevant but boring aspects of Servlet development that need to be presented now. Do attempt to read the whole chapter straight through, but also remember you can always reference this chapter when needed.

This chapter discusses the following topics:

- An explanation of what Servlets are and why you would want to use them.
- The Servlet life cycle—that is, how a container manages a Servlet.
- Building Servlets for use on the World Wide Web, which includes a review of the HTTP protocol.
- Configuring Servlets using `web.xml`.
- Coding both text-producing and non-text-producing Servlets.
- Handling HTML forms and file uploads.
- *Request dispatching*—Servlet to Servlet communication and including or forwarding to other resources in the Web Application.
- *Application context* and communicating with the container via a Servlet.
- Servlet event listeners.

## What Servlets Are and Why You Would Want to Use Them

Java Servlets are an efficient and powerful solution for creating dynamic content for the Web. Over the past few years Servlets have become the fundamental building block of mainstream server-side Java. The power behind Servlets comes from the use of Java as a platform and from interaction with a Servlet container. The Java platform provides a Servlet developer with a robust API, object-orientated programming, platform neutrality, strict types, garbage collection, and all the security features of the JVM. Complimenting this, a Servlet container provides life cycle management, a single process to share and manage application-wide resources, and interaction with a Web server. Together this functionality makes Servlets a desirable technology for server-side Java developers.

Java Servlets is currently in version 2.4 and a part of the Java 2 Enterprise Edition (J2EE). Downloads of the J2SE do not include the Servlet API, but the official Servlet API can be found on Sun Microsystems' Servlet product page, <http://java.sun.com/products/servlets>, or bundled with the Java 2 Enterprise Edition. Servlet API development is done through the Java Community Process, <http://www.jcp.org>, but the official reference implementation of the Servlet API is open source and available for public access through the Tomcat project, <http://jakarta.apache.org/tomcat>.

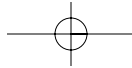
The Servlet 2.4 API includes many features that are officially defined by the Servlet 2.4 specification, <http://java.sun.com/products/servlets>, and can be broken down as follows.

### Web Applications

Servlets are always part of a larger project called a Web Application. A Web Application is a complete collection of resources for a Web site. Nothing stops a Web Application from consisting of zero, one, or multiple Servlets, but a Servlet container manages Servlets on a per Web Application basis. Web Applications and the configuration files for them are specified by the Servlet specification.

### Servlets and HTTP Servlets

The primary purpose of the Servlet specification is to define a robust mechanism for sending content to a client as defined by the Client/Server model. Servlets are most popularly used for generating dynamic content on the Web and have native support for HTTP.



## Filters

Filters were officially introduced in the Servlet 2.3 specification. A *filter* provides an abstracted method of manipulating a client's request and/or response before it actually reaches the endpoint of the request. Filters greatly complement Servlets and are commonly used for things such as authentication, content compression, and logging.

## Security

Servlets already use the security features provided by the Java Virtual Machine, but the Servlet specification also defines a mechanism for controlling access to resources in a Web Application.

## Internationalization

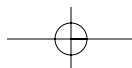
One of the best features of a Servlet is the ability to develop content for just about any language. A large part of this functionality comes directly from the Java platform's support for internationalization and localization. The Servlet API keeps this functionality and can be easily used to create content in most of the existing languages.

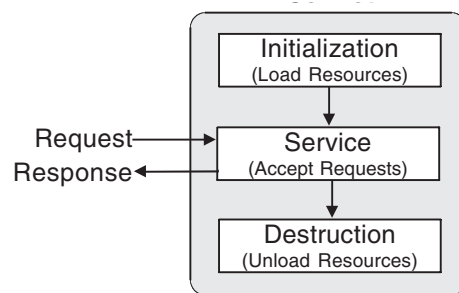
The focus of this chapter is to introduce Servlets and explain how to use HTTP Servlets for creating dynamic content on the Web. For simplicity, this chapter focuses on the basics of Servlets and leaves more complex but practical examples for discussion in pertinent, later chapters. Filters, security, and true internationalization issues are all discussed in later chapters as they pertain to both Servlets and JSP.

## Servlet Life Cycle

The key to understanding the low-level functionality of Servlets is to understand the simple life cycle they follow. This life cycle governs the multi-threaded environment that Servlets run in and provides an insight to some of the mechanisms available to a developer for sharing server-side resources. Understanding the Servlet life cycle is also the start of this book's descent to a lower level of discussion, one the majority of this book follows. Functional code examples appear often to illustrate an idea or point. Compiling and running these examples is encouraged to fully understand concepts and to familiarize yourself with Servlets for the later chapters.

The Servlet life cycle (see Figure 2-1) is the primary reason Servlets and also JSP outperform traditional CGI. Opposed to the single-use CGI life cycle,





**Figure 2-1** Diagram of the Servlet Life Cycle

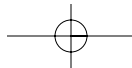
Servlets follow a three-phase life: *initialization*, *service*, and *destruction*, with initialization and destruction typically performed once, and service performed many times.

Initialization is the first phase of the Servlet life cycle and represents the creation and initialization of resources the Servlet may need to service requests. All Servlets must implement the `javax.servlet.Servlet` interface. This interface defines the `init()` method to match the initialization phase of a Servlet life cycle. When a container loads a Servlet, it invokes the `init()` method before servicing any requests.

The service phase of the Servlet life cycle represents all interactions with requests until the Servlet is destroyed. The `Servlet` interface matches the service phase of the Servlet life cycle to the `service()` method. The `service()` method of a Servlet is invoked once per a request and is responsible for generating the response to that request. The Servlet specification defines the `service()` method to take two parameters: a `javax.servlet.ServletRequest` and a `javax.servlet.ServletResponse` object. These two objects represent a client's request for the dynamic resource and the Servlet's response to the client. By default a Servlet is multi-threaded, meaning that typically only one instance of a Servlet<sup>1</sup> is loaded by a JSP container at any given time. Initialization is done once, and each request after that is handled concurrently<sup>2</sup> by threads executing the Servlet's `service()` method.

1. This description of Servlets is slightly misleading. There are many complications to do with loading Servlets that will be touched upon throughout this chapter and the rest of the book.

2. Servlets require the same state synchronization required by all multi-threaded Java objects. For simplicity, state management–related issues, including proper synchronization, are not discussed until Chapter 9. Read Chapter 9 before assuming you know everything about Servlets.



The destruction phase of the Servlet life cycle represents when a Servlet is being removed from use by a container. The `Servlet` interface defines the `destroy()` method to correspond to the destruction life cycle phase. Each time a Servlet is about to be removed from use, a container calls the `destroy()` method, allowing the Servlet to gracefully terminate and tidy up any resources it might have created. By proper use of the initialization, service, and destruction phases of the Servlet life cycle, a Servlet can efficiently manage application resources. During initialization a Servlet loads everything it needs to use for servicing requests. The resources are then readily used during the service phase and can then be cleaned up in the destruction phase.

These three events form the Servlet life cycle, but in practice there are more methods a Web developer needs to worry about. Content on the Web is primarily accessed via the HyperText Transfer Protocol (HTTP). A basic Servlet knows nothing about HTTP, but there is a special implementation of Servlet, `javax.servlet.http.HttpServlet`, that is designed especially for it.

## Servlets for the World Wide Web

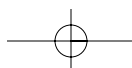
When the term Servlet is mentioned, it is almost always implied that the Servlet is an instance of `HttpServlet`<sup>3</sup>. The explanation of this is simple. The HyperText Transfer Protocol (HTTP)<sup>4</sup> is used for the vast majority of transactions on the World Wide Web—every Web page you visit is transmitted using HTTP, hence the `http://` prefix. Not that HTTP is the best protocol to ever be made, but HTTP does work and HTTP is already widely used. Servlet support for HTTP transactions comes in the form of the `javax.servlet.http.HttpServlet` class.

Before showing an example of an `HttpServlet`, it is helpful to reiterate the basics of the HyperText Transfer Protocol. Many developers do not fully understand HTTP, which is critical in order to fully understand an `HttpServlet`. HTTP is a simple, stateless protocol. The protocol relies on a client, usually a Web browser, to make a request and a server to send a response. Connections only last long enough for one transaction. A transaction can be one or more request/response pairs. For example, a browser will send a request for an HTML page followed by multiple requests for each image on that page. All of these

---

3. Note that at the time of writing there is only one protocol-specific servlet and it is HTTP. However, at least one JSR is looking to add additional protocol-specific servlets. In this particular case, it is the SIP (Session Initiation Protocol).

4. Voracious readers are advised to read the current HTTP specification, <http://www.ietf.org/rfc/rfc2616.txt>. This book is not a substitute for the complete specification. However, this book does provide more than enough detail for the average Web developer.



requests and responses will be done over the same connection. The connection will then be closed at the end of the last response. The whole process is relatively simple and occurs each time a browser requests a resource from an HTTP server<sup>5</sup>.

### Requests, Responses, and Headers

The first part of an HTTP transaction is when an HTTP client creates and sends a *request* to a server. An HTTP request in its simplest form is nothing more than a line of text specifying what resource a client would like to retrieve. The line of text is broken into three parts: the type of action, or method, that the client would like to do; the resource the client would like to access; and the version of the HTTP protocol that is being used. For example:

```
GET /index.html HTTP/1.0
```

The preceding is a completely valid HTTP request. The first word, `GET`, is a method defined by HTTP to ask a server for a specific resource; `/index.html` is the resource being requested from the server; `HTTP/1.0` is the version of HTTP that is being used. When any device using HTTP wants to get a resource from a server, it would use something similar to the above line. Go ahead and try this by hand against Tomcat. Open up a telnet session with your local computer on port 80. From the command prompt this is usually accomplished with:

```
telnet 127.0.0.1 80
```

Something similar to Figure 2-2 should appear.

The telnet program has just opened a connection to Tomcat's Web server. Tomcat understands HTTP, so type<sup>6</sup> in the example HTTP statement. This HTTP request can be terminated by a blank line, so hit Enter a second time to place an additional blank line and finish the request<sup>7</sup>.

```
GET /jspbook/index.html HTTP/1.0
```

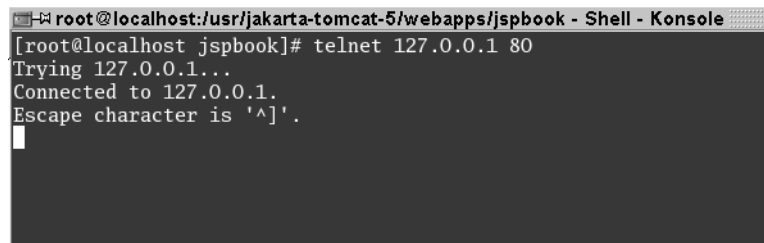
The content of `index.html` is returned from the Web Application mapped to `/jspbook` (the application we started last chapter), as shown in Figure 2-3.

---

5. HTTP 1.1 allows these “long-lived” connections automatically; in HTTP 1.0 you need to use the Connection: Keep-Alive header.

6. Microsoft's telnet input will not appear in the window as you type. To fix this, type `LOCAL_ECHO` and hit Return. Also note that if you are using Microsoft XP, then the telnet window is not cleared after it is connected.

7. If using Microsoft Window's default telnet program, be aware that the connection is *live*—that is, type in the full request correctly (even if it does not appear when you are typing it) and do not hit Backspace or Delete.



```
root@localhost:usr/jakarta-tomcat-5/webapps/jspbook - Shell - Konsole
[root@localhost jspbook]# telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

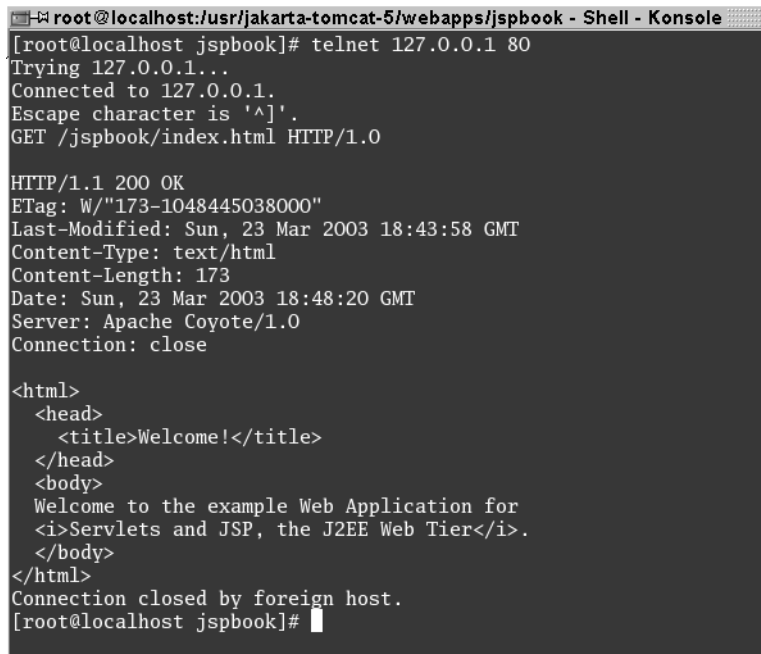
```

**Figure 2-2** Telnet to localhost:80

You just sent a basic HTTP request, and Tomcat returned an HTTP response. While usually done behind the scenes, all HTTP requests resemble the preceding. There are a few more methods to accompany GET, but before discussing those, let's take a closer look at what Tomcat sent back.

The first thing Tomcat returned was a line of text:

```
HTTP/1.1 200 OK
```



```
root@localhost:usr/jakarta-tomcat-5/webapps/jspbook - Shell - Konsole
[root@localhost jspbook]# telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /jspbook/index.html HTTP/1.0

HTTP/1.1 200 OK
ETag: W/"173-1048445038000"
Last-Modified: Sun, 23 Mar 2003 18:43:58 GMT
Content-Type: text/html
Content-Length: 173
Date: Sun, 23 Mar 2003 18:48:20 GMT
Server: Apache Coyote/1.0
Connection: close

<html>
  <head>
    <title>Welcome!</title>
  </head>
  <body>
    Welcome to the example Web Application for
    <i>Servlets and JSP, the J2EE Web Tier</i>.
  </body>
</html>
Connection closed by foreign host.
[root@localhost jspbook]#
```

**Figure 2-3** Manual HTTP Request and the Server's Response

This is an HTTP status line. Every HTTP response starts with a status line. The status line consists of the HTTP version, a status code, and a reason phrase. The HTTP response code 200 means everything was fine; that is why Tomcat included the requested content with the response. If there was some sort of issue with the request, a different response code would have been used. Another HTTP response code you are likely familiar with is the 404 “File Not Found” code. If you have ever followed a broken hyperlink, this is probably the code that was returned.

### HTTP Response Codes

In practice, you usually do not need to understand all of the specific HTTP response codes. JSP, Servlets, and Web servers usually take care of these codes automatically, but nothing stops you from sending specific HTTP response codes. Later on we will see examples of doing this with both Servlets and JSP. A complete list of HTTP response codes along with other HTTP information is available in the current HTTP specification, <http://www.ietf.org/rfc/rfc2616.txt>.

Along with the HTTP response code, Tomcat also sent back a few lines of information before the contents of `index.html`, as shown in Figure 2-4.

All of these lines are HTTP headers. HTTP uses *headers* to send meta-information with a request or response. A header is a colon-delimited name:value pair—that is, it contains the header’s name, delimited by a colon followed by the header’s value. Typical response headers include content-type descriptions, content length, a time-stamp, server information, and the date the content was last changed. This information helps a client figure out what is being sent, how big it is, and if the data are newer than a previously seen response. An HTTP request will always contain a few headers<sup>8</sup>. Common request headers consist of the user-agent details and preferred formats, languages, and content encoding to receive. These headers help tell a server what the client is and how they would prefer to get back information. Understanding HTTP headers is important, but for now put the concept on hold until you learn a little more about Servlets. HTTP headers provide some very helpful functionality, but it is better to explain them further with some `HttpServlet` examples.

---

8. There are no mandatory headers in HTTP 1.0; in HTTP 1.1 the only mandatory header is the `Host` header.



```
HTTP/1.1 200 OK
ETag: W/"173-1048445038000"
Last-Modified: Sun, 23 Mar 2003 18:43:58 GMT
Content-Type: text/html
Content-Length: 173
Date: Sun, 23 Mar 2003 18:48:20 GMT
Server: Apache Coyote/1.0
Connection: close
```

**Figure 2-4** Example HTTP Headers

## GET and POST

The first relatively widely used version of HTTP was HTTP 0.9. This had support for only one HTTP method, or verb; that was `GET`. As part of its execution, a `GET` request can provide a limited amount of information in the form of a *query string*<sup>9</sup>. However, the `GET` method is not intended to send large amounts of information. Most Web servers restrict the length of complete URLs, including query strings, to 255 characters. Excess information is usually ignored. For this reason `GET` methods are great for sending small amounts of information that you do not mind having visible in a URL. There is another restriction on `GET`; the HTTP specification defines `GET` as a “safe” method which is also idempotent<sup>10</sup>. This means that `GET` must only be used to execute queries in a Web application. `GET` must not be used to perform updates, as this breaks the HTTP specification.

To overcome these limitations, the HTTP 1.0 specification introduced the `POST` method. `POST` is similar to `GET` in that it may also have a query string, but the `POST` method can use a completely different mechanism for sending information. A `POST` sends an unlimited amount of information over a socket connection as part of the HTTP request. The extra information does not appear as part of a URL and is only sent once. For these reasons the `POST` method is usually used for sending sensitive<sup>11</sup> or large amounts of information, or when uploading files. Note that `POST` methods do not have to be idempotent. This is very important, as it now means applications have a way of updating data in a Web application. If an application needs to modify data, or add new data and is

9. A query string is a list started by a question mark, `?`, and followed by name-value pairs in the following format, `paramName=paramValue`, and with an ampersand, `&`, separating pairs, for example, `/index.html?fname=bruce&lname=wayne&password=batman`.

10. An idempotent operation is an operation that if run multiple times has no affect on state—that is, it is query only not update.

11. However, realize that the data are still visible to snoopers; it just doesn't appear in the URL.

sending a request over HTTP, then the application must not use `GET` but must instead use `POST`. Notice that `POST` requests may be idempotent; that is, there is nothing to stop an application using `POST` instead of `GET`, and this is often done when a retrieval requires sending large amounts of data<sup>12</sup>. However, note that `GET` can never be used in place of `POST` if the HTTP request is nonidempotent.

In the current HTTP version, 1.1, there are in total seven HTTP methods that exist: `GET`, `PUT`, `POST`, `TRACE`, `DELETE`, `OPTIONS`, and `HEAD`. In practice only two of these methods are used—the two we have already talked about: `GET` and `POST`.

The other five methods are not very helpful to a Web developer. The `HEAD` method requests only the headers of a response. `PUT` is used to place documents directly to a server, and `DELETE` does the exact opposite. The `TRACE` method is meant for debugging. It returns an exact copy of a request to a client. Lastly, the `OPTIONS` method is meant to ask a server what methods and other options the server supports for the requested resource.

As far as this book is concerned, the HTTP methods will not be explained further. As will soon be shown, it is not important for a Servlet developer to fully understand exactly how to construct and use all the HTTP methods manually. `HttpServlet` objects take care of low-level HTTP functionality and translate HTTP methods directly into invocations of Java methods.

## HTTP Response Codes

An HTTP server takes a request from a client and generates a response. Responses, like requests, consist of a response line, headers, and a body. The response line contains the HTTP version of the server, a response code, and a *reason phrase*. The reason phrase is some text that describes the response, and could be anything, although a recommended set of reason phrases is given in the specification. *Response codes* themselves are three-digit numbers that are divided into groups. Each group has a meaning as shown here:

- 1xx: Informational: Request received, continuing process.
- 2xx: Success: The action was successfully received, understood, and accepted.
- 3xx: Redirection: Further action must be taken in order to complete the request.

---

12. The other issue is that `GET` sends data encoded using the `application/x-www-urlencoded` MIME type. If the application needs to send data in some other format, say XML, then this cannot be done using `GET`; `POST` must be used. For example, SOAP mandates the use of `POST` for SOAP requests to cover this exact problem.

- **4xx: User-Agent Error:** The request contains bad syntax or cannot be fulfilled.
- **5xx: Server Error:** The server failed to fulfill an apparently valid request.
- **Each Status:** Code has an associated string (reason phrase).
- **The status code you'll see most often is 200.** This means that everything has succeeded and you have a valid response. The others you are likely to see are:
  - **401:** you are not authorized to make this request
  - **404:** cannot find the requested URI
  - **405:** the HTTP method you have tried to execute is not supported by this URL (e.g., you have sent a `POST` and the URL will only accept `GET`)
  - **500: Internal Server Error.** You are likely to see this if the resource to where you are browsing (such as a Servlet) throws an exception.

If you send a request to a Servlet and get a 500 code, then the chances are your Servlet has itself thrown an exception. To discover the root cause of this exception, you should check the application output logs. Tomcat's logs are stored in `/logs`<sup>13</sup> directory of the Tomcat installation.

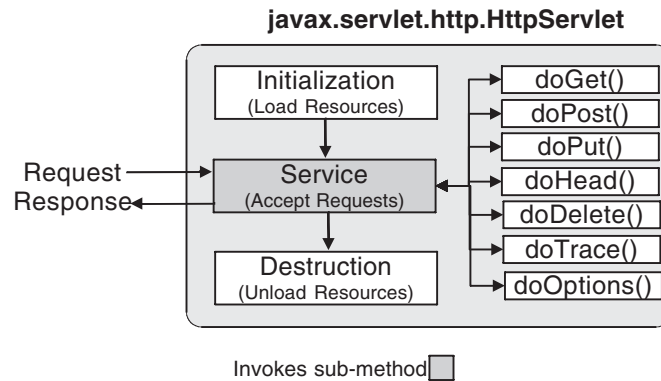
## Coding an HttpServlet

Previously, it has been shown that Servlets have a three-part life cycle: initialization, service, and destruction. An `HttpServlet` object shares this life cycle but makes a few modifications for the HTTP protocol. The `HttpServlet` object's implementation of the `service()` method, which is called during each service request, calls one of seven different helper methods. These seven methods correspond directly to the seven HTTP methods and are named as follows: `doGet()`, `doPost()`, `doPut()`, `doHead()`, `doOptions()`, `doDelete()`, and `doTrace()`. The appropriate helper method is invoked to match the type of method on a given HTTP request. The `HttpServlet` life cycle can be illustrated as shown in Figure 2-5.

While all seven methods are shown, remember that normally only one of them is called on a given request. More than one might be called if a developer

---

13. Note that you can configure Tomcat to log output to the console window. This is often done during development because it is easier to read the console than open a log file. See the Tomcat documentation if you would like to do this.



**Figure 2-5** HttpServlet Life Cycle

overrides the methods and has them call each other. The initialization and destruction stages of the Servlet life cycle are the same as described before.

Coding an `HttpServlet` is straightforward. The `javax.servlet.http.HttpServlet` class takes care of handling the redundant parts of an HTTP request and response, and requires a developer only to override methods that need to be customized. Manipulation of a given request and response is done through two objects, `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse`. Both of these objects are passed as parameters when invoking the HTTP service methods.

It is time to step through coding and using a basic Servlet. A basic “Hello World” Servlet is appropriate for getting started (see Listing 2-1). Take the following code and save it as `HelloWorld.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application.

**Listing 2-1** `HelloWorld.java`

```
package com.jspbook;

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

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
```

```
throws IOException, ServletException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html>");
    out.println("<head>");
    out.println("<title>Hello World!</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Hello World!</h1>");
    out.println("</body>");
    out.println("</html>");
}
}
```

You can probably see exactly what the preceding code is doing. If not, do not worry about understanding everything just yet since we have not learned how to deploy a Servlet for use which has to come before dissecting the code. For now understand that the preceding is the complete code for an `HttpServlet`. Once deployed, this example Servlet will generate a simple HTML page that says “Hello World!”

### Deploying a Servlet

By itself a Servlet is not a full Java application. Servlets rely on being part of a Web Application that a container manages. Using a Servlet to generate dynamic responses involves both creating the Servlet and deploying the Servlet for use in the Web Application.

Deploying a Servlet is not difficult, but it is not as intuitive as you might think. Unlike a static resource, a Servlet is not simply placed in the root directory of the Web Application. A Servlet class file goes in the `/WEB-INF/classes` directory of the application with all the other Java classes. For a client to access a Servlet, a unique URL, or set of URLs, needs to be declared in the Web Application Deployment Descriptor. The `web.xml` deployment description relies on new *elements*<sup>14</sup>: `servlet` and `servlet-mapping` need to be introduced for use in `web.xml`. The `servlet` element is used to define a Servlet that should be loaded by a Web Application. The `servlet-mapping` element is used to map a Servlet to a given URL or set of URLs. Multiple tags using either of these elements can appear to define as many Servlets and Servlet mappings as needed. Both of these elements

---

14. An element is the proper name for the unique word that comes immediately after the starting less than, “<”, of an XML tag.

also have sub-elements used to further describe them. These sub-elements are self-descriptive, and they are introduced by use in an upcoming example.

Open up the `/WEB-INF/web.xml` file of the jspbook Web Application and edit it to match Listing 2-2.

**Listing 2-2** Deploying HelloWorld Servlet

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">

  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>com.jspbook.HelloWorld</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/HelloWorld</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>welcome.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Highlighted is the new addition to `web.xml`. In the highlight, notice that both an instance of the `servlet` and `servlet-mapping` element is used. In general this is how every Servlet is deployed. A Servlet is first declared by a `servlet` element that both names the Servlet and gives the location of the appropriate Java class. After declaration, the Servlet can be referenced by the previously given name and mapped to a URL path. The name and class values are assigned by a given string in the `servlet-name` and `servlet-class` tags, respectively. The Servlet's name is arbitrary, but it must be unique from any other Servlet name for that Web Application. In the body of the `servlet-mapping` tag, the name and URL path for a Servlet are given by a string value in the body of the `servlet-name` and `url-pattern` tags, respectively. The name must match a name previously defined by a `servlet` element. The URL path can be anything as defined by the Servlet specification:

- An exact pattern to match. The pattern must start with a `/`, but can contain anything afterwards. This type of pattern is used for a one-to-one mapping of a request to a specific Servlet.
- An extension match, `*.extension`. In this case all URLs ending with the given extension are forwarded to the specified Servlet. This is

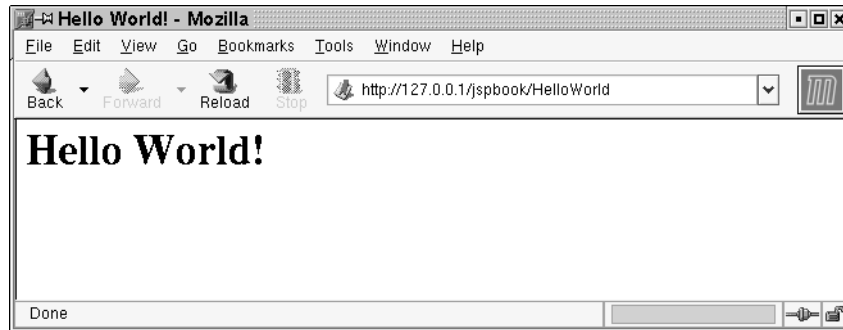


Figure 2-6 HelloWorld Servlet

commonly used in Servlet frameworks and can force many requests to go to the same Servlet<sup>15</sup>.

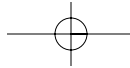
- A path mapping. Path mappings must start with a / and end with a /\*. In between anything can appear. Path mappings are usually used for forwarding all requests that fall in a certain directory to a specific Servlet.
- Default Servlet, /. A default Servlet mapping is used to define a Servlet for forwarding requests when no path information is given. This is analogous to a directory listing<sup>16</sup>.

With the HelloWorld Servlet, an exact pattern match was used that forwards any request for /HelloWorld directly to the Servlet (see Figure 2-6). Translating any of these URL patterns to a full URL involves prefixing the pattern with the URL to the Web Application. For the HelloWorld Servlet in the jspbook Web Application, this would be `http://127.0.0.1/jspbook/HelloWorld`. Restart Tomcat to update your changes and use this URL to browse to the HelloWorld Servlet<sup>17</sup>. A simple HTML page should be displayed that says “Hello World!”. For non-English readers, our apologies; internationalizing this properly would require chapter precedence of 1, 12, then 2.

15. It is used, for example, by Tomcat to map all requests to .jsp to a Servlet that knows how to process JavaServer Pages.

16. The default Servlet was used when you sent the first request to `http://localhost/jspbook`.

17. Tomcat can be configured to automatically reload a Web Application when any part of that application changes. See the Tomcat documentation for more details.



### **Understand Servlet Deployment!**

Deploying a Servlet is relatively simple but very important. Pay attention in the preceding example because for brevity further examples do not include the verbose deployment description. A single sentence such as “Deploy Servlet *x* to the URL mapping *y*” is used to mean the same thing. Only when it is exceptionally important to the example is the full deployment descriptor provided.

### **Web Application Deployment Descriptor Structure**

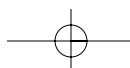
Each and every Servlet needs to be deployed before it is available for a client to use. The HelloWorld Servlet example introduced the Web Application Deployment Descriptor elements that do this, but before you go on deploying more Servlets, there is some more information to be aware of. The schema for `web.xml` defines which elements can be used and in what order they must appear. In the previous example this is the reason that both the `servlet` and `servlet-mapping` elements appeared before the `welcome-file-list` element. This is also the reason that the `servlet` element was required to appear before the `servlet-mapping` element<sup>18</sup>.

From the preceding three elements it might seem arrangement is of alphabetical precedence, but this is not the case. The arrangement of elements must match the given listing with the Web Application Deployment Descriptor schema. This rather long title should sound familiar—it is the same XML schema that defines what can appear in `web.xml`. The current complete schema can be found in the Servlet 2.4 specification. The element ordering is defined by the root `web-inf` element and is, in ascending order, as follows: `icon`, `display-name`, `description`, `distributable`, `context-param`, `filter`, `filter-mapping`, `listener`, `servlet`, `servlet-mapping`, `session-config`, `mime-mapping`, `welcome-file-list`, `error-page`, `jsp-config`, `resource-env-ref`, `message-destination-ref`, `resource-ref`, `security-constraint`, `login-config`, `security-role`, `env-entry`, `ejb-ref`, `ejb-local-ref`, `message-destination`, and `locale-encoding-mapping-list`.

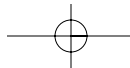
Understanding the order is not difficult, but it is a problem quite a few new Servlet developers ask about. It is well worth mentioning it now to avoid causing any confusion later. Keep in mind that this order also applies to multiple elements of the same name. If two Servlets are deployed, both of the `servlet` elements must be listed before any of the `servlet-mapping` elements. It does not

---

18. Not all Servlet containers enforce the schema, however. Consult your container’s documentation for more information.







matter what order a group of the same elements are in, but it does matter that they are properly grouped.

## Servlet Configuration

Sometimes it is necessary to provide initial configuration information for Servlets. Configuration information for a Servlet may consist of a string or a set of string values included in the Servlet's `web.xml` declaration. This functionality allows a Servlet to have initial parameters specified outside of the compiled code and changed without needing to recompile the Servlet. Each servlet has an object associated with it called the `ServletConfig`<sup>19</sup>. This object is created by the container and implements the `javax.servlet.ServletConfig` interface. It is the `ServletConfig` that contains the initialization parameters. A reference to this object can be retrieved by calling the `getServletConfig()` method. The `ServletConfig` object provides the following methods for accessing initial parameters:

```
getInitParameter(String name)
```

The `getInitParameter()` returns a `String` object that contains the value of the named initialization parameter or null if the parameter does not exist.

```
getInitParameterNames()
```

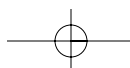
The `getInitParameterNames()` method returns the names of the Servlet's initialization parameters as an `Enumeration` of `String` objects or an empty `Enumeration` if the Servlet has no initialization parameters.

Defining initial parameters for a Servlet requires using the `init-param`, `param-name`, and `param-value` elements in `web.xml`. Each `init-param` element defines one initial parameter and must contain a parameter name and value specified by children `param-name` and `param-value` elements, respectively. A Servlet may have as many initial parameters as needed, and initial parameter information for a specific Servlet should be specified within the `servlet` element for that particular Servlet.

Using initial parameters, the `HelloWorld` Servlet can be modified to be more internationally correct. Instead of assuming the Servlet should say "Hello World!"; it will be assumed the Servlet should say the equivalent for any given language. To accomplish this, an initial parameter will be used to configure the

---

19. In fact, in the standard Servlet library a Servlet and a `ServletConfig` are the same object—that is, `GenericServlet` implements both `javax.servlet.Servlet` and `javax.servlet.ServletConfig`.



proper international “Hello” message. While `HelloWorld.java` will still not be perfectly compliant for all languages, it does demonstrate initial parameters. Modify `HelloWorld.java` to match the code in Listing 2-3.

**Listing 2-3** InternationalizedHelloWorld.java

```
package com.jspbook;

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

public class InternationalizedHelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        String greeting;
        greeting =
            getServletConfig().getInitParameter("greeting");
        out.println("<title>" +greeting+"</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>" +greeting+"</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Save the preceding code as `InternationalizedHelloWorld.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application. Since this is the second code example, a full walk-through is given for deploying the Servlet. In future examples it will be expected that you deploy Servlets on your own to a specified URL.

Open up `web.xml` in the `/WEB-INF` folder of the `jspbook` Web Application and add in a declaration and mapping to `/InternationalizedHelloWorld` for the `InternationalizedHelloWorld` Servlet. When finished, `web.xml` should match Listing 2-4.

**Listing 2-4** Updated web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">

  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>com.jspbook.HelloWorld</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>InternationalizedHelloWorld</servlet-name>
    <servlet-class>
      com.jspbook.InternationalizedHelloWorld
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>InternationalizedHelloWorld</servlet-name>
    <url-pattern>/InternationalizedHelloWorld</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/HelloWorld</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>welcome.html</welcome-file>
  </welcome-file-list>
</web-app>
```

The InternationalizedHelloWorld Servlet relies on an initial parameter for the classic “Hello World” greeting. Specify this parameter by adding in the following entry to web.xml.

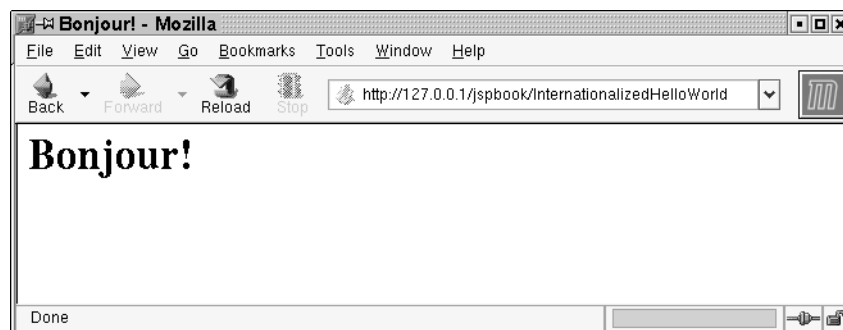
```
...
  <servlet>
    <servlet-name>InternationalizedHelloWorld</servlet-name>
    <servlet-class>
      com.jspbook.InternationalizedHelloWorld
    </servlet-class>
    <init-param>
      <param-name>greeting</param-name>
      <param-value>Bonjour!</param-value>
    </init-param>
  </servlet>
...
```

Leave the `param-name` element's body as `greeting`, but change the value specified in the body of the `param-value` tag to be a greeting of your choice. A candidate for a welcome message *en français*<sup>20</sup> would be "Bonjour!" After saving any changes, reload the `jspbook` Web Application and visit the `InternationalizedHelloWorld` Servlet to see the new message. Figure 2-7 shows an example browser rendering of `InternationalizedHelloWorld` Servlet's output.

Instead of the basic "Hello World!", the Servlet now displays the initial parameter's value. This approach is nowhere near the best of solutions for internationalization issues, but it does work in some cases and is a good example to introduce initial Servlet configuration. In general, the initial parameter mechanism shown previously is used to provide simple configuration information for an entire Web Application. The `HelloWorld` Servlet example demonstrated initial parameters for one Servlet, but later on in the chapter it will be shown that the same method is used to provide initial parameters for an entire Web Application.

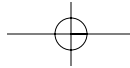
### Limitations of Configuration: `web.xml` Additions

Initial parameters are a good method of providing simple one-string values that Servlets can use to configure themselves. This approach is simple and effective, but is a limited method of configuring a Servlet. For more complex Servlets it is not uncommon to see a completely separate configuration file created to accompany `web.xml`. When developing Servlets, keep in mind that nothing stops you from doing this. If the parameter name and parameter values mappings are



**Figure 2-7** Browser Rendering of `InternationalizedHelloWorld` Servlet

20. In French.



not adequate, do not use them! It is perfectly OK to create a custom configuration file and package it in a WAR with the rest of a Web Application. A great example of doing this is shown by the Jakarta Struts framework appearing in Chapter 11. The Struts framework relies on a control Servlet that is configured via a custom and usually lengthy XML file.

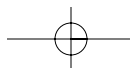
### **Client/Server Servlet Programming**

A Servlet request and response is represented by the `javax.servlet.ServletRequest` and `javax.servlet.ServletResponse` objects, or a corresponding subclass of them. For HTTP Servlets the corresponding classes are `HttpServletRequest` and `HttpServletResponse`. These two objects were quickly introduced with the HelloWorld Servlet example, but the example was primarily focused on showing how a Servlet is deployed for use. Coding and deploying are the fundamental parts of Servlet development. Deployment was explained first because it is the exact same process for any given Servlet. Once explained it is a fairly safe assumption that you can repeat the process or simply copy and edit what already exists. Servlet code varies greatly depending on what the Servlet are designed to do. Understanding and demonstrating some of the different uses of Servlets are a lot easier if time and space are not devoted to rehashing the mundane act of deployment. Servlet code is where discussion is best focused, and that is exactly what the rest of the chapter does.

Since this is a Servlet-focused book, very little time is going to be spent on discussing the normal techniques and tricks of coding with Java. Any good Java book will discuss these, and they all are valid for use with Servlets. Time is best spent focusing on the Servlet API. Understanding HTTP and the `HttpServletRequest` class is a good start, but knowledge of the `HttpServletRequest` and `HttpServletResponse` objects are needed before some useful Servlets can be built.

### **HttpServletRequest and HttpServletResponse**

The Servlet API makes manipulating an HTTP request and response pair relatively simple through use of the `HttpServletRequest` and `HttpServletResponse` objects. Both of these objects encapsulate a lot of functionality. Do not worry if it seems like this section is skimming through these two objects. Detailing all of the methods and members would be both tedious and confusing without understanding the rest of the Servlet API, but API discussion has to start somewhere and these two objects are arguably the most important. In this section discussion will only focus on a few of the most commonly used methods of each object.



Later chapters of the book cover the other methods in full and in the context of which they are best used.

## HttpServletResponse

The first and perhaps most important functionality to discuss is how to send information back to a client. As its name implies, the `HttpServletResponse` object is responsible for this functionality. By itself the `HttpServletResponse` object only produces an empty HTTP response. Sending back custom content requires using either the `getWriter()` or `getOutputStream()` method to obtain an output stream for writing content. These two methods return suitable objects for sending either text or binary content to a client, respectively. Only one of the two methods may be used with a given `HttpServletResponse` object. Attempting to call both methods causes an exception to be thrown.

With the Hello World Servlet example, Listing 2-1, the `getWriter()` method was used to get an output stream for sending the HTML markup. In the first few lines of `HelloWorld.java`, a `getWriter()` call obtained a `java.io.PrintWriter` object suitable for sending back the text.

```
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Hello World!</title>");
```

Using an instance of a `PrintWriter` object consists of providing a `String` object and calling either the `print()`, `println()`, or `write()` methods. The difference between the methods is that `println` appends a new line character, `'\n'`, to each line of response text. In both the `HelloServlet.java` code and the generated HTML page, the `println()` method was used to make the lines of HTML easy to read. As Table 2-1 shows, the HTML markup matches each `println()` call used in `HelloWorld.java`. In practice the lines of HTML will not always match up so nicely to the code in a Servlet, but the same idea is the reason `println()` is usually preferred over solely using the `print()` method. When the HTML markup does not match what is expected, it is far easier to debug by matching calls to the `println()` method.

Using a `PrintWriter` is not meant to be complex, and it should now be clear how to use the `PrintWriter` object for sending text. Above and beyond what has previously been shown is sending custom encoded text. So far only one type of text has been sent, the default text encoding of HTTP, ISO-8895-1, but changing the character encoding is possible and is covered in full in Chapter 12.

**Table 2-1** HTML Markup from HelloWorld Servlet

Generated Markup	HelloWorld.java
<code>&lt;html&gt;</code>	<code>out.println("&lt;html&gt;");</code>
<code>&lt;head&gt;</code>	<code>out.println("&lt;head&gt;");</code>
<code>&lt;title&gt;Hello World!&lt;/title&gt;</code>	<code>out.println("&lt;title&gt;Hello World!&lt;/title&gt;");</code>
<code>&lt;/head&gt;</code>	<code>out.println("&lt;/head&gt;");</code>
<code>&lt;body&gt;</code>	<code>out.println("&lt;/head&gt;");</code>
<code>&lt;h1&gt;Hello World!&lt;/h1&gt;</code>	<code>out.println("&lt;h1&gt;Hello World!&lt;/h1&gt;");</code>
<code>&lt;/body&gt;</code>	<code>out.println("&lt;/body&gt;");</code>
<code>&lt;/html&gt;</code>	<code>out.println("&lt;/html&gt;");</code>

Compared to using the `getWriter()` method, the `getOutputStream()` method is used when more control is needed over what is sent to a client. The returned `OutputStream` can be used for sending text, but is usually used for sending non-text-related binary information such as images. The reason for this is because the `getOutputStream()` method returns an instance of a `javax.servlet.ServletOutputStream` object, not a `PrintWriter`. The `ServletOutputStream` object directly inherits from `java.io.OutputStream` and allows a developer to write raw bytes. The `PrintWriter` objects lack this functionality because it always assumes you are writing text.

In most practical situations it is rarely needed to send raw bytes rather than text to a client, but this functionality is something a good Servlet developer should be aware of<sup>21</sup>. Often the incorrect mindset is to think Servlets can only send dynamically created text. By sending raw bytes, a Servlet can dynamically provide any form of digital content. The primary restriction on this functionality is being able to create the needed bytes for a desired content. For commonly used formats, including images and audio, it is not uncommon to see a Java API built to simplify the task. Combining this API with the Servlet API, it is then relatively easy to send the custom format. A good example to use would be the Java API for

21. Note that for better efficiency you may want to use the `OutputStream` rather than the `PrintWriter` to send text. The `PrintWriter` accepts Unicode strings whereas the `OutputStream` accepts bytes. See *Java Performance and Scalability Volume 1* by Dov Bulka for more details.

Advanced Imaging (JAI). Using this API many of the popular image formats can be produced from the server-side, even on servers not supporting a GUI.

Full discussion of non-text-producing Servlets is outside the scope of this book. Producing custom images, audio, and other non-text formats via Java is not something specific to Servlets. The only thing a Servlet needs to do is appropriately set a MIME type and send a client some bytes, but that is not a good reason to completely avoid an example. For completeness, Listing 2-5 provides a Servlet that dynamically generates an image and sends the bytes using a `ServletOutputStream`.

**Listing 2-5** `DynamicImage.java`

```
package com.jspbook;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.awt.*;
import java.awt.image.*;
import com.sun.image.codec.jpeg.*;

public class DynamicImage extends HttpServlet {

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

        response.setContentType("image/jpeg");

        // Create Image
        int width = 200;
        int height = 30;
        BufferedImage image = new BufferedImage(
            width, height, BufferedImage.TYPE_INT_RGB);

        // Get drawing context
        Graphics2D g = (Graphics2D) image.getGraphics();

        // Fill background
        g.setColor(Color.gray);
        g.fillRect(0, 0, width, height);
```



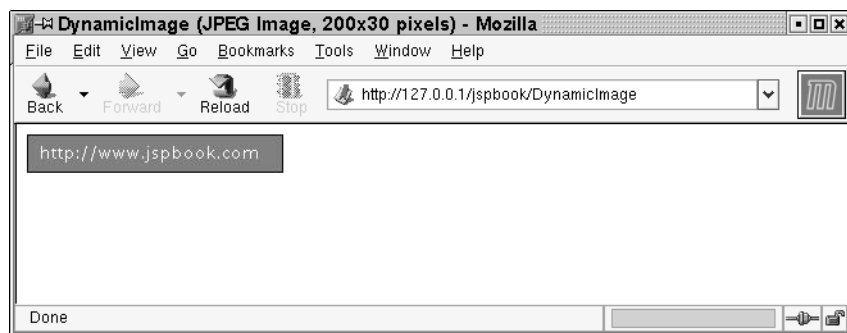
```
// Draw a string
g.setColor(Color.white);
g.setFont(new Font("Dialog", Font.PLAIN, 14));
g.drawString("http://www.jspbook.com",10,height/2+4);

// Draw a border
g.setColor(Color.black);
g.drawRect(0,0,width-1,height-1);

// Dispose context
g.dispose();

// Send back image
ServletOutputStream sos = response.getOutputStream();
JPEGImageEncoder encoder =
    JPEGCodec.createJPEGEncoder(sos);
encoder.encode(image);
}
}
```

Save the preceding code as `DynamicImage.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application. Compile and deploy the `DynamicImage` Servlet with a mapping to the `/DynamicImage` URL extension of the `jspbook` Web Application. After reloading the Web Application, browse to `http://127.0.0.1/jspbook/DynamicImage`. A JPEG formatted image is dynamically generated on each request to the Servlet. Figure 2-8 shows an example of one of the dynamically generated images.



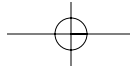
**Figure 2-8** DynamicImage Servlet

Before going out and creating your own image-producing Servlet, a fair warning should be given regarding the preceding code. For simplicity the code uses an object from the `com.sun.image.codec.jpeg` package that is unofficially included in the J2SDK 1.4. Code from the `com.sun` package is not guaranteed to be around in future Java releases, nor is it meant for developers to use. A proper solution would be to use an instance of the `ImageEncoder` class from the Java Advanced Imaging API, but that would have required you download and install the JAI before running the example.

### Response Headers

Along with sending content back to a client, the `HttpServletResponse` object is also used to manipulate the HTTP headers of a response. HTTP response headers are helpful for informing a client of information such as the type of content being sent back, how much content is being sent, and what type of server is sending the content. The `HttpServletResponse` object includes the following methods for manipulating HTTP response headers:

- **`addHeader(java.lang.String name, java.lang.String value)`:** The `addHeader()` method adds a response header with the given name and value. This method allows response headers to have multiple values.
- **`containsHeader(java.lang.String name)`:** The `containsHeader()` method returns a `boolean` indicating whether the named response header has already been set.
- **`setHeader(java.lang.String name, java.lang.String value)`:** The `setHeader()` method sets a response header with the given name and value. If the header had already been set, the new value overwrites the previous one. The `containsHeader()` method can be used to test for the presence of a header before setting its value.
- **`setIntHeader(java.lang.String name, int value)`:** The `setIntHeader()` sets a response header with the given name and integer value. If the header had already been set, the new value overwrites the previous one. The `containsHeader()` method can be used to test for the presence of a header before setting its value.
- **`setDateHeader(java.lang.String name, long date)`:** The `setDateHeader()` sets a response header with the given name and date value. The date is specified in terms of milliseconds since the epoch. If the header had already been set, the new value overwrites



the previous one. The `containsHeader()` method can be used to test for the presence of a header before setting its value.

- **`addIntHeader(java.lang.String name, int value)`:** The `addIntHeader()` method adds a response header with the given name and integer value. This method allows response headers to have multiple values.
- **`addDateHeader(java.lang.String name, long date)`:** The `addDateHeader()` method adds a response header with the given name and date value. The date is specified in terms of milliseconds since the epoch<sup>22</sup>. This method doesn't override previous response headers and allows response headers to have multiple values.

In the introduction to HTTP that appeared earlier in this chapter, a few HTTP response headers were seen, and in the HelloWorld Servlet the `Content-Type` response header was used. In both these cases, elaboration on the headers' semantics was conveniently skipped. This was done intentionally to simplify the examples, but it is time to clarify what these unexplained HTTP headers mean (see Table 2-2), along with introducing some of the other helpful headers that can be set by an `HttpServletResponse` object.

In most cases the most important header to worry about as a Servlet author is `Content-Type`. This header should always be set to `'text/html'` when a Servlet is sending back HTML. For other formats the appropriate MIME type<sup>23</sup> should be set.

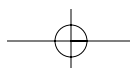
### **Response Redirection**

Any HTTP response code can be sent to a client by using the `setStatus()` method of an `HttpServletResponse` object. If everything works OK, Servlet will send back a status code 200, OK. Another helpful status code to understand is 302, "Resource Temporarily Moved". This status code informs a client that the resource they were looking for is not at the requested URL, but is instead at the URL specified by the `Location` header in the HTTP response. The 302 response code is helpful because just about every Web browser automatically follows the new link without informing a user. This allows a Servlet to take a user's request and forward it any other resource on the Web.

Because of the common implementation of the 302 response code, there is an excellent use for it besides the intended purpose. Most Web sites track where vis-

22. A common reference in time; January 1, 1970 GMT.

23. Multipart Internet Mail Extensions defined in RFCs 2045, 2046, 2047, 2048, and 2049



**Table 2-2** HTTP 1.1 Response Header Fields

Header Field	Header Value
Age	A positive integer representing the estimated amount of time since the response was generated from the server.
Location	Some HTTP response codes redirect a client to a new resource. The location of this resource is specified by the Location header as an absolute URI.
Retry-After	The Retry-After response header field can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client. The value of this field can be either a date or an integer number of seconds (in decimals) after the time of the response.
Server	The Server field is a string representing information about the server that generated this response.
Content-Length	The Content-Length entity header field indicates the size of the message body, in decimal number of octets (8-bit bytes), sent to the recipient or, in the case of the HEAD method, the size of the entity body that would have been sent had the request been a GET.
Content-Type	The MIME type that corresponds to the content of the HTTP response. This value is often used by a browser to determine if the content should be rendered internally or launched for rendering by an external application.
Date	The Date field represents the date and time at which the message was originated.
Pragma	The Pragma field is used to include implementation-specific directives that may apply to any recipient along the request-response chain. The most commonly used value is "no-cache", indicating a resource shouldn't be cached.

itors come from to get an idea of what other sites are sending traffic. The technique for accomplishing involves extracting the "referer" (note the slightly inaccurate spelling) header of an HTTP request. While this is simple, there is no equally easy way of tracking where a site sends traffic. The problem arises because any link on a site that leads to an external resource does send a request back to the site it was sent from. To solve the problem, a clever trick can be used that relies on the HTTP 302 response code. Instead of providing direct links to

external resources, encode all links to go to the same Servlet on your site but include the real link as a parameter. Link tracking is then provided using the Servlet to log the intended link while sending the client back a 302 status code along with the real link to visit.

As you might imagine, using a Servlet to track links is very commonly done by sites with HTTP-aware developers. The HTTP 302 response code is used so often it has a convenience method, `sendRedirect()`, in the `HttpServletResponse` object. The `sendRedirect()` method takes one parameter, a string representing the new URL, and automatically sets the HTTP 302 status code with appropriate headers. Using the `sendRedirect()` method and a `java.util.Hashtable`, it is easy to create a Servlet for tracking link use. Save the code in Listing 2-6 as `LinkTracker.java` in the `/WEB-INF/classes/com/jspbook` directory of the jspbook Web Application. Deploy the Servlet to the `/LinkTracker` URL mapping.

**Listing 2-6** `LinkTracker.java`

```
package com.jspbook;

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

public class LinkTracker extends HttpServlet {
    static private Hashtable links = new Hashtable();

    String tstamp;
    public LinkTracker() {
        tstamp = new Date().toString();
    }

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

        String link = request.getParameter("link");
        if (link != null && !link.equals("")) {
            synchronized (links){
                Integer count = (Integer) links.get(link);
                if (count == null) {
                    links.put(link, new Integer(1));
                }
            }
        }
    }
}
```

```
        else {
            links.put(link, new Integer(1+count.intValue()));
        }
    }
    response.sendRedirect(link);
}
else {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    request.getSession();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Links Tracker Servlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>Links Tracked Since");
    out.println(timestamp+":</p>");
    if (links.size() != 0) {
        Enumeration enum = links.keys();
        while (enum.hasMoreElements()) {
            String key = (String)enum.nextElement();
            int count = ((Integer)links.get(key)).intValue();
            out.println(key+" : "+count+" visits<br>");
        }
    }
    else {
        out.println("No links have been tracked!<br>");
    }
    out.println("</body>");
    out.println("</html>");
}
}
public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    doGet(request, response);
}
}
```

To complement the LinkTracker Servlet, some links are needed that use it. The links can be to any resource as long as they are encoded properly. Encoding the links is not difficult; it requires the real link be passed as the link parameter in a query string. Listing 2-7 is a simple HTML page that includes a few properly

encoded links. Save the HTML as `links.html` in the base directory of the jspbook Web Application.

**Listing 2-7** Some Links Encoded for the LinkTracker Servlet

```
<html>
  <head>
    <title>Some Links Tracked by the LinkTracker Servlet</title>
  </head>
  <body>
    Some good links for Servlets and JSP. Each link is directed
    through the LinkTracker Servlet. Click on a few and visit
    the <a href="LinkTracker">LinkTracker Servlet</a>.
    <ul>
      <li><a href="LinkTracker?link=http://www.jspbook.com">
        Servlets and JSP Book Support Site</a></li>
      <li><a href="LinkTracker?link=http://www.jspinsider.com">
        JSP Insider</a></li>
      <li><a href="LinkTracker?link=http://java.sun.com">
        Sun Microsystems</a></li>
    </ul>
  </body>
</html>
```

After reloading the Web Application, browse to `http://127.0.0.1/jspbook/links.html`. Figure 2-9 shows what the page looks like after rendered by a browser. Click a few times on any combination of the links.

Each link is directed through the LinkTracker Servlet, which in turn directs a browser to visit the correct link. Before each redirection the LinkTracker Servlet



**Figure 2-9** Browser Rendering of `links.html`

logs the use of the link by keying the link URL to an `Integer` object in a `Hashtable`. If you browse directly to the LinkTracker Servlet, `http://127.0.0.1/jspbook/LinkTracker`, it displays information about links visited. Figure 2-10 shows what the results look like after tracking a few links. Results are current as of the last reloading of the LinkTracker Servlet. This example does not log the information for long-term use, but nothing stops such a modification from being made.

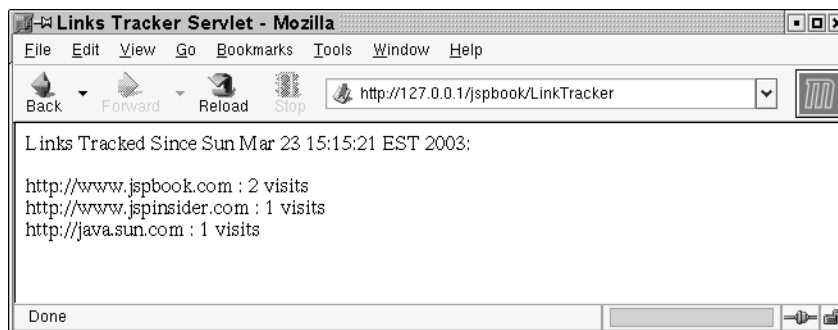
### **Response Redirection Translation Issues**

Response redirection is a good tool to be aware of and works with any implementation of the Servlet API. However, there is a specific bug that tends to arise when using relative response redirection. For instance:

```
response.sendRedirect("../foo/bar.html");
```

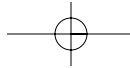
would work perfectly fine when used in some Servlets but would not in others. The trouble comes from using the relative back, `..`, to traverse back a directory. A JSP can correctly use this (assuming the browser translates the URL correctly), but the JSP can use it only if the request URL combined with the redirection ends up at the appropriate resource. For instance, if `http://127.0.0.1/foo/bar.html` is a valid URL, then `http://127.0.0.1/foo/./foo/bar.html` should also be valid. However, `http://127.0.0.1/foo/foo/./foo/bar.html` will not reach the same resource.

This may seem like an irrelevant problem, but we will soon introduce request dispatching that will make it clear why this is an issue. Request dispatching allows for requests to be forwarded on the server-side—meaning the



**Figure 2-10** Browser Rendering of Link Statistics from the LinkTracker Servlet





requested URL does not change, but the server-side resource that handles it can. Relative redirections are not always safe; “..” can be bad. The solution is to always use absolute redirections. Either use a complete URL such as:

```
response.sendRedirect("http://127.0.0.1/foo/bar.html");
```

Or use an absolute URL from the root, “/”, of the Web Application.

```
response.sendRedirect("/foo/bar.html")24;
```

In cases where the Web application can be deployed to a non-root URL, the `HttpServletRequest.getContextPath()` method should be used in conjunction:

```
response.sendRedirect(request.getContextPath()+"/foo/bar.html");
```

Further information about the `HttpServletRequest` object and use of the `getContextPath()` method is provided later in this chapter.

### **Auto-Refresh/Wait Pages**

Another response header technique that is uncommon but helpful is to send a wait page or a page that will auto-refresh to a new page after a given period of time. This tactic is helpful in any case where a response might take an uncontrollable time to generate, or for cases where you want to ensure a brief pause in a response. The entire mechanism revolves around setting the Refresh response header<sup>25</sup>. The header can be set using the following:

```
response.setHeader("Refresh", "time; URL=url" );
```

Where “time” is replaced with the amount of seconds, the page should wait, and “url” is replaced with the URL that the page should eventually load. For instance, if it was desired to load `http://127.0.0.1/foo.html` after 10 seconds of waiting, the header would be set as so:

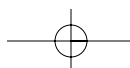
```
response.setHeader("Refresh", "10; URL=http://127.0.0.1/foo.html");
```

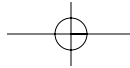
Auto-refreshing pages are helpful because they allow for a normal “pull” model, waiting for a client’s request, to “push” content. A good practical use case

---

24. Another option is to use the JavaServer Pages Standard Tag Libraries `redirect` tag. The JSTL is covered in Chapter 7.

25. The Refresh header is not part of the HTTP 1.0 or HTTP 1.1 standards. It is an extension supported by Microsoft Internet Explorer, Netscape Navigator 4.x, and Mozilla-based clients.





would be a simple your-request-is-being-processed-page that after a few seconds refreshes to show the results of the response. The alternative (also the most commonly used approach) is to wait until a request is officially finished before sending back any content. This results in a client's browser waiting for the response, sometimes appearing as if the request might time-out and resulting in the user making a time-consuming request twice<sup>26</sup>.

Another practical use case for wait page would be slowing down a request, perhaps to better ensure pertinent information is seen by the user. For example, a wait page that showed either an advertisement or legal information before redirecting to the appropriately desired page.

It should be clear that there are several situations where the Refresh response header can come in handy. While it is not a standard HTTP 1.1 header, it is something that is considered a de facto standard<sup>27</sup>.

### HttpServletRequest

A client's HTTP request is represented by an `HttpServletRequest` object. The `HttpServletRequest` object is primarily used for getting request headers, parameters, and files or data sent by a client. However, the Servlet specification enhances this object to also interact with a Web Application. Some of the most helpful features include session management and forwarding of requests between Servlets.

### Headers

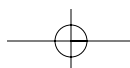
HTTP headers set by a client are used to inform a server about what software the client is using and how the client would prefer a server send back requested information. From a Servlet, HTTP request headers can be accessed by calling the following methods:

- **getHeader(java.lang.String name):** The `getHeader()` method returns the value of the specified request header as a string. If the request did not include a header of the specified name, this method returns null. The header name is case insensitive. You can use this method with any request header.

---

26. Auto-refresh pages help tremendously reduce this problem, but you should also ensure the Web application accurately maintains state. Chapter 9 thoroughly covers state management.

27. Be aware, however, that the Refresh and Redirect solutions shown here do have a downside. They both involved extra roundtrips from the client to the server. Roundtrips are expensive in terms of time and resources used, and a Web application should seek to minimize them.



- **getHeaders(java.lang.String name):** The `getHeaders()` method returns all the values of the specified request header as an Enumeration of `String` objects. Some headers, such as `Accept-Language`, can be sent by clients as several headers, each with a different value rather than sending the header as a comma-separated list. If the request did not include any headers of the specified name, this method returns an empty Enumeration object. The header name is case insensitive. You can use this method with any request header.
- **getHeaderNames():** The `getHeaderNames()` method returns an Enumeration of all the names of headers sent by a request. In combination with the `getHeader()` and `getHeaders()` methods, `getHeaderNames()` can be used to retrieve names and values of all the headers sent with a request. Some containers do not allow access of HTTP headers. In this case `null` is returned.
- **getIntHeader(java.lang.String name):** The `getIntHeader()` method returns the value of the specified request header as an `int`. If the request does not have a header of the specified name, this method returns `-1`. If the header cannot be converted to an integer, this method throws a `NumberFormatException`.
- **getDateHeader(java.lang.String name):** The `getDateHeader()` method returns the value of the specified request header as a `long` value that represents a `Date` object. The date is returned as the number of milliseconds since the epoch. The header name is case insensitive. If the request did not have a header of the specified name, this method returns `-1`. If the header cannot be converted to a date, the method throws an `IllegalArgumentException`.

HTTP request headers are very helpful for determining all sorts of information. In the later chapters HTTP request headers are used as the primary resource for mining data about a client. This includes figuring out what language a client would prefer, what type of Web browser is being used, and if the client can support compressed content for efficiency. For now it is helpful to understand that these headers exist, and to get a general idea about what type of information the headers contain. Listing 2-8 is a Servlet designed to do just that. Save the following code as `ShowHeaders.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application.

**Listing 2-8** ShowHeaders.java

```
package com.jspbook;

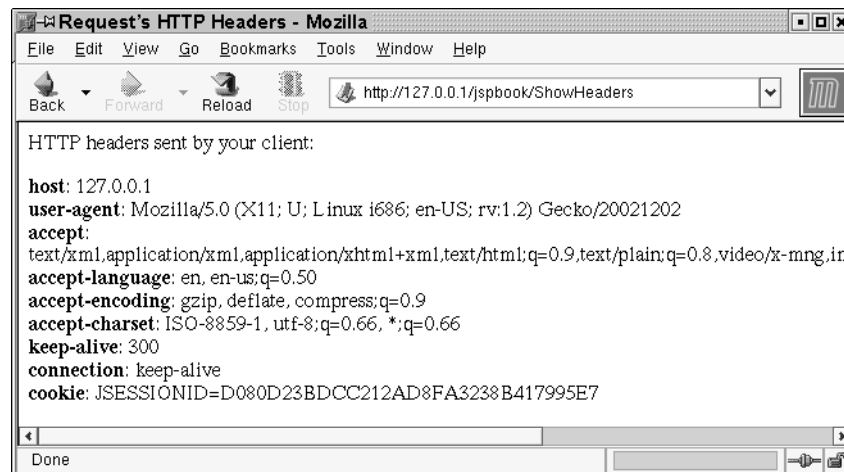
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShowHeaders extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Request's HTTP Headers</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>HTTP headers sent by your client:</p>");
        Enumeration enum = request.getHeaderNames();
        while (enum.hasMoreElements()) {
            String headerName = (String) enum.nextElement();
            String headerValue = request.getHeader(headerName);
            out.print("<b>"+headerName + "</b>: ");
            out.println(headerValue + "<br>");
        }
        out.println("</body>");
        out.println("</html>");
    }
}
```

Compile the Servlet and deploy it to the `/ShowHeaders` path of the `jspbook` Web Application. After reloading the Web application, browse to `http://127.0.0.1/jspbook/ShowHeaders` to see a listing of all the HTTP headers your browser sends (see Figure 2-11).

The preceding is a good example of the headers normally sent by a Web browser. They are fairly self-descriptive. You can probably imagine how these headers can be used to infer browser and internationalization information. Later



**Figure 2-11** Browser Rendering of the ShowHeaders Servlet

on we will do just that<sup>28</sup>, but for now we end our discussion of request headers with Table 2-3, which lists some of the most relevant and helpful ones.

### Form Data and Parameters

Perhaps the most commonly used methods of the `HttpServletRequest` object are the ones that involve getting request parameters: `getParameter()` and `getParameterNames()`. Anytime an HTML form is filled out and sent to a server, the fields are passed as parameters. This includes any information sent via input fields, selection lists, combo boxes, check boxes, and hidden fields, but excludes file uploads. Any information passed as a query string is also available on the server-side as a request parameter. The `HttpServletRequest` object includes the following methods for accessing request parameters.

- **`getParameter(java.lang.String parameterName)`:** The `getParameter()` method takes as a parameter a parameter name and returns a `String` object representing the corresponding value. A null is returned if there is no parameter of the given name.

28. In particular, the `Referer` header is perfect for tracking who sends traffic to your Web site; this is similar to the `LinkTracker` Servlet and very complementary. However, implementing referer tracking via a Servlet is cumbersome compared to using a filter. Therefore, a referer-tracking example is saved for the later chapter about filters.

**Table 2-3** HTTP Request Headers

Name	Value
Host	The <code>Host</code> request header field specifies the Internet host and port number of the resource being requested, as obtained from the original URL given by the user or referring resource. Mandatory for HTTP 1.1.
User-Agent	The <code>User-Agent</code> request header field contains information about the user agent (or browser) originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user-agent limitations.
Accept	The <code>Accept</code> request header field can be used to specify certain media types that are acceptable for the response. Accept headers can be used to indicate that the request is specifically limited to a small set of desired types.
Accept-Language	The <code>Accept-Language</code> request header field is similar to <code>Accept</code> , but restricts the set of natural languages that are preferred as a response to the request.
Accept-Charset	The <code>Accept-Charset</code> request header field can be used to indicate what character sets are acceptable for the response. This field allows clients capable of understanding more comprehensive or special-purpose character sets to signal that capability to a server that is capable of representing documents in those character sets. The ISO-8859-1 character set can be assumed to be acceptable to all user-agents. <i>Referer (sic)</i> The <code>Referer</code> request header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request URI was obtained.

- **getParameters(java.lang.String parameterName):** The `getParameters()` method is similar to the `getParameter()` method, but it should be used when there are multiple parameters with the same name. Often an HTML form check box or combo box sends multiple values for the same parameter name. The `getParameter()` method is a convenient method of getting all the parameter values for the same parameter name returned as an array of strings.

- **getParameterNames():** The `getParameterNames()` method returns a `java.util.Enumeration` of all the parameter names used in a request. In combination with the `getParameter()` and `getParameters()` method, it can be used to get a list of names and values of all the parameters included with a request.

Like the `ShowHeaders Servlet`, it is helpful to have a `Servlet` that reads and displays all the parameters sent with a request. You can use such a `Servlet` to get a little more familiar with parameters, and to debug HTML forms by seeing what information is being sent. Listing 2-9 is such a `Servlet`.

**Listing 2-9** `ShowParameters.java`

```
package com.jspbook;

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

public class ShowParameters extends HttpServlet {

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

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Request HTTP Parameters Sent</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>Parameters sent with request:</p>");
        Enumeration enum = request.getParameterNames();
        while (enum.hasMoreElements()) {
            String pName = (String) enum.nextElement();
            String[] pValues = request.getParameterValues(pName);
            out.print("<b>" + pName + "</b>: ");
            for (int i=0; i<pValues.length; i++) {
                out.print(pValues[i]);
            }
            out.print("<br>");
        }
    }
}
```

```
        out.println("</body>");
        out.println("</html>");
    }
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {
        doGet(request, response);
    }
}
```

Compile and deploy the ShowParameters Servlet in the jspbook Web Application with a mapping to /ShowParameters path. After reloading the jspbook Web Application, create a few simple HTML forms and use the Servlet to see what parameters are sent. If your HTML is out of practice, do not worry. Listing 2-10 provides a sample HTML form along with a link to a great online HTML reference, <http://www.jspinsider.com/reference/html.jsp>.

**Listing 2-10** exampleform.html

```
<html>
  <head>
    <title>Example HTML Form</title>
  </head>
  <body>
    <p>To debug a HTML form set its 'action' attribute
      to reference the ShowParameters Servlet.</p>
    <form action="http://127.0.0.1/jspbook/ShowParameters"
          method="post">
      Name: <input type="text" name="name"><br>
      Password: <input type="password" name="password"><br>
      Select Box:
      <select name="selectbox">
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
      </select><br>
      Importance:
      <input type="radio" name="importance" value="very">Very,
      <input type="radio"
        name="importance" value="normal">Normal,
      <input type="radio" name="importance" value="not">Not<br>
```



```
Comment: <br>
<textarea name="textarea"
  cols="40" rows="5"></textarea><br>
<input value="Submit" type="submit">
</form>
</body>
</html>
```

Either save the preceding HTML, or create any other HTML form and set the action attribute to `http://127.0.0.1/jspbook/ShowParameters`, and browse to the page. Save the preceding HTML as `exampleform.html` in the base directory of jspbook Web Application and browse to `http://127.0.0.1/jspbook/exampleform.html`. Figure 2-12 shows what the page looks like rendered by a Web browser.

Fill out the form and click on the button labeled Submit to send the information to the ShowParameters Servlet. Something resembling Figure 2-13 will appear.

On the server-side each piece of information sent by a form is referenced by the same name as defined in your HTML form and is linked to a value that a user entered. The ShowParameters Servlet shows this by using `getParameterNames()` for a list of all parameter names and subsequently calling `getParameter()` for the matching value or set of values for each name. The core of the Servlet is a simple loop.

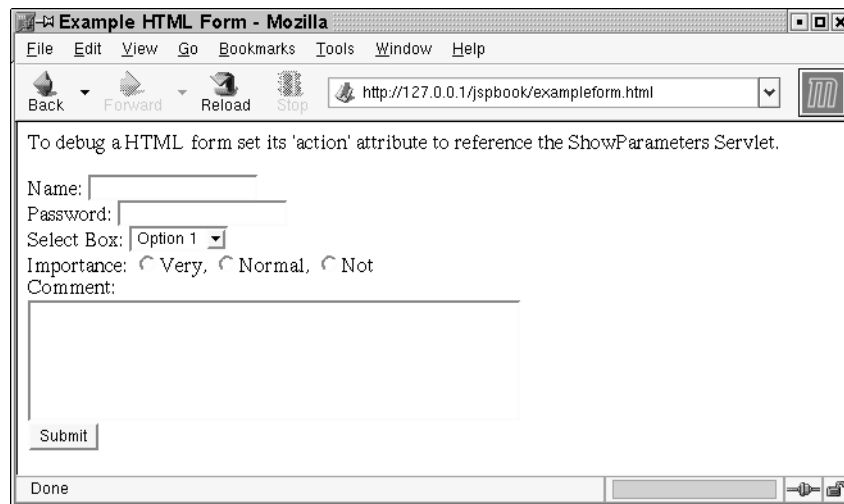
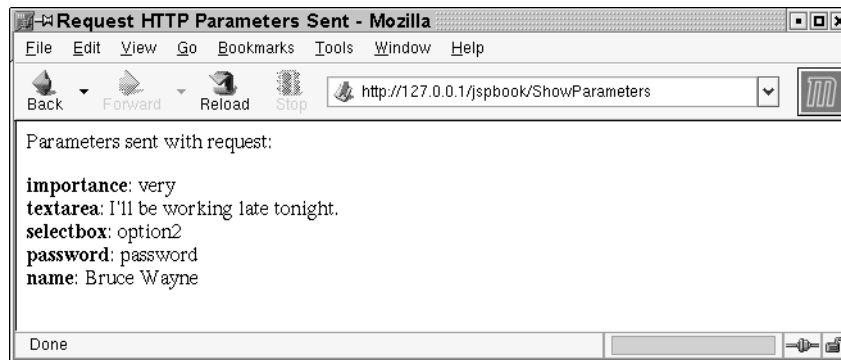


Figure 2-12 Browser Rendering of exampleform.html



**Figure 2-13** Browser Rendering of the ShowParameters Servlet

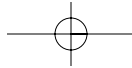
```
Enumeration enum = request.getParameterNames();
while (enum.hasMoreElements()) {
    String pName = (String) enum.nextElement();
    String[] pValues = request.getParameterValues(pName);
    out.print("<b>" + pName + "</b>: ");
    for (int i=0; i<pValues.length; i++) {
        out.print(pValues[i]);
    }
    out.print("<br>");
}
```

Using parameters, information can be solicited from HTML clients for use by a Servlet. While the ShowParameters Servlet only takes parameters and echoes them back to a client, normally those parameter values are used with other code to generate responses. Later on in the book this functionality will commonly be used with Servlets and JSP for further interacting with clients, including sending email and user authentication.

### **File Uploads**

File uploads<sup>29</sup> are simple for HTML developers but difficult for server-side developers. Sadly, this often results in discussion of Servlets and HTML forms that conveniently skip the topic of file uploads, but understanding HTML form file uploads is a needed skill for any good Servlet developer. Consider any situation

29. Be aware that it is possible to execute a Web Application directly from a WAR file, in which case the application may not have access to the file system, so the file upload code may fail.



where a client needs to upload something besides a simple string of text, perhaps when a picture needs to be uploaded. Using the `getParameter()` method will not work because it produces unpredictable results—usually mangling the file being sent or failing to find the content of the file at all.

The reason file uploads are usually considered difficult is because of how the Servlet API handles them. There are two primary MIME types for form information: `application/x-www-form-urlencoded` and `multipart/form-data`. The first MIME type, `application/x-www-form-urlencoded`, is the MIME type most everyone is familiar with and results in the Servlet API automatically parsing out name and value pairs. The information is then available by invoking `HttpServletRequest.getParameter()` or any of the other related methods as previously described. The second MIME type, `multipart/form-data`, is the one that is usually considered difficult. The reason why is because the Servlet API does nothing to help you with it<sup>30</sup>. Instead the information is left as is and you are responsible for parsing the request body via either `HttpServletRequest.getInputStream()` or `getReader()`.

The complete `multipart/form-data` MIME type and the format of the associated HTTP request are explained in RFC 1867, <http://www.ietf.org/rfc/rfc1867.txt>. You can use this RFC to determine how to appropriately handle information posted to a Servlet. The task is not very difficult, but, as will be explained later, this is usually not needed because other developers have created complementary APIs to handle file uploads.

To best understand what you are dealing with when `multipart/form-data` information is sent, it is valuable to actually look at the contents of such a request. This can be accomplished by making a file-uploading form and a Servlet that regurgitates the information obtained from the `ServletInputStream` provided. Listing 2-11 provides the code for such a Servlet. This Servlet accepts a `multipart/form-data` request and displays the contents of it as plain text.

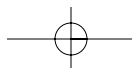
**Listing 2-11** ShowForm.java

```
package com.jspbook;

import java.util.*;
import java.io.*;
```

---

30. In practice, most implementations of the Servlet API try to be merciful on developers who attempt to invoke `getParameter()` functionality on forms that post information as `multipart/form-data`. However, this almost always results in parsing the name-value pairs and missing the uploaded file.



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

public class ShowForm extends HttpServlet {

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

        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();

        ServletInputStream sis = request.getInputStream();
        for (int i = sis.read(); i != -1; i = sis.read()) {
            out.print((char)i);
        }
    }

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

Save the preceding code as `ShowForm.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application. Deploy the Servlet to `/ShowForm`. From now on information posted by any form can be viewed by directing the request to `http://127.0.0.1/jspbook/ShowForm`. Try the Servlet out by creating an HTML form that uploads a file (Listing 2-12).

**Listing 2-12** multipartform.html

```
<html>
  <head>
    <title>Example HTML Form</title>
  </head>
  <body>
    <p>The ShowForm Servlet will display the content
    posted by an HTML form. Try it out by choosing a
    file (smaller size is preferred) to reference the
    ShowParameters Servlet.</p>
    <form action="http://127.0.0.1/jspbook/ShowForm"
          method="post" enctype="multipart/form-data">
      Name: <input type="text" name="name"><br>
```

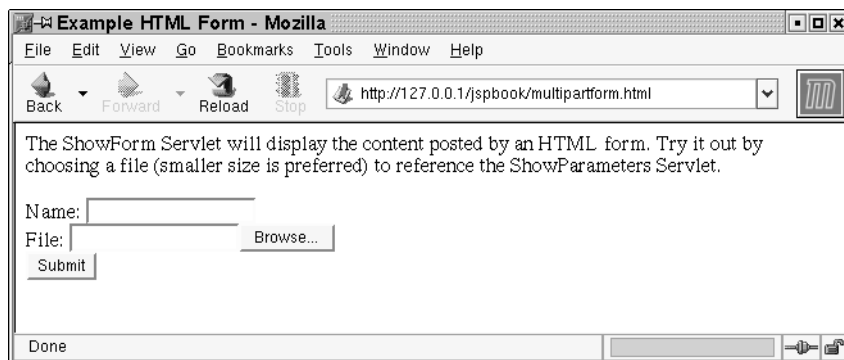
```
File: <input type="file" name="file"><br>
<input value="Submit" type="submit">
</form>
</body>
</html>
```

Save the preceding code as `multipartform.html` in the base directory of the `jspbook` Web Application and browse to it. Displayed is a small HTML form with two inputs, a name and a file to upload. Figure 2-14 provides a browser rendering of the page.

Fill in the form with any value for the Name field and any file for the File field. A smaller file is preferred because its contents are going to be displayed by the `ShowForm` Servlet and a large file will take up a lot of space. A good candidate for a file is `multipartform.html` itself. After completing the form, click on the Submit button. The `ShowForm` Servlet will then show the content that was posted. For example, using `multipartform.html` as the file would result in something similar to the following:

```
-----196751392613651805401540383426
Content-Disposition: form-data; name="name"

blah blah
-----196751392613651805401540383426
Content-Disposition: form-data; name="file";
filename="multipartform.html"
Content-Type: text/html
```



**Figure 2-14** Browser Rendering of `multipartform.html`

```

<html>
  <head>
    <title>Example HTML Form</title>
  </head>
  <body>
    <p>The ShowForm Servlet will display the content
    posted by an HTML form. Try it out by choosing a
    file (smaller size is preferred) to reference the
    ShowParameters Servlet.</p>
    <form action="http://127.0.0.1/jspbook/ShowForm"
      method="post" enctype="multipart/form-data">
      Name: <input type="text" name="name"><br>
      File: <input type="file" name="file"><br>
      <input value="Submit" type="submit">
    </form>
  </body>
</html>

```

```
-----196751392613651805401540383426--
```

This would be the type of information you have to parse through when handling a `multipart/form-data` request. If the file posted is not text, it will not be as pretty as the preceding, but there will always be a similar format. Each multipart has a unique token declaring its start. In the preceding the following was used:

```
-----196751392613651805401540383426
```

This declares the start of the multipart section and concluded at the ending token, which is identical to the start but with `'--'` appended. Between the starting and ending tokens are sections of data (possibly nested multiparts) with headers used to describe the content. For example, in the preceding code the first part described a form parameter:

```
Content-Disposition: form-data; name="name"
```

```
blah blah
```

The `Content-Disposition` header defines the information as being part of the form and identified by the name “name”. The value of “name” is the content following; by default its MIME type is `text/plain`. The second part describes the uploaded file:

```
Content-Disposition: form-data; name="file";
filename="multipartform.html"
```

```
Content-Type: text/html
```

```
<html>
  <head>
    <title>Example HTML Form</title>
  </head>
  <body>
  ...
```

This time the `Content-Disposition` header defines the name of the form field to be “file”, which matches what was in the HTML form, and describes the content-type as `text/html`, since it is not `text/plain`. Following the headers is the uploaded content, the code for `multipartform.html`.

You should be able to easily imagine how to go about creating a custom class that parses this information and saves the uploaded file to the correct location. In cases where the uploaded file is not using a special encoding, the task is as easy as parsing the file’s name from the provided headers and saving the content as is. Listing 2-13 provides the code for doing exactly that, and accommodates a file of any size. The Servlet acts as a file upload Servlet, which places files in the `/files` directory of the `jspbook` Web Application and lets users browse through, optionally downloading previously uploaded files.

**Listing 2-13** FileUpload.java

```
package com.jspbook;

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

public class FileUpload extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.print("File upload success. <a href=\"/jspbook/files/\"");
        out.print("\>Click here to browse through all uploaded ");
        out.println("files.</a><br>");
    }
}
```

```
ServletInputStream sis = request.getInputStream();
StringWriter sw = new StringWriter();
int i = sis.read();
for (; i != -1 && i != '\r'; i = sis.read()) {
    sw.write(i);
}
sis.read(); // ditch '\n'
String delimiter = sw.toString();

int count = 0;
while(true) {
    StringWriter h = new StringWriter();
    int[] temp = new int[4];
    temp[0] = (byte)sis.read();
    temp[1] = (byte)sis.read();
    temp[2] = (byte)sis.read();
    h.write(temp[0]);
    h.write(temp[1]);
    h.write(temp[2]);
    // read header
    for (temp[3]=sis.read(); temp[3] != -1; temp[3]=sis.read()) {
        if (temp[0] == '\r' &&
            temp[1] == '\n' &&
            temp[2] == '\r' &&
            temp[3] == '\n') {
            break;
        }
        h.write(temp[3]);
        temp[0] = temp[1];
        temp[1] = temp[2];
        temp[2] = temp[3];
    }
    String header = h.toString();

    int startName = header.indexOf("name=");
    int endName = header.indexOf("\"", startName+6);
    if (startName == -1 || endName == -1) {
        break;
    }
    String name = header.substring(startName+6, endName);
    if (name.equals("file")) {
        startName = header.indexOf("filename=");
        endName = header.indexOf("\"", startName+10);
        String filename =
```



```
        header.substring(startName+10,endName);
ServletContext sc =
    request.getSession().getServletContext();

File file = new File(sc.getRealPath("/files"));
file.mkdirs();
FileOutputStream fos =
    new FileOutputStream(
        sc.getRealPath("/files")+ "/" + filename);

// write whole file to disk
int length = 0;
delimiter = "\r\n"+delimiter;
byte[] body = new byte[delimiter.length()];
for (int j=0;j<body.length;j++) {
    body[j] = (byte)sis.read();
}
// check it wasn't a 0 length file
if (!delimiter.equals(new String(body))) {
    int e = body.length-1;
    i=sis.read();
    for (;i!=-1;i=sis.read()) {
        fos.write(body[0]);
        for (int l=0;l<body.length-1;l++) {
            body[l]=body[l+1];
        }
        body[e] = (byte)i;
        if (delimiter.equals(new String(body))) {
            break;
        }
        length++;
    }
}

fos.flush();
fos.close();
}
if (sis.read() == '-' && sis.read() == '-') {
    break;
}
}
out.println("</html>");
}
public void doGet(HttpServletRequest request,
```

```

        HttpServletResponse response)
    throws IOException, ServletException {
        doPost(request, response);
    }
}

```

Save the preceding code as `FileUpload.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application. The code parses through each part of the form data—that is, each parameter or file—and saves all files in the `/files` directory of the `jspbook` Web Application.

The code is purposely left as one large Servlet because it is initially easier to digest the information if it is all in one place; nothing stops you from reimplementing the preceding code in a more object-oriented fashion, perhaps modeling an enhanced version of the `getParameter()` method. Two important points should be noted about the code for the `FileUpload` Servlet. At all times information is read directly from the `ServletInputStream` object—that is, directly from the client and minimally buffered. This allows for very large files to be handled equally well as files of a small size. Additionally, this parses completely through the information in the HTTP post—that is, it determines the delimiter and keeps parsing until the end of the request, when the delimiter with `'--'` appended is found.

```

    ServletInputStream sis = request.getInputStream();
    StringWriter sw = new StringWriter();
    int i = sis.read();
    for (; i != -1 && i != '\r'; i = sis.read()) {
        sw.write(i);
    }
    sis.read(); // ditch '\n'
    String delimiter = sw.toString();

    int count = 0;
    while(true) {
        ...
        if (sis.read() == '-' && sis.read() == '-') {
            break;
        }
    }
}

```

The while loop loops indefinitely, reading through each part of the form data. Recall form information is posted with a delimiter such as:

```
-----196751392613651805401540383426\r\n
```

Used to separate each section, ended with the same delimiter with ‘--’ appended:

```
-----196751392613651805401540383426--
```

The code for FileUpload.java automatically reads a section using the delimiter, but the code conditionally continues based on what is found immediately after the delimiter. Should the character sequence ‘\r\n’ be found then the loop continues, with the characters discarded. However, should the character ‘—’ be found, the loop is terminated because the end of the form information has been found.

The body of the indefinite while loop is responsible for parsing out the header and the content of the form-data part. Header information is found by parsing, starting from the delimiter, until the appropriate character sequence, ‘\r\n\r\n’, is found.

```
StringWriter h = new StringWriter();
int[] temp = new int[4];
temp[0] = (byte)sis.read();
temp[1] = (byte)sis.read();
temp[2] = (byte)sis.read();
h.write(temp[0]);
h.write(temp[1]);
h.write(temp[2]);
// read header
for (temp[3]=sis.read();temp[3]!=-1;temp[3]=sis.read())
{
    if (temp[0] == '\r' &&
        temp[1] == '\n' &&
        temp[2] == '\r' &&
        temp[3] == '\n') {
        break;
    }
    h.write(temp[3]);
    temp[0] = temp[1];
    temp[1] = temp[2];
    temp[2] = temp[3];
}
String header = h.toString();
```

Recall that form-part header information is separated from content by a blank line, ‘\r\n’—meaning the end of a line followed by a blank line; ‘\r\n\r\n’, signifies the division between header and content information, which is why

'\r\n\r\n' is being searched for. The actual search is trivial; a temporary array, `temp`, that holds four characters is used to check the last four characters parsed. After the entire header is parsed, it is echoed in the response, and the name of the form-part is determined.

```
int startName = header.indexOf("name=\"");
int endName = header.indexOf("\"",startName+6);
if (startName == -1 || endName == -1) {
    break;
}
String name = header.substring(startName+6, endName);
```

The form-part's name is specified, if it was provided, using the following format, `name="name"`, where `name` is the name as specified in the HTML form using the `name` attribute. The preceding code does nothing more than use the string manipulation functions of the `String` class to determine the value of `name`.

After the name of the form-part is determined, the matching content is selectively handled: if the name is "file", the contents are saved as a file; any other name is treated as a form parameter and echoed back in the response. There is nothing special about the name "file"; it is an arbitrary name chosen so that `FileUpload.java` knows to save the content as a file. The code used to save the file is similar to the code used to parse header information, except this time the delimiter is the form-part delimiter, not '\r\n\r\n'.

```
if (name.equals("file")) {
    startName = header.indexOf("filename=\"");
    endName = header.indexOf("\"",startName+10);
    String filename =
        header.substring(startName+10,endName);

    ServletConfig config = getServletConfig();
    ServletContext sc = config.getServletContext();
    FileOutputStream fos =
        new FileOutputStream(sc.getRealPath("/") + filename);

    // write whole file to disk
    int length = delimiter.length();
    byte[] body = new byte[delimiter.length()];
    for (int j=0;j<body.length-1;j++) {
        body[j] = (byte)sis.read();
        fos.write(body[j]);
    }
    int e = body.length-1;
```

```
i = sis.read();
for (;i!=-1;i=sis.read()) {
    body[e] = (byte)i;
    if (delimiter.equals(new String(body))) {
        break;
    }
    fos.write(body[e]);
    for(int k=0;k<body.length-1;k++) {
        body[k] = body[k+1];
    }
    length++;
}

fos.flush();
fos.close();
out.println("<p><b>Saved File:</b> "+filename+"</p>");
out.println("<p><b>Length:</b> "+ length+"</p>");
}
```

The code first determines the name of the file being uploaded by searching for a special string, `filename="name"`, where `name` is the file's name, in the header. Next, a file is created in the `/files` directory of the Web Application<sup>31</sup> with the same name, and the content is saved.

In order to use the `FileUpload` Servlet, an HTML form, similar to `multipartform.html`, needs to be created. The form may include any number of input elements of any type and in any order, but one file input must be present and the input must be named "file". Listing 2-14 is a simple example. Save the following code as `fileupload.html` in the root directory of the `jspbook` Web Application.

**Listing 2-14** `fileupload.html`

```
<html>
  <head>
    <title>Example HTML Form</title>
  </head>
  <body>
    <p>Select a file to upload or
    <a href="/jspbook/files/">browse
```

---

31. The `ServletContext` and `ServletConfig` objects, both discussed later in this chapter, are required in order to save a file relative to the Web Application. Discussion of this is skipped right now in favor of the proper explanation provided later.

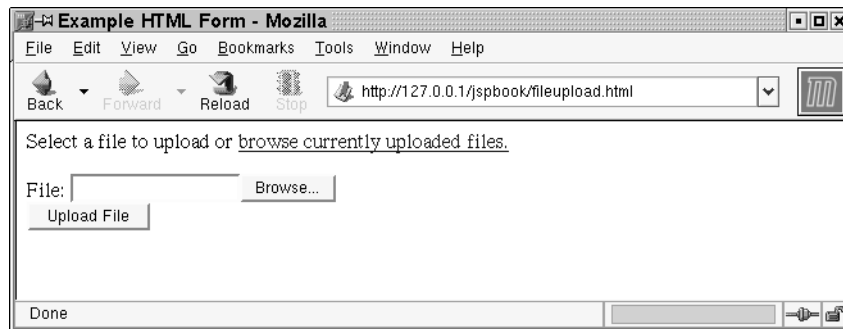
```
currently uploaded files.</a></p>
<form action="http://127.0.0.1/jspbook/FileUpload"
  method="post" enctype="multipart/form-data">
  File: <input type="file" name="file"><br>
  <input value="Upload File" type="submit">
</form>
</body>
</html>
```

The HTML form posts information to `/FileUpload`, the deployment of the `FileUpload` Servlet. Browse to `http://127.0.0.1/jspbook/fileupload.html` to try out the newly created HTML form. Figure 2-15 provides a browser rendering of the form.

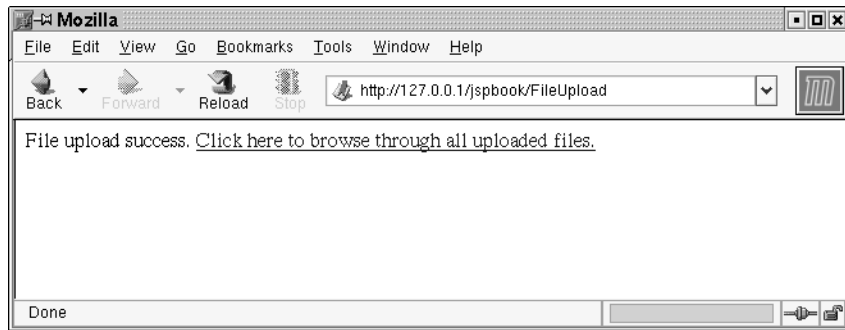
Fill in the form fields. The file field should be a file you wish to have on the server; try something such as a picture. Upon clicking on the Submit button, the form's information, including the file, is uploaded to the `FileUpload` Servlet. The `FileUpload` Servlet saves the file and displays a summary page, as shown in Figure 2-16.

Verify the file has been successfully uploaded by clicking on the link provided to browse the `/files` directory of the `jspbook` Web Application. Tomcat displays the default directory listing that includes the file just uploaded. To download the file or any other file in the directory, simply click on the file's listing.

In general, the `UploadFile` Servlet is demonstrating how a Servlet can parse multi-part form data and save uploaded files. The code can be adapted for other situations where files need to be uploaded, perhaps an online photo album or a more robust file sharing service. It should be noted that no restriction exists on what may



**Figure 2-15** Browser Rendering of `fileupload.html`



**Figure 2-16** Browser Rendering of the FileUpload Servlet's Response

be done with a file uploaded by an HTML form. In the FileUpload.java example, the file is saved in the /files directory of the Web Application, but the file could just as easily been saved elsewhere in the Web Application or not saved at all.

### **Using a File Upload API**

As a good developer, it is helpful to understand exactly how the Servlet API handles file uploads; however, in most every practical case you can do away with manually parsing and handling a file upload. File uploads are nothing new, and several implementations of file upload API exist. A good, free, open source file upload API is the Jakarta Commons File Upload API, <http://jakarta.apache.org/commons/fileupload/>. Download the latest release of the code (it is a very small JAR) and put the JAR file in the /WEB-INF/lib directory of the jspbook Web Application.

There are several reasons a file upload API can be helpful. A great reason is it can greatly simplify your code. Consider the FileUpload Servlet in the previous section. Using the Jakarta Commons File Upload API, the code can be reduced to Listing 2-15.

**Listing 2-15** FileUploadCommons.java

```
package com.jspbook;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.commons.fileupload.*;
import java.util.*;
```

```
public class FileUploadCommons extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.print("File upload success. <a href=\"/jspbook/files/\"");
        out.print("\>Click here to browse through all uploaded ");
        out.println("files.</a><br>");

        ServletContext sc = getServletContext();
        String path = sc.getRealPath("/files");
        org.apache.commons.fileupload.FileUpload fu = new
            org.apache.commons.fileupload.FileUpload();
        fu.setSizeMax(-1);
        fu.setRepositoryPath(path);
        try {
            List l = fu.parseRequest(request);
            Iterator i = l.iterator();
            while (i.hasNext()) {
                FileItem fi = (FileItem)i.next();
                fi.write(path+"/"+fi.getName());
            }
        }
        catch (Exception e) {
            throw new ServletException(e);
        }

        out.println("</html>");
    }
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        doPost(request, response);
    }
}
```

The code more closely follows good object-oriented programming by abstracting request parsing logic. Instead of implementing RFC 1867 by hand, as we did in `FileUpload.java`, the Jakarta Commons File Upload API handles all the work.



```
ServletContext sc = getServletContext();
String path = sc.getRealPath("/files");
FileUpload fu = new FileUpload();
fu.setSizeMax(-1);
fu.setRepositoryPath("/root/todelete");
try {
    List l = fu.parseRequest(request);
    Iterator i = l.iterator();
    while (i.hasNext()) {
        FileItem fi = (FileItem)i.next();
        fi.write(path+"/"+fi.getName());
    }
}
catch (Exception e) {
    throw new ServletException(e);
}
```

We will not discuss the file upload API in depth, but it should be easy to follow what is going on. The `FileUpload` object abstracts all of the code responsible for parsing `multipart/form-data` information. The only thing we need to care about is limiting the size of file uploads and specifying a temporary directory for the API to work with. The `parseRequest()` method takes a `HttpServletRequest` and returns an array of files parsed from the input—what more could you ask for?

In addition to simplifying code, there are a few other reasons that justify using an existing file upload API. There are several nuances of file uploads that need to be dealt with; multiple files can be uploaded, different encodings can be used, and excessively large files might be uploaded. In short, the less code you have to manage the better, and a good file upload API can easily take care of handling `multipart/form-data` information. Certainly consider using an existing file upload API when working with Servlets and file uploads. If anything, the Jakarta Commons File Upload API provides an excellent starting point for handling file uploads or creating a custom file upload API.

### ***Request Delegation and Request Scope***

Request delegation is a powerful feature of the Servlet API. A single client's request can pass through many Servlets and/or to any other resource in the Web Application. The entire process is done completely on the server-side and, unlike response redirection, does not require any action from a client or extra information sent between the client and server. Request delegation is available through the `javax.servlet.RequestDispatcher` object. An appropriate instance of a

`RequestDispatcher` object is available by calling either of the following methods of a `ServletRequest` object:

- **`getRequestDispatcher(java.lang.String path)`:** The `getRequestDispatcher()` method returns the `RequestDispatcher` object for a given path. The path value may lead to any resource in the Web Application and must start from the base directory, “/”.
- **`getNamedDispatcher(java.lang.String name)`:** The `getNamedDispatcher()` method returns the `RequestDispatcher` object for the named Servlet. Valid names are defined by the `servlet-name` elements of `web.xml`.

A `RequestDispatcher` object provides two methods for including different resources and for forwarding a request to a different resource.

- **`forward(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`:** The `forward()` method delegates a request and response to the resource of the `RequestDispatcher` object. A call to the `forward()` method may be used only if no content has been previously sent to a client. No further data can be sent to the client after the forward has completed.
- **`include(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`:** The `include()` method works similarly to `forward()` but has some restrictions. A Servlet can `include()` any number of resources that can each generate responses, but the resource is not allowed to set headers or commit a response.

Request delegation is often used to break up a large Servlet into smaller, more relevant parts. A simple case would include separating out a common HTML header that all pages on a site share. The `RequestDispatcher` object's `include()` method then provides a convenient method of including the header with the Servlet it was separated from and in any other Servlet needing the header. Any future changes to the header, and all the Servlets automatically reflect the change. For now, an example of simple Servlet server-side includes will be held in abeyance. JavaServer Pages<sup>32</sup> provide a much more elegant solution to this problem, and in practice Servlet request delegation is usually used for an entirely different purpose.

---

32. JSP is based directly off Servlets and is covered in full in Chapter 3.

In addition to simple server-side includes, request delegation is a key part of server-side Java implementations of popular design patterns. With respect to Servlet and JSP, design patterns are commonly agreed-upon methods for building Web Applications that are robust in functionality and easily maintainable. The topic of design is given a whole chapter to itself, Chapter 11, so no direct attempt will be given to demonstrate it now. Instead, discussion will focus on laying the foundation for Chapter 11 by explaining the new object scope that request delegation introduces.

With Java there are well-defined scopes for variables that you should already be familiar with. Local variables declared inside methods are by default only available inside the scope of that method. Instance variables, declared in a class but outside a method or constructor, are available to all methods in the Java class. There are many other scopes too, but the point is that these scopes are helpful to keep track of objects and help the JVM accurately garbage-collect memory. In Servlets, all of the previous Java variable scopes still exist, but there are some new scopes to be aware of. Request delegation introduces the *request scope*.

Request scope and the other scopes mentioned in this chapter are not something officially labeled by the Servlet specification<sup>33</sup>. The Servlet specification only defines a set of methods that allow objects to be bound to and retrieved from various containers (that are themselves objects) in the `javax.servlet` package. Since an object bound in this manner is referenced by the container it was bound to, the bound object is not destroyed until the reference is removed. Hence, bound objects are in the same “scope” as the container they are bound to. The `HttpServletRequest` object is such a container and includes such methods. These methods can be used to bind, access, and remove objects to and from the request scope that is shared by all Servlets to which a request is delegated. This concept is important to understand and can easily be shown with an example.

An easy way to think of request scope is as a method of passing any object between two or more Servlets and being assured the object goes out of scope (i.e., will be garbage-collected) after the last Servlet is done with it. More powerful examples of this are provided in later chapters, but to help clarify the point now, here are two Servlets that pass an object. Save the code in Listing 2-16 as `Servlet2Servlet.java` in the `/WEB-INF/classes/com/jspbook` directory of the jspbook Web Application.

---

33. Request scope and other scopes are officially recognized in the JSP specification

**Listing 2-16** Servlet2Servlet.java

```
package com.jspbook;

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

public class Servlet2Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");

        String param = request.getParameter("value");
        if(param != null && !param.equals("")) {
            request.setAttribute("value", param);
            RequestDispatcher rd =
request.getRequestDispatcher("/Servlet2Servlet2");
            rd.forward(request, response);
            return;
        }

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet #1</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>A form from Servlet #1</h1>");
        out.println("<form>");
        out.println("Enter a value to send to Servlet #2.");
        out.println("<input name=\"value\"><br>");
        out.print("<input type=\"submit\" ");
        out.println("value=\"Send to Servlet #2\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Deploy the preceding Servlet and map it to the `/Servlet2Servlet` URL extension. Next, save the code in Listing 2-17 as `Servlet2Servlet2.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application.

**Listing 2-17** `Servlet2Servlet2.java`

```
package com.jspbook;

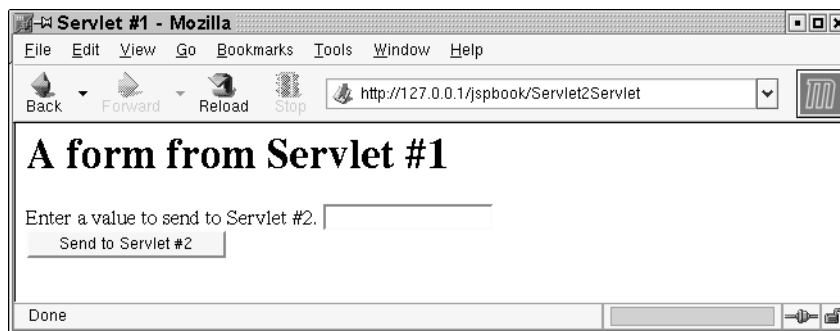
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Servlet2Servlet2 extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet #2</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet #2</h1>");
        String value = (String)request.getAttribute("value");
        if(value != null && !value.equals("")) {
            out.print("Servlet #1 passed a String object via ");
            out.print("request scope. The value of the String is: ");
            out.println("<b>"+value+"</b>.");
        }
        else {
            out.println("No value passed!");
        }
        out.println("</body>");
        out.println("</html>");
    }
}
```

Deploy the second Servlet and map it to the `/Servlet2Servlet2` URL extension. Reload the `jspbook` Web Application and the example is ready for use. Browse to `http://127.0.0.1/jspbook/Servlet2Servlet`. Figure 2-17 shows what the Servlet response looks like after being rendered by a Web browser. A



**Figure 2-17** Browser Rendering of Servlet2Servlet

simple HTML form is displayed asking for a value to pass to the second Servlet. Type in a value and click on the button labeled Send to Servlet #2.

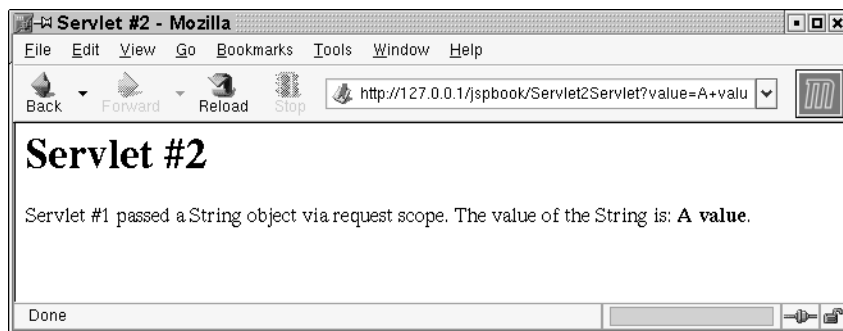
Information sent from the HTML form is sent straight back to the same Servlet that made the form, Servlet2Servlet. The Servlet verifies a value was sent; creates a new String object; places the String in request scope; and forwards the request to the second Servlet, Servlet2Servlet2. Figure 2-18 shows what a browser rendering of the second Servlet's output looks like. The content on the page shows the request was delegated to the second Servlet, but you can verify the request was delegated by the first Servlet by looking at the URL. This technique is extremely useful and discussed further in the design pattern chapter.

## ServletContext

The `javax.servlet.ServletContext` interface represents a Servlet's view of the Web Application it belongs to. Through the `ServletContext` interface, a Servlet can access raw input streams to Web Application resources, virtual directory translation, a common mechanism for logging information, and an *application scope* for binding objects. Individual container vendors provide specific implementations of `ServletContext` objects, but they all provide the same functionality defined by the `ServletContext` interface.

## Initial Web Application Parameters

Previously in this chapter initial parameters for use with individual Servlets were demonstrated. The same functionality can be used on an application-wide basis to provide initial configuration that all Servlets have access to. Each Servlet has a



**Figure 2-18** Browser Rendering of Request Delegated to Servlet2Servlet2

`ServletConfig` object accessible by the `getServletConfig()` method of the Servlet. A `ServletConfig` object includes methods for getting initial parameters for the particular Servlet, but it also includes the `getServletContext()` method for accessing the appropriate `ServletContext` instance. A `ServletContext` object implements similar `getInitParam()` and `getInitParamNames()` methods demonstrated for `ServletConfig`. The difference is that these methods do not access initial parameters for a particular Servlet, but rather parameters specified for the entire Web Application.

Specifying application-wide initial parameters is done in a similar method as with individual Servlets, but requires replacement of the `init-param` element with the `context-param` element of `Web.xml`, and requires the tag be placed outside any specific `servlet` tag. Occurrences of `context-param` tags should appear before any Servlet tags. A helpful use of application context parameters is specifying contact information for an application's administration. Using the current `jspbook web.xml`, an entry for this would be placed as follows.

```
...
<web-app>
  <context-param>
    <param-name>admin email</param-name>
    <param-value>admin@jspbook.com</param-value>
  </context-param>
  <servlet>
    <servlet-name>helloworld</servlet-name>
    <servlet-class>com.jspbook.HelloWorld</servlet-class>
  </servlet>
...

```

We have yet to see how to properly handle errors and exceptions thrown from Servlets, but this initial parameter is ideal for error handling Servlets. For now, create a Servlet that assumes it will be responsible for handling errors that might arise with the Web Application. In Chapter 4 we will show how to enhance this Servlet to properly handle thrown exceptions, but for now pretend a mock error was thrown. Save the code in Listing 2-18 as `MockError.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application.

**Listing 2-18** `MockError.java`

```
package com.jspbook;

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

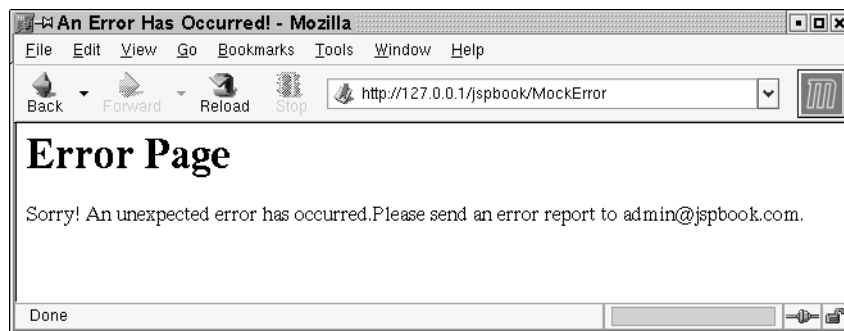
public class MockError extends HttpServlet {

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

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>An Error Has Occurred!</title>");
        out.println("</head>");
        out.println("<body>");
        ServletContext sc =
            getServletConfig().getServletContext();
        String adminEmail = sc.getInitParameter("admin email");
        out.println("<h1>Error Page</h1>");
        out.print("Sorry! An unexpected error has occurred.");
        out.print("Please send an error report to "+adminEmail+".");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Reload the `jspbook` Web Application and browse to `http://127.0.0.1/jspbook/MockError` to see the page with the context parameter information





**Figure 2-19** Browser Rendering of the MockError Servlet

included. Figure 2-19 shows what the MockError Servlet looks like when rendered by a Web browser.

Notice the correct context parameter value was inserted in the error message. This same parameter might also be used in other various Servlets as part of a header or footer information. The point to understand is that this parameter can be used throughout the entire Web Application while still being easily changed when needed.

### **Application Scope**

Complementing the request scope, a `ServletContext` instance allows for server-side objects to be placed in an application-wide scope<sup>34</sup>. This type of scope is ideal for placing resources that need to be used by many different parts of a Web Application during any given time. The functionality is identical to that as described for the `HttpRequest` object and relies on binding objects to a `ServletContext` instance. For brevity this functionality will not be iterated over again, but will be left for demonstration in later examples of the book.

It is important to note that an application scope should be used sparingly. Objects bound to a `ServletContext` object will not be garbage collected until the `ServletContext` is removed from use, usually when the Web Application is turned off or restarted. Placing large amounts of unused objects in application scope does tax a server's resources and is not good practice. Another issue (that will be gone into in more detail later) is that the `ServletContext` is not truly

34. There is also a session scope that will be covered in detail in Chapter 9.

application-wide. If the Web Application is running on multiple servers (say, a Web farm), then there will be multiple `ServletContext` objects; any updates to one `ServletContext` on one server in the farmer will not be replicated to the other `ServletContext` instances.

### **Virtual Directory Translation**

All the resources of a Web Application are abstracted to a virtual directory. This directory starts with a root, “/”, and continues on with a virtual path to sub-directories and resources. A client on the World Wide Web can access resources of a Web Application by appending a specific path onto the end of the HTTP address for the server the Web Application runs on. The address for reaching the jspbook Web Application on your local computer is `http://127.0.0.1`. Combining this address with any virtual path to a Web Application resource provides a valid URL for accessing the resource via HTTP.

A Web Application’s virtual directory is helpful because it allows fictitious paths to link to real resources located in the Web Application. The only downside to the functionality is that Web Application developers cannot directly use virtual paths to obtain the location of a physical resource. To solve this problem, the `ServletContext` object provides the following method:

```
getRealPath(java.lang.String path)35
```

The `getRealPath()` method returns a `String` containing the real path for a given virtual path. The real path represents the location of the resource on the computer running the Web Application.

To compliment the `getRealPath()` method, the `ServletContext` object also defines methods for obtaining a listing of resources in a Web Application or for an `InputStream` or URL connection to a particular resource:

- **getResourcePaths(java.lang.String path):** The `getResourcePaths()` method returns a `java.util.Set` of all the resources in the directory specified by the path. The path must start from the root of the Web Application, “/”.
- **getResourceAsStream(java.lang.String path):** The `getResourceAsStream()` method returns an instance of an

---

35. Again, be aware that a Servlet container is free to load Web Applications from places other than the file system (for example, directly from WAR files or from a database); in that case this method may return null.

`InputStream` to the physical resource of a Web Application. This method should be used when a resource needs to be read verbatim rather than processed by a Web Application.

- **`getResource(java.lang.String path)`:** The `getResource()` method returns a URL to the resource that is mapped to a specified path. This method should be used when a resource needs to be read as it would be displayed to a client.

It is important to remember that a Web Application is designed to be portable. Hard coding file locations in Servlet code is not good practice because it usually causes the Servlet not to work when deployed on a different server or if the Web Application is run directly from a compressed WAR file. The correct method for reading a resource from a Web Application is by using either the `getResource()` or `getResourceAsStream()` methods. These two methods ensure the Servlet will always obtain access to the desired resource even if the Web Application is deployed on multiple servers or as a compressed WAR.

The most common and practical use for virtual directory translation is for accessing important flat files packaged with a Web Application. This primarily includes configuration files but is also used for miscellaneous purposes such as simple flat file databases. An ideal example would be one involving a complex Servlet using a custom configuration file; however, a complex Servlet like this has yet to appear in this book. For a demonstration, a simple Servlet will be created that reads raw files and resources from a Web Application (Listing 2-19). While not necessary for most real-world uses, this Servlet is ideal for learning as it effectively shows the source code of an entire Web Application.

**Listing 2-19** ShowSource.java

```
package com.jspbook;

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

public class ShowSource extends HttpServlet {

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

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

// Get the ServletContext
ServletConfig config = getServletConfig();
ServletContext sc = config.getServletContext();

// Check to see if a resource was requested
String resource = request.getParameter("resource");
if (resource != null && !resource.equals("")) {

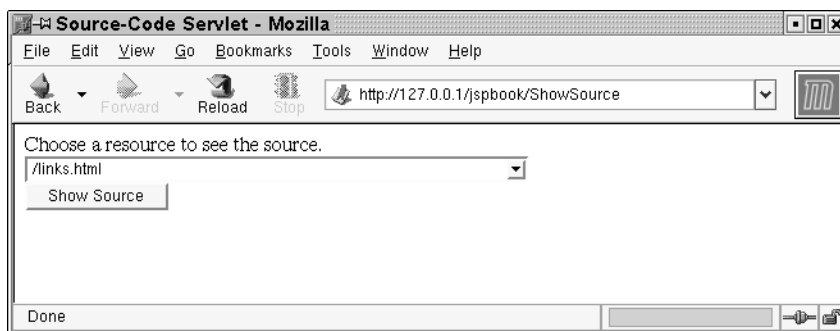
    // Use getResourceAsStream() to properly get the file.
    InputStream is = sc.getResourceAsStream(resource);
    if (is != null) {
        response.setContentType("text/plain");
        StringWriter sw = new StringWriter();
        for (int c = is.read(); c != -1; c = is.read()) {
            sw.write(c);
        }
        out.print(sw.toString());
    }
}
// Show the HTML form.
else {
    response.setContentType("text/html");
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Source-Code Servlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<form>");
    out.println("Choose a resource to see the source.<br>");
    out.println("<select name=\"resource\">");
    // List all the resources in this Web Application
    listFiles(sc, out, "/");
    out.println("</select><br>");
    out.print("<input type=\"submit\" ");
    out.println("value=\"Show Source\">");
    out.println("</body>");
    out.println("</html>");
}
}

// Recursively list all resources in Web App
void listFiles(ServletContext sc, PrintWriter out,
```

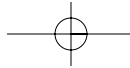
```
String base){
Set set = sc.getResourcePaths(base);
Iterator i = set.iterator();
while (i.hasNext()) {
String s = (String)i.next();
if (s.endsWith("/")) {
listFiles(sc, out, s);
}
else {
out.print("<option value=\"" + s);
out.println("\">" + s + "</option>");
}
}
}
```

Save Listing 2-15 as `ShowSource.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application. Compile the code and deploy the Servlet to the `/ShowSource` path extension of the `jspbook` Web Application, and after reloading the Web Application, browse to `http://127.0.0.1/jspbook/ShowSource`. Figure 2-20 shows what a browser rendering of the Servlet's output looks like.

The Servlet uses the `getResourcePaths()` method to obtain a listing of all the files in the Web Application. After selecting a file, the Servlet uses the `getResourceAsStream()` method to obtain an `InputStream` object for reading and displaying the source code of the resource.



**Figure 2-20** Browser Rendering of the ShowSource Servlet



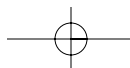
### Application-Wide Logging

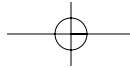
A nice but not commonly used feature of Servlets is Web Application-wide logging. A `ServletContext` object provides a common place all Servlets in a Web Application can use to log arbitrary information and stack traces for thrown exceptions. The advantage of this functionality is that it consolidates the often odd mix of custom code that gets used for logging errors and debugging information. The following two methods are available for logging information via a Web Application's `ServletContext`:

- **`log(java.lang.String msg)`:** The `log()` method writes the specified string to a Servlet log file or log repository. The Servlet API only specifies a `ServletContext` object implement these methods. No specific direction is given as to where the logging information is to be saved and or displayed. Logged information is sent to `System.out` or to a log file for the container.
- **`log(java.lang.String message, java.lang.Throwable throwable)`:** This method writes both the given message and the stack trace for the `Throwable` exception passed in as parameters. The message is intended to be a short explanation of the exception.

With J2SDK 1.4 the Servlet logging feature is not as helpful as it has previously been. The main advantage of the two `log()` methods is that they provided a common place to send information regarding the Web Application. Most often, as is with Tomcat, a container also allowed for a Servlet developer to write a custom class to handle information logged by a Web Application. This functionality makes it easy to customize how and where Servlet logging information goes. The downside to using the `ServletContext` `log()` methods is that only code that has access to a `ServletContext` object can easily log information. Non-Servlet classes require a creative hack to log information in the same manner. A better and more commonly used solution for Web Application logging is to create a custom set of API that any class can access and use. The idea is nothing new and can be found in popularized projects such as Log4j, <http://jakarta.apache.org/log4j> or with the J2SDK 1.4 standard logging API, the `java.util.logging` package. Both of these solutions should be preferred versus the Servlet API logging mechanism when robust logging is required.

In lieu of demonstrating the two `ServletContext` `log()` methods, a brief explanation and example of the `java.util.logging` package is given in Chapter 4. A flexible and consolidated mechanism for logging information is needed in any





serious project. The Servlet API logging mechanism does work for simple cases, but this book encourages the use of a more robust logging API.

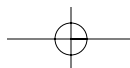
### ***Distributed Environments***

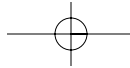
In most cases a `ServletContext` object can be considered as a reference to the entire Web Application. This holds true for single machine servers running one JVM for the Web Application. However, J2EE is not designed to be restricted for use on a single server. A full J2EE server, not just a container supporting Servlets and JSP, allows for multiple servers to manage and share the burden of a large-scale Web Application. These types of applications are largely outside the scope of this book, but it should be noted that application scope and initial parameters will not be the same in a distributed environment. Different servers and JVM instances usually do not share direct access to each other's information. If developing for a group of servers, do not assume that the `ServletContext` object will be the same for every request. This topic is discussed further in later chapters of the book.

### ***Temporary Directory***

One subtle but incredibly helpful feature of the Servlet specification is the requirement of a temporary work directory that Servlets and other classes in a Web Application can use to store information. The exact location of the temporary directory is not specified, but all containers are responsible for creating a temporary directory and setting a `java.io.File` object, which describes that directory as the `javax.servlet.context.tempdir` `ServletContext` attribute. Information stored in this directory is only required to persist while the Web Application is running, and information stored in the temporary directory will always be hidden from other Web Applications running on the same server.

The container-defined temporary directory is helpful for one really important reason: Web Applications can be run directly from a WAR; there is no guarantee that you can rely on using `ServletContext` `getRealPath()` to always work. To solve the problem, all Web Applications have access to at least one convenient place where temporary information can be stored, and that place is provided using the `javax.servlet.context.tempdir` attribute. There are a few good use cases where the `javax.servlet.context.tempdir` attribute is needed. In any situation where a Web Application is caching content locally (not in memory), the `javax.servlet.context.tempdir` attribute temporary directory is the ideal place to store the cache. Additionally, the temporary directory provides a place to temporarily store file uploads or any other information a Web Application is working with.





In practice, it is usually safe to assume your Web Application will be deployed outside of a WAR, especially when you have control over the deployment; however, in cases where a Web Application is intended for general use, the provided temporary directory should always be used to ensure maximum portability of your code. Later on in the book some use cases that deal with the temporary directory are provided; if you took a look at either of the previously mentioned file-upload API, you would have noticed they both take advantage of the temporary directory.

## Servlet Event Listeners

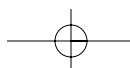
The final topic to discuss in this chapter is Servlet event listeners. In many situations it is desirable to be notified of container-managed life cycle events. An easy example to think of is knowing when the container initializes a Web Application and when a Web Application is removed from use. Knowing this information is helpful because you may have code that relies on it, perhaps a database that needs to be loaded at startup and saved at shutdown. Another good example is keeping track of the number of concurrent clients using a Web Application. This functionality can be done with what you currently know of Servlets; however, it can much more easily be done using a listener that waits for a client to start a new session. The greater point being presented here is that a container can be used to notify a Web Application of important events, and Servlet event listeners are the mechanism.

All of the Servlet event listeners are defined by interfaces. There are event listener interfaces for all of the important events related to a Web Application and Servlets. In order to be notified of events, a custom class, which implements the correct listener interface, needs to be coded and the listener class needs to be deployed via `web.xml`. All of the Servlet event listeners will be mentioned now; however, a few of the event listeners will not make complete sense until the later chapters of the book. In general, all of the event listeners work the same way so this fact is fine as long as at least one good example is provided here.

The interfaces for event listeners correspond to request life cycle events, request attribute binding, Web Application life cycle events, Web Application attribute binding, session<sup>36</sup> life cycle events, session attribute binding, and session serialization, and appear, respectively, as follows:

---

36. As described by this chapter, HTTP is a stateless protocol. However, it is often necessary to maintain state for the duration of all of a particular client's requests. A mechanism exists for this, and the mechanism is commonly called session management. See Chapter 9 for a detailed discussion on the topic.





- `javax.servlet.ServletRequestListener`
- `javax.servlet.ServletRequestAttributeListener`
- `javax.servlet.ServletContextListener`
- `javax.servlet.ServletContextAttributeListener`
- `javax.servlet.http.HttpSessionListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.servlet.http.HttpSessionAttributeListener`

Use of each listener is intuitive given an implementation of one and an example deployment descriptor. All of the listener interfaces define events for either the creation and destruction of an object or for notification of the binding or unbinding of an object to a particular scope.

As an example, let us create a listener that tracks the number of concurrent users. We could create a simple hit counter, by tracking how many requests are created; however, the previous hit counter examples in this chapter do a fine job providing the same functionality. Tracking the number of concurrent users is something that we have yet to see, and allows the introduction of *session scope*. For now, think of sessions as being created only once per unique client—regardless if the same person visits the site more than once. This is different from requests, which are created every time a client requests a resource. The listener interface we are going to implement is `HttpSessionListener`. It provides notification of two events: session creation and session destruction. In order to keep track of concurrent users, we will keep a static count variable that will increase incrementally when a session is created and decrease when a session is destroyed.

The physical methods required by the `HttpSessionListener` interface are as follows:

- **`void sessionCreated(HttpSessionEvent evt)`:** The method invoked when a session is created by the container. This method will almost always be invoked only once per a unique client.
- **`void sessionDestroyed(HttpSessionEvent evt)`:** The method invoked when a session is destroyed by the container. This method will be invoked when a unique client's session times out—that is, after they fail to revisit the Web site for a given period of time, usually 15 minutes.

Listing 2-20 provides our listener class's code, implementing the preceding two methods. Save the code as `ConcurrentUserTracker.java` in the `/WEB-INF/classes/com/jspbook` directory of the `jspbook` Web Application.

**Listing 2-20** ConcurrentUserTracker.java

```
package com.jspbook;

import javax.servlet.*;
import javax.servlet.http.*;

public class ConcurrentUserTracker implements HttpSessionListener {
    static int users = 0;

    public void sessionCreated(HttpSessionEvent e) {
        users++;
    }
    public void sessionDestroyed(HttpSessionEvent e) {
        users--;
    }
    public static int getConcurrentUsers() {
        return users;
    }
}
```

The listener's methods are intuitive and the logic is simple. The class is trivial to create because the container is managing all the hard parts: handling HTTP requests, trying to keep a session, and keeping a timer in order to successfully time-out unused sessions.

Deploy the listener by adding the entry in Listing 2-21 to `web.xml`. Add the entry after the starting `webapp` element but before any Servlet deployments.

**Listing 2-21** Deployment of the Concurrent User Listener

```
<listener>
  <listener-class>
    com.jspbook.ConcurrentUserTracker
  </listener-class>
</listener>
```

Notice that the deployment does not specify what type of listener interface is being used. This type of information is not needed because the container can figure it out; therefore, deployment of all the different listeners is similar to the above code. Create a `listener` element with a child `listener-class` element that has the name of the listener class.

One more addition is required before the concurrent user example can be tested. The concurrent user tracker currently doesn't output information about concurrent users! Let us create a Servlet that uses the static `getConcurrentUsers()` method to display the number of concurrent users. Save the code in Listing 2-22 as `DisplayUsers.java` in the `/WEB-INF/com/jspbook` directory of the jspbook Web Application.

**Listing 2-22** `DisplayUsers.java`

```
package com.jspbook;

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

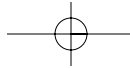
public class DisplayUsers extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {

        request.getSession();

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.print("Users:");
        out.print(ConcurrentUserTracker.getConcurrentUsers());
        out.println("</html>");
    }
}
```

Deploy the Servlet to the `/DisplayUsers` URL mapping. Compile both `ConcurrentUserTracker.java` and `DisplayUser.java` and reload the jspbook Web Application for the changes to take effect. Test the new functionality out by browsing to `http://127.0.0.1/jspbook/DisplayUsers`. An HTML page appears describing how many users are currently using the Web Application. A browser rendering would be provided; however, the page literally displays nothing more than the text "Users:" followed by the number of current users.

When browsing to the Servlet, it should say one user is currently using the Web Application. If you refresh the page, it will still say only one user is using the Web Application. Try opening a second Web browser and you should see



that the page states two people are using the Web Application<sup>37</sup>. Upon closing your Web browser, the counter will not go down, but after not visiting the Web Application for 15 minutes, it will. You can test this out by opening two browsers (to create two concurrent users) and only using one of the browsers for the next 15 minutes. Eventually, the unused browser's session will time-out and the Web Application will report only one concurrent user. In real-world use, odds are that several independent people will be using a Web Application at any given time. In this situation you don't need to do anything but visit the `DisplayUsers` Servlet to see the current amount of concurrent users.

The concurrent user tracker is a handy piece of code; however, the greater point to realize is how Servlet event listeners can be coded and deployed for use. Event listeners in general are intuitive to use; the hard part is figuring out when and how they are best used, which is hard to fully explain in this chapter. In later chapters of the book, event listeners will be used to solve various problems and complement code. Keep an eye out for them.

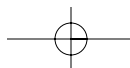
## Summary

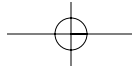
In this chapter the Servlet API was introduced, and focus was placed specifically on Servlets. Servlets are the fundamental building block of server-side Java. A Servlet is highly scalable and easily outperforms traditional CGI by means of a simple three-phase life cycle: initialization, service, and destruction. Commonly, the term Servlets actually refers to HTTP Servlets used on the World Wide Web. The `HttpServlet` class is designed especially for this user and greatly simplifies server-side Java support for HTTP.

The basics of the Servlet API consist of objects that represent a client's request, `HttpServletRequest`, the server's response, `HttpServletResponse`, a session for connecting separate requests, `HttpSession`, and an entire Web Application, `ServletContext`. Each of these objects provides a complete set of methods for accessing and manipulating related information. These objects also introduce two new scopes for Servlet developers to use: request and application. Binding an object to these various scopes allows a Servlet developer to share an object between multiple Servlets and requests for extended periods of time. What

---

37. This won't always appear to work on some browsers, namely Internet Explorer, due to browser tricks designed to be user-friendly. If you can open two complete different browsers, such as Mozilla and Internet Explorer, the Web Application will always report two concurrent users. If you are using Internet Explorer, make sure you open a new copy of the browser and not simply a new browser window.





was only briefly mentioned is that a third, commonly used scope, `session`, is also available. Session scope introduces several issues which merit a complete chapter's worth of information. Chapter 9 fully explains session scope and the issues relating to it.

Overall, this chapter is a reference to creating and using Servlets. Chapter 3 continues discussion of Servlets by introducing a complementary technology: JavaServer Pages.

