SESSION TRACKING

Topics in This Chapter

- Implementing session tracking from scratch
- Using basic session tracking
- Understanding the session-tracking API
- Differentiating between server and browser sessions
- Encoding URLs
- Storing immutable objects vs. storing mutable objects
- Tracking user access counts
- Accumulating user purchases
- Implementing a shopping cart
- Building an online store

Chapter 9

This chapter introduces the servlet session-tracking API, which keeps track of user-specific data as visitors move around your site.

9.1 The Need for Session Tracking

HTTP is a "stateless" protocol: each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server does not automatically maintain contextual information about the client. Even with servers that support persistent (keep-alive) HTTP connections and keep sockets open for multiple client requests that occur in rapid succession, there is no built-in support for maintaining contextual information. This lack of context causes a number of difficulties. For example, when clients at an online store add an item to their shopping carts, how does the server know what's already in the carts? Similarly, when clients decide to proceed to checkout, how can the server determine which previously created shopping carts are theirs? These questions seem very simple, yet, because of the inadequacies of HTTP, answering them is surprisingly complicated.

There are three typical solutions to this problem: cookies, URL rewriting, and hidden form fields. The following subsections quickly summarize what would be required if you had to implement session tracking yourself (without using the built-in session-tracking API) for each of the three ways.

Cookies

You can use cookies to store an ID for a shopping session; with each subsequent connection, you can look up the current session ID and then use that ID to extract information about that session from a lookup table on the server machine. So, there would really be two tables: one that associates session IDs with user tables, and the user tables themselves that store user-specific data. For example, on the initial request a servlet could do something like the following:

```
String sessionID = makeUniqueString();
HashMap sessionInfo = new HashMap();
HashMap globalTable = findTableStoringSessions();
globalTable.put(sessionID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessionID);
sessionCookie.setPath("/");
response.addCookie(sessionCookie);
```

Then, in later requests the server could use the `globalTable` hash table to associate a session ID from the `JSESSIONID` cookie with the `sessionInfo` hash table of user-specific data.

Using cookies in this manner is an excellent solution and is the most widely used approach for session handling. Still, it is nice that servlets have a higher-level API that handles all this plus the following tedious tasks:

- Extracting the cookie that stores the session identifier from the other cookies (there may be many cookies, after all).
- Determining when idle sessions have expired, and reclaiming them.
- Associating the hash tables with each request.
- Generating the unique session identifiers.

URL Rewriting

With this approach, the client appends some extra data on the end of each URL. That data identifies the session, and the server associates that identifier with user-specific data it has stored. For example, with `http://host/path/file.html;jsessionid=a1234`, the session identifier is attached as `jsessionid=a1234`, so `a1234` is the ID that uniquely identifies the table of data associated with that user.

URL rewriting is a moderately good solution for session tracking and even has the advantage that it works when browsers don't support cookies or when the user has disabled them. However, if you implement session tracking yourself, URL rewriting has the same drawback as do cookies, namely, that the server-side program has a lot of straightforward but tedious processing to do. Even with a high-level API that handles most of the details for you, you have to be very careful that every URL that references your site and is returned to the user (even by indirect means like `Location` fields in server redirects) has the extra information appended. This restriction means that you cannot have any static HTML pages on your site (at least not any that have links back to dynamic pages at the site). So, every page has to be dynamically generated with servlets or JSP. Even when all the pages are dynamically generated, if the user leaves the session and comes back via a bookmark or link, the session information can be lost because the stored link contains the wrong identifying information.

Hidden Form Fields

As discussed in Chapter 19 (Creating and Processing HTML Forms), HTML forms can have an entry that looks like the following:

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="a1234">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the `GET` or `POST` data. This hidden field can be used to store information about the session but has the major disadvantage that it only works if every page is dynamically generated by a form submission. Clicking on a regular (`<A HREF...>`) hypertext link does not result in a form submission, so hidden form fields cannot support general session tracking, only tracking within a specific series of operations such as checking out at a store.

Session Tracking in Servlets

Servlets provide an outstanding session-tracking solution: the `HttpSession` API. This high-level interface is built on top of cookies or URL rewriting. All servers are required to support session tracking with cookies, and most have a setting by which you can globally switch to URL rewriting.

Either way, the servlet author doesn't need to bother with many of the implementation details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store arbitrary objects that are associated with each session.

9.2 Session Tracking Basics

Using sessions in servlets is straightforward and involves four basic steps. Here is a summary; details follow.

1. **Accessing the session object associated with the current request.** Call `request.getSession` to get an `HttpSession` object, which is a simple hash table for storing user-specific data.
2. **Looking up information associated with a session.** Call `getAttribute` on the `HttpSession` object, cast the return value to the appropriate type, and check whether the result is `null`.
3. **Storing information in a session.** Use `setAttribute` with a key and a value.
4. **Discarding session data.** Call `removeAttribute` to discard a specific value. Call `invalidate` to discard an entire session. Call `logout` to log the client out of the Web server and invalidate all sessions associated with that user.

Accessing the Session Object Associated with the Current Request

Session objects are of type `HttpSession`, but they are basically just hash tables that can store arbitrary user objects (each associated with a key). You look up the `HttpSession` object by calling the `getSession` method of `HttpServletRequest`, as below.

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

Behind the scenes, the system extracts a user ID from a cookie or attached URL data, then uses that ID as a key into a table of previously created `HttpSession` objects. But this is all done transparently to the programmer: you just call `getSession`. If no session ID is found in an incoming cookie or attached URL information, the system creates a new, empty session. And, if cookies are being used (the default situation), the system also creates an outgoing cookie named `JSESSIONID` with a unique value representing the session ID. So, although you call `getSession` on the *request*, the call can affect the *response*. Consequently, you are permitted to call `request.getSession` only when it would be legal to set HTTP response headers: before any document content has been sent (i.e., flushed or committed) to the client.

Core Approach

*Call `request.getSession` **before** you send any document content to the client.*

Now, if you plan to add data to the session regardless of whether data was there already, `getSession()` (or, equivalently, `getSession(true)`) is the appropriate method call because it creates a new session if no session already exists. However, suppose that you merely want to print out information on what is already in the session, as you might at a "View Cart" page at an e-commerce site. In such a case, it is wasteful to create a new session when no session exists already. So, you can use `getSession(false)`, which returns `null` if no session already exists for the current client. Here is an example.

```
HttpSession session = request.getSession(false);
if (session == null) {
  printMessageSayingCartIsEmpty();
} else {
  extractCartAndPrintContents(session);
}
```

Looking Up Information Associated with a Session

`HttpSession` objects live on the server; they don't go back and forth over the network; they're just automatically associated with the client by a behind-the-scenes mechanism like cookies or URL rewriting. These session objects have a built-in data structure (a hash table) in which you can store any number of keys and associated values. You use `session.getAttribute("key")` to look up a previously stored value. The return type is `Object`, so you must do a typecast to whatever more specific type of data was associated with that attribute name in the session. The return value is `null` if there is no such attribute, so you need to check for `null` before calling methods on objects associated with sessions.

Here's a representative example.

```
HttpSession session = request.getSession();
SomeClass value =
  (SomeClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
  value = new SomeClass(...);
  session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

In most cases, you have a specific attribute name in mind and want to find the value (if any) already associated with that name. However, you can also discover all the attribute names in a given session by calling `getAttributeNames`, which returns an `Enumeration`.

Associating Information with a Session

As discussed in the previous subsection, you *read* information associated with a session by using `getAttribute`. To *specify* information, use `setAttribute`. To let your values perform side effects when they are stored in a session, simply have the object you are associating with the session implement the `HttpSessionBindingListener` interface. That way, every time `setAttribute` is called on one of those objects, its `valueBound` method is called immediately afterward.

Be aware that `setAttribute` replaces any previous values; to remove a value without supplying a replacement, use `removeAttribute`. This method triggers the `valueUnbound` method of any values that implement `HttpSessionBindingListener`.

Following is an example of adding information to a session. You can add information in two ways: by adding a new session attribute (as with the bold line in the example) or by augmenting an object that is already in the session (as in the last line of the example). This distinction is fleshed out in the examples of Sections 9.7 and 9.8, which contrast the use of immutable and mutable objects as session attributes.

```
HttpSession session = request.getSession();
SomeClass value =
  (SomeClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
  value = new SomeClass(...);
  session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

In general, session attributes merely have to be of type `Object` (i.e., they can be anything other than `null` or a primitive like `int`, `double`, or `boolean`). However, some application servers support distributed Web applications in which an application is shared across a cluster of physical servers. Session tracking needs to still work in such a case, so the system needs to be able to move session objects from machine to machine. Thus, if you run in such an environment and you mark your Web application as being distributable, you must meet the additional requirement that session attributes implement the `Serializable` interface.

Discarding Session Data

When you are done with a user's session data, you have three options.

- **Remove only the data your servlet created.** You can call `removeAttribute("key")` to discard the value associated with the specified key. This is the most common approach.

- **Delete the whole session (in the current Web application).** You can call `invalidate` to discard an entire session. Just remember that doing so causes all of that user's session data to be lost, not just the session data that your servlet or JSP page created. So, *all* the servlets and JSP pages in a Web application have to agree on the cases for which `invalidate` may be called.
- **Log the user out and delete all sessions belonging to him or her.** Finally, in servers that support servlets 2.4 and JSP 2.0, you can call `logout` to log the client out of the Web server and invalidate all sessions (at most one per Web application) associated with that user. Again, since this action affects servlets other than your own, be sure to coordinate use of the `logout` command with the other developers at your site.

9.3 The Session-Tracking API

Although the session attributes (i.e., the user data) are the pieces of session information you care most about, other information is sometimes useful as well. Here is a summary of the methods available in the `HttpSession` class.

public Object getAttribute(String name)

This method extracts a previously stored value from a session object. It returns `null` if no value is associated with the given name.

public Enumeration getAttributeNames()

This method returns the names of all attributes in the session.

public void setAttribute(String name, Object value)

This method associates a value with a name. If the object supplied to `setAttribute` implements the `HttpSessionBindingListener` interface, the object's `valueBound` method is called after it is stored in the session. Similarly, if the previous value implements `HttpSessionBindingListener`, its `valueUnbound` method is called.

public void removeAttribute(String name)

This method removes any values associated with the designated name. If the value being removed implements `HttpSessionBindingListener`, its `valueUnbound` method is called.

public void invalidate()
This method invalidates the session and unbinds all objects associated with it. Use this method with caution; remember that sessions are associated with users (i.e., clients), not with individual servlets or JSP pages. So, if you invalidate a session, you might be destroying data that another servlet or JSP page is using.

public void logout()
This method logs the client out of the Web server and invalidates *all* sessions associated with that client. The scope of the logout is the same as the scope of the authentication. For example, if the server implements single sign-on, calling `logout` logs the client out of all Web applications on the server and invalidates all sessions (at most one per Web application) associated with the client. For details, see the chapters on Web application security in Volume 2 of this book.

public String getId()
This method returns the unique identifier generated for each session. It is useful for debugging or logging or, in rare cases, for programmatically moving values out of memory and into a database (however, some J2EE servers can do this automatically).

public boolean isNew()
This method returns `true` if the client (browser) has never seen the session, usually because the session was just created rather than being referenced by an incoming client request. It returns `false` for preexisting sessions. The main reason for mentioning this method is to steer you away from it: `isNew` is much less useful than it appears at first glance. Many beginning developers try to use `isNew` to determine whether users have been to their servlet before (within the session timeout period), writing code like the following:

```
HttpSession session = request.getSession();
if (session.isNew()) {
  doStuffForNewbies();
} else {
  doStuffForReturnVisitors();  // Wrong!
}
```

Wrong! Yes, if `isNew` returns `true`, then as far as you can tell this is the user's first visit (at least within the session timeout). But if `isNew` returns `false`, it merely shows that they have visited the Web application before, not that they have visited your servlet or JSP page before.

public long getCreationTime()
This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was first built. To get a value useful for printing, pass the value to the `Date` constructor or the `setTimeInMillis` method of `GregorianCalendar`.

public long getLastAccessedTime()
This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was last accessed by the client.

public int getMaxInactiveInterval()
public void setMaxInactiveInterval(int seconds)
These methods get or set the length of time, in seconds, that a session should go without access before being automatically invalidated. A negative value specifies that the session should never time out. Note that the timeout is maintained on the server and is *not* the same as the cookie expiration date. For one thing, sessions are normally based on in-memory cookies, not persistent cookies, so there *is* no expiration date. Even if you intercepted the `JSESSIONID` cookie and sent it out with an expiration date, browser sessions and server sessions are two distinct things. For details on the distinction, see the next section.

9.4 Browser Sessions vs. Server Sessions

By default, session-tracking is based on cookies that are stored in the browser's memory, not written to disk. Thus, unless the servlet explicitly reads the incoming `JSESSIONID` cookie, sets the maximum age and path, and sends it back out, quitting the browser results in the session being broken: the client will not be able to access the session again. The problem, however, is that the server does not know that the browser was closed and thus the server has to maintain the session in memory until the inactive interval has been exceeded.

Consider a physical shopping trip to a Wal-Mart store. You browse around and put some items in a physical shopping cart, then leave that shopping cart at the end of an aisle while you look for another item. A clerk walks up and sees the shopping cart. Can he reshelve the items in it? No—you are probably still shopping and will come back for the cart soon. What if you realize that you have lost your wallet, so you get in your car and drive home? Can the clerk reshelve the items in your shopping cart now? Again, no—the clerk presumably does not *know* that you have left the store. So, what can the clerk do? He can keep an eye on the cart, and if nobody has touched it for some period of time, he can then conclude that it is abandoned and take the items out of it. The only exception is if you brought the cart to him and said "I'm sorry, I left my wallet at home, so I have to leave."

The analogous situation in the servlet world is one in which the server is trying to decide if it can throw away your `HttpSession` object. Just because you are not currently using the session does not mean the server can throw it away. Maybe you will be back (submit a new request) soon? If you quit your browser, thus causing the browser-session-level cookies to be lost, the session is effectively broken. But, as with the case of getting in your car and leaving Wal-Mart, the server does not *know* that you quit your browser. So, the server still has to wait for a period of time to see if the session has been abandoned. Sessions automatically become inactive when the amount of time between client accesses exceeds the interval specified by `getMaxInactiveInterval`. When this happens, objects stored in the `HttpSession` object are removed (unbound). Then, if those objects implement the `HttpSessionBindingListener` interface, they are automatically notified. The one exception to the "the server waits until sessions time out" rule is if `invalidate` or `logout` is called. This is akin to your explicitly telling the Wal-Mart clerk that you are leaving, so the server can immediately remove all the items from the session and destroy the session object.

9.5 Encoding URLs Sent to the Client

By default, servlet containers (engines) use cookies as the underlying mechanism for session tracking. But suppose you reconfigure your server to use URL rewriting instead? How will your code have to change?

The goods news: your core session-tracking code does not need to change at all.

The bad news: lots of *other* code has to change. In particular, if any of your pages contain links back to your own site, you have to explicitly add the session data to the URL. Now, the servlet API provides methods to add this information to whatever URL you specify. The problem is that you have to *call* these methods; it is not technically feasible for the system to examine the output of all of your servlets and JSP pages, figure out which parts contain hyperlinks back to your site, and modify those URLs. You have to tell it which URLs to modify. This requirement means that you cannot have static HTML pages if you use URL rewriting for session tracking, or at least you cannot have static HTML pages that refer to your own site. This is a significant burden in many applications, but worth the price in a few.

Core Warning

If you use URL rewriting for session tracking, most or all of your pages will have to be dynamically generated. You cannot have any static HTML pages that contain hyperlinks to dynamic pages at your site.

There are two possible situations in which you might use URLs that refer to your own site.

The first one is where the URLs are embedded in the Web page that the servlet generates. These URLs should be passed through the `encodeURL` method of `HttpServletResponse`. The method determines if URL rewriting is currently in use and appends the session information only if necessary. The URL is returned unchanged otherwise.

Here's an example:

```
String originalURL = someRelativeOrAbsoluteURL;
String encodedURL = response.encodeURL(originalURL);
out.println("<A HREF=\"" + encodedURL + "\">...</A>");
```

The second situation in which you might use a URL that refers to your own site is in a `sendRedirect` call (i.e., placed into the `Location` response header). In this second situation, different rules determine whether session information needs to be attached, so you cannot use `encodeURL`. Fortunately, `HttpServletResponse` supplies an `encodeRedirectURL` method to handle that case. Here's an example:

```
String originalURL = someURL;
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

If you think there is a reasonable likelihood that your Web application will eventually use URL rewriting instead of cookies, it is good practice to plan ahead and encode URLs that reference your own site.

9.6 A Servlet That Shows Per-Client Access Counts

Listing 9.1 presents a simple servlet that shows basic information about the client's session. When the client connects, the servlet uses `request.getSession` either to retrieve the existing session or, if there is no session, to create a new one. The servlet then looks for an attribute called `accessCount` of type `Integer`. If it cannot find such an attribute, it uses 0 as the number of previous accesses. This value is then incremented and associated with the session by `setAttribute`. Finally, the servlet prints a small HTML table showing information about the session.

Note that `Integer` is an *immutable* (nonmodifiable) data structure: once built, it cannot be changed. That means you have to allocate a new `Integer` object on each request, then use `setAttribute` to replace the old object. The following snippet shows the general approach for session tracking when an immutable object will be stored.

```
HttpSession session = request.getSession();
SomeImmutableClass value =
  (SomeImmutableClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
  value = new SomeImmutableClass(...);
} else {
  value = new SomeImmutableClass(calculatedFrom(value));
}
session.setAttribute("someIdentifier", value);
doSomethingWith(value);
```

This approach contrasts with the approach used in the next section (Section 9.7) with a mutable (modifiable) data structure. In that approach, the object is allocated and `setAttribute` is called only when there is no such object already in the session. That is, the *contents* of the object change each time, but the session maintains the same object *reference*.

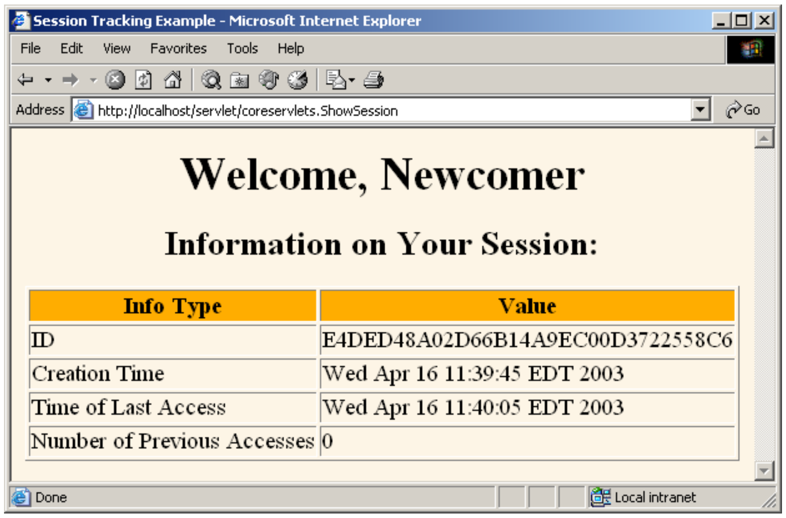
Figures 9–1 and 9–2 show the servlet on the initial visit and after the page was reloaded several times.

Listing 9.1 ShowSession.java

```
package coreservlets;

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

/** Servlet that uses session tracking to keep per-client
 *  access counts. Also shows other info about the session.
 */

public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession();
```

Listing 9.1 ShowSession.java *(continued)*

```
    String heading;
    Integer accessCount =
      (Integer)session.getAttribute("accessCount");
    if (accessCount == null) {
      accessCount = new Integer(0);
      heading = "Welcome, Newcomer";
    } else {
      heading = "Welcome Back";
      accessCount = new Integer(accessCount.intValue() + 1);
    }
    // Integer is an immutable data structure. So, you
    // cannot modify the old one in-place. Instead, you
    // have to allocate a new one and redo setAttribute.
    session.setAttribute("accessCount", accessCount);
    PrintWriter out = response.getWriter();
    String title = "Session Tracking Example";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                "<H1>" + heading + "</H1>\n" +
                "<H2>Information on Your Session:</H2>\n" +
                "<TABLE BORDER=1>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "  <TH>Info Type<TH>Value\n" +
                "<TR>\n" +
                "  <TD>ID\n" +
                "  <TD>" + session.getId() + "\n" +
                "<TR>\n" +
                "  <TD>Creation Time\n" +
                "  <TD>" +
                new Date(session.getCreationTime()) + "\n" +
                "<TR>\n" +
                "  <TD>Time of Last Access\n" +
                "  <TD>" +
                new Date(session.getLastAccessedTime()) + "\n" +
                "<TR>\n" +
                "  <TD>Number of Previous Accesses\n" +
                "  <TD>" + accessCount + "\n" +
                "</TABLE>\n" +
                "</CENTER></BODY></HTML>");
  }
}
```

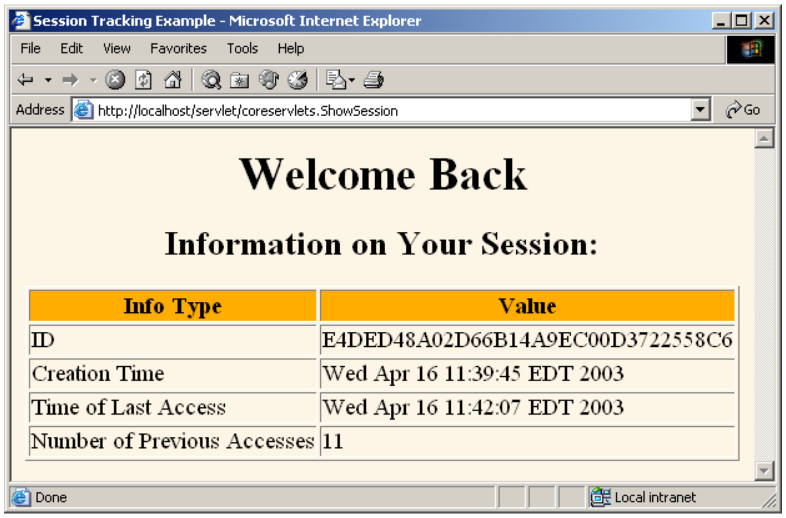

Session Tracking Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.ShowSession
Go
Done
Local intranet

Welcome, Newcomer

Information on Your Session:

Info Type	Value
ID	E4DED48A02D66B14A9EC00D3722558C6
Creation Time	Wed Apr 16 11:39:45 EDT 2003
Time of Last Access	Wed Apr 16 11:40:05 EDT 2003
Number of Previous Accesses	0

Figure 9–1 First visit by client to `ShowSession` servlet.


Session Tracking Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.ShowSession
Go
Done
Local intranet

Welcome Back

Information on Your Session:

Info Type	Value
ID	E4DED48A02D66B14A9EC00D3722558C6
Creation Time	Wed Apr 16 11:39:45 EDT 2003
Time of Last Access	Wed Apr 16 11:42:07 EDT 2003
Number of Previous Accesses	11

Figure 9–2 Twelfth visit to `ShowSession` servlet. Access count for this client is independent of number of visits by other clients.

9.7 Accumulating a List of User Data

The example of the previous section (Section 9.6) stores user-specific data in the user's `HttpSession` object. The object stored (an `Integer`) is an immutable data structure: one that cannot be modified. Consequently, a new `Integer` is allocated for each request, and that new object is stored in the session with `setAttribute`, overwriting the previous value.

Another common approach is to use a *mutable* data structure such as an array, `List`, `Map`, or application-specific data structure that has writable fields (instance variables). With this approach, you do not need to call `setAttribute` except when the object is first allocated. Here is the basic template:

```
HttpSession session = request.getSession();
SomeMutableClass value =
  (SomeMutableClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
  value = new SomeMutableClass(...);
  session.setAttribute("someIdentifier", value);
}
value.updateInternalState(...);
doSomethingWith(value);
```

Mutable data structures are most commonly used to maintain a set of data associated with the user. In this section we present a simplified example in which we maintain a basic list of items that each user has purchased. In the next section (Section 9.8), we present a full-fledged shopping cart example. Most of the code in that example is for automatically building the Web pages that display the items and for the shopping cart itself. Although these application-specific pieces can be somewhat complicated, the basic session tracking is quite simple. Even so, it is useful to see the fundamental approach without the distractions of the application-specific pieces. That's the purpose of the example here.

Listing 9.2 shows an application that uses a simple `ArrayList` (the Java 2 platform's replacement for `Vector`) to keep track of the items each user has purchased. In addition to finding or creating the session and inserting the newly purchased item (the value of the `newItem` request parameter) into it, this example outputs a bulleted list of whatever items are in the "cart" (i.e., the `ArrayList`). Notice that the code that outputs this list is synchronized on the `ArrayList`. This precaution is worth taking, but you should be aware that the circumstances that make synchronization necessary are exceedingly rare. Since each user has a separate session, the only way a race condition could occur is if the same user submits two purchases in rapid succession. Although unlikely, this *is* possible, so synchronization is worthwhile.

Listing 9.2 ShowItems.java

```
package coreservlets;

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

/** Servlet that displays a list of items being ordered.
 *  Accumulates them in an ArrayList with no attempt at
 *  detecting repeated items. Used to demonstrate basic
 *  session tracking.
 */

public class ShowItems extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    ArrayList previousItems =
      (ArrayList)session.getAttribute("previousItems");
    if (previousItems == null) {
      previousItems = new ArrayList();
      session.setAttribute("previousItems", previousItems);
    }
    String newItem = request.getParameter("newItem");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Items Purchased";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>");
    synchronized(previousItems) {
      if (newItem != null) {
        previousItems.add(newItem);
      }
      if (previousItems.size() == 0) {
        out.println("<I>No items</I>");
```

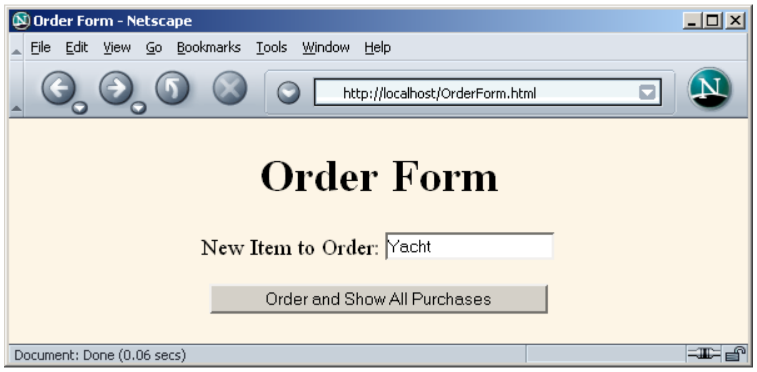
Listing 9.2 ShowItems.java *(continued)*

```
      } else {
        out.println("<UL>");
        for(int i=0; i<previousItems.size(); i++) {
          out.println("<LI>" + (String)previousItems.get(i));
        }
        out.println("</UL>");
      }
    }
    out.println("</BODY></HTML>");
  }
}
```

Listing 9.3 shows an HTML form that collects values of the `newItem` parameter and submits them to the servlet. Figure 9–3 shows the result of the form; Figures 9–4 and 9–5 show the results of the servlet before the order form is visited and after it is visited several times, respectively.

Listing 9.3 OrderForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Order Form</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1>Order Form</H1>
<FORM ACTION="servlet/coreservlets.ShowItems">
  New Item to Order:
  <INPUT TYPE="TEXT" NAME="newItem" VALUE="Yacht"><P>
  <INPUT TYPE="SUBMIT" VALUE="Order and Show All Purchases">
</FORM>
</CENTER></BODY></HTML>
```

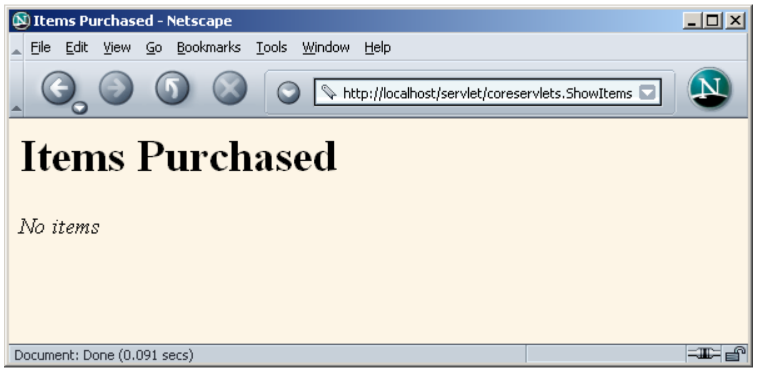

Order Form - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/OrderForm.html
Document: Done (0.06 secs)

Order Form

New Item to Order: Yacht

Order and Show All Purchases

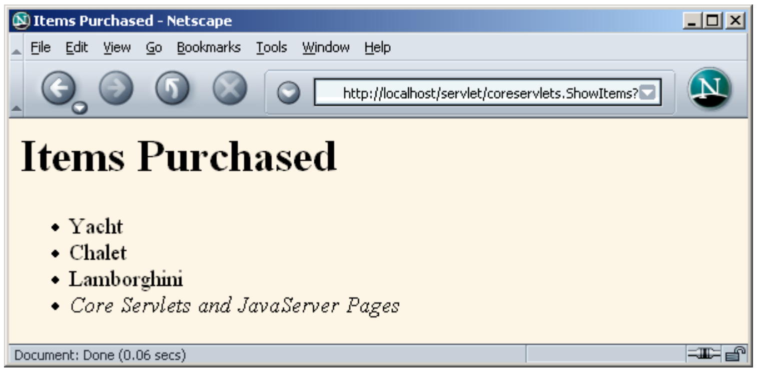
Figure 9–3 Front end to the item display servlet.


Items Purchased - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.ShowItems
Document: Done (0.091 secs)

Items Purchased

No items

Figure 9–4 The item display servlet before any purchases are made.


Items Purchased - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/servlet/coreservlets.ShowItems?
Document: Done (0.06 secs)

Items Purchased

- Yacht
- Chalet
- Lamborghini
- *Core Servlets and JavaServer Pages*

Figure 9–5 The item display servlet after a few valuable items are purchased.

9.8 An Online Store with a Shopping Cart and Session Tracking

This section gives an extended example of how you might build an online store that uses session tracking. The first subsection shows how to build pages that display items for sale. The code for each display page simply lists the page title and the identifiers of the items listed on the page. The actual page is then built automatically by methods in the parent class, based on item descriptions stored in the catalog. The second subsection shows the page that handles the orders. It uses session tracking to associate a shopping cart with each user and permits the user to modify orders for any previously selected item. The third subsection presents the implementation of the shopping cart, the data structures representing individual items and orders, and the catalog.

Creating the Front End

Listing 9.4 presents an abstract base class used as a starting point for servlets that want to display items for sale. It takes the identifiers for the items for sale, looks them up in the catalog, and uses the descriptions and prices found there to present an order page to the user. Listing 9.5 (with the result shown in Figure 9–6) and Listing 9.6 (with the result shown in Figure 9–7) show how easy it is to build actual pages with this parent class.

Listing 9.4 CatalogPage.java

```
package coreservlets;

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

/** Base class for pages showing catalog entries.
 *  Servlets that extend this class must specify
 *  the catalog entries that they are selling and the page
 *  title <I>before</I> the servlet is ever accessed. This
 *  is done by putting calls to setItems and setTitle
 *  in init.
 */
```

Listing 9.4 CatalogPage.java *(continued)*

```
public abstract class CatalogPage extends HttpServlet {
  private CatalogItem[] items;
  private String[] itemIDs;
  private String title;

  /** Given an array of item IDs, look them up in the
   *  Catalog and put their corresponding CatalogItem entry
   *  into the items array. The CatalogItem contains a short
   *  description, a long description, and a price,
   *  using the item ID as the unique key.
   *  <P>
   *  Servlets that extend CatalogPage <B>must</B> call
   *  this method (usually from init) before the servlet
   *  is accessed.
   */

  protected void setItems(String[] itemIDs) {
    this.itemIDs = itemIDs;
    items = new CatalogItem[itemIDs.length];
    for(int i=0; i<items.length; i++) {
      items[i] = Catalog.getItem(itemIDs[i]);
    }
  }

  /** Sets the page title, which is displayed in
   *  an H1 heading in resultant page.
   *  <P>
   *  Servlets that extend CatalogPage <B>must</B> call
   *  this method (usually from init) before the servlet
   *  is accessed.
   */

  protected void setTitle(String title) {
    this.title = title;
  }

  /** First display title, then, for each catalog item,
   *  put its short description in a level-two (H2) heading
   *  with the price in parentheses and long description
   *  below. Below each entry, put an order button
   *  that submits info to the OrderPage servlet for
   *  the associated catalog entry.
   *  <P>
   *  To see the HTML that results from this method, do
   *  "View Source" on KidsBooksPage or TechBooksPage, two
   *  concrete classes that extend this abstract class.
   */
```

Listing 9.4 CatalogPage.java *(continued)*

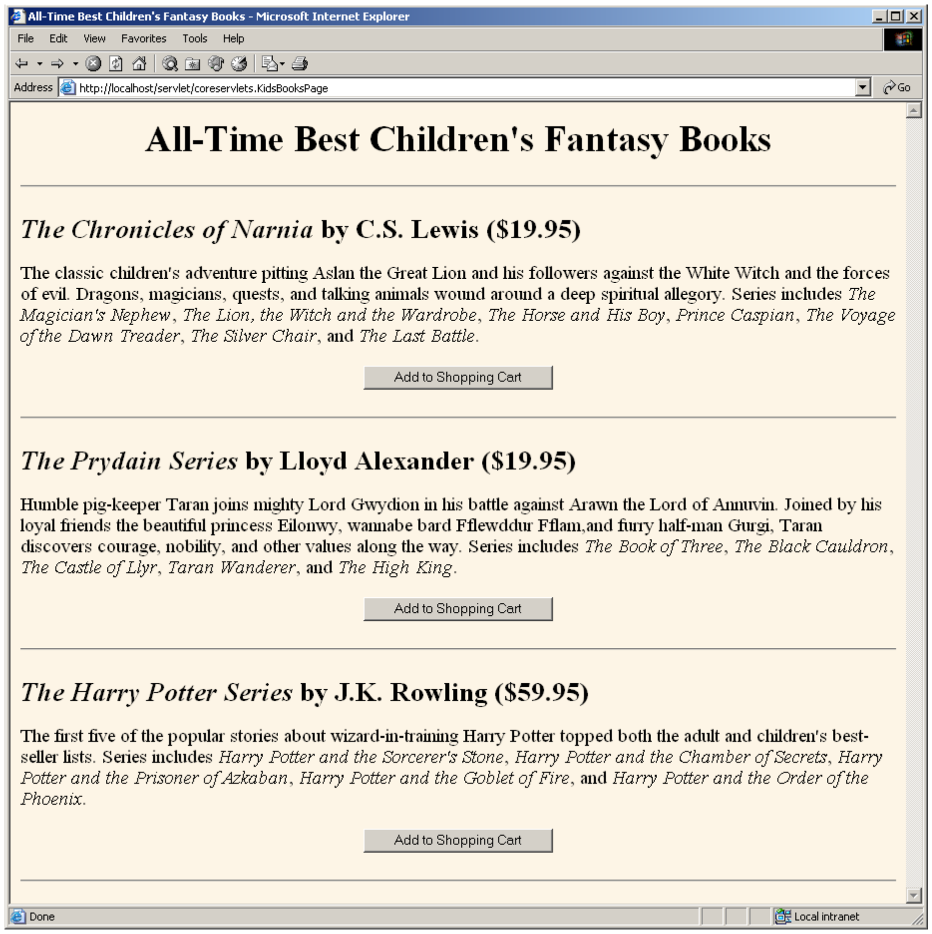
```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    if (items == null) {
      response.sendError(response.SC_NOT_FOUND,
                         "Missing Items.");
      return;
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
    CatalogItem item;
    for(int i=0; i<items.length; i++) {
      out.println("<HR>");
      item = items[i];
      // Show error message if subclass lists item ID
      // that's not in the catalog.
      if (item == null) {
        out.println("<FONT COLOR=\"RED\">" +
                    "Unknown item ID " + itemIDs[i] +
                    "</FONT>");
      } else {
        out.println();
        String formURL =
          "/servlet/coreservlets.OrderPage";
        // Pass URLs that reference own site through encodeURL.
        formURL = response.encodeURL(formURL);
        out.println
          ("<FORM ACTION=\"" + formURL + "\">\n" +
           "<INPUT TYPE=\"HIDDEN\" NAME=\"itemID\" " +
           "       VALUE=\"" + item.getItemID() + "\">\n" +
           "<H2>" + item.getShortDescription() +
           " ($" + item.getCost() + ")</H2>\n" +
           item.getLongDescription() + "\n" +
           "<P>\n<CENTER>\n" +
           "<INPUT TYPE=\"SUBMIT\" " +
           "VALUE=\"Add to Shopping Cart\">\n" +
           "</CENTER>\n<P>\n</FORM>");
      }
    }
    out.println("<HR>\n</BODY></HTML>");
  }
}
```

Listing 9.5 KidsBooksPage.java

```
package coreservlets;

/** A specialization of the CatalogPage servlet that
 *  displays a page selling three famous kids-book series.
 *  Orders are sent to the OrderPage servlet.
 */

public class KidsBooksPage extends CatalogPage {
  public void init() {
    String[] ids = { "lewis001", "alexander001", "rowling001" };
    setItems(ids);
    setTitle("All-Time Best Children's Fantasy Books");
  }
}
```

Listing 9.6 TechBooksPage.java

```
package coreservlets;

/** A specialization of the CatalogPage servlet that
 *  displays a page selling two famous computer books.
 *  Orders are sent to the OrderPage servlet.
 */

public class TechBooksPage extends CatalogPage {
  public void init() {
    String[] ids = { "hall001", "hall002" };
    setItems(ids);
    setTitle("All-Time Best Computer Books");
  }
}
```


All-Time Best Children's Fantasy Books - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.KidsBooksPage
Go
Done
Local intranet

All-Time Best Children's Fantasy Books

The Chronicles of Narnia by C.S. Lewis ($19.95)

The classic children's adventure pitting Aslan the Great Lion and his followers against the White Witch and the forces of evil. Dragons, magicians, quests, and talking animals wound around a deep spiritual allegory. Series includes *The Magician's Nephew*, *The Lion, the Witch and the Wardrobe*, *The Horse and His Boy*, *Prince Caspian*, *The Voyage of the Dawn Treader*, *The Silver Chair*, and *The Last Battle*.

Add to Shopping Cart

The Prydain Series by Lloyd Alexander ($19.95)

Humble pig-keeper Taran joins mighty Lord Gwydion in his battle against Arawn the Lord of Annuvin. Joined by his loyal friends the beautiful princess Eilonwy, wannabe bard Fflewddur Fflam,and furry half-man Gurgi, Taran discovers courage, nobility, and other values along the way. Series includes *The Book of Three*, *The Black Cauldron*, *The Castle of Llyr*, *Taran Wanderer*, and *The High King*.

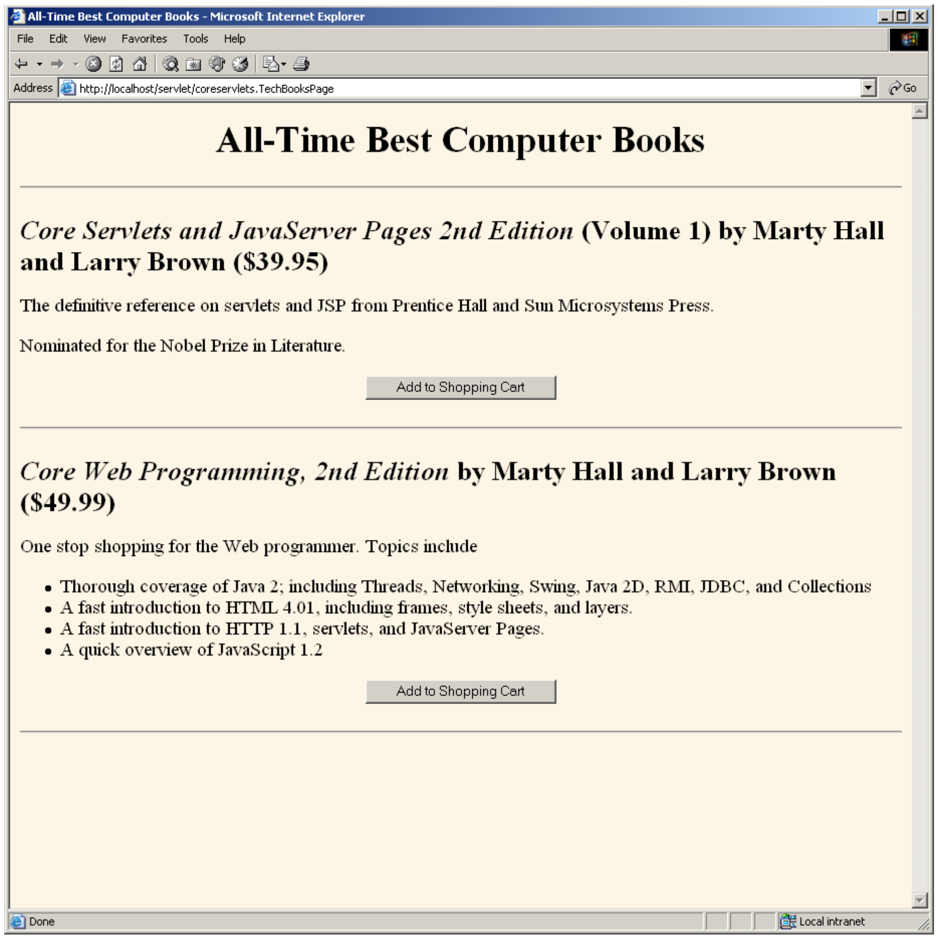
Add to Shopping Cart

The Harry Potter Series by J.K. Rowling ($59.95)

The first five of the popular stories about wizard-in-training Harry Potter topped both the adult and children's best-seller lists. Series includes *Harry Potter and the Sorcerer's Stone*, *Harry Potter and the Chamber of Secrets*, *Harry Potter and the Prisoner of Azkaban*, *Harry Potter and the Goblet of Fire*, and *Harry Potter and the Order of the Phoenix*.

Add to Shopping Cart

Figure 9–6 Result of the `KidsBooksPage` servlet.


All-Time Best Computer Books - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.TechBooksPage
Go
Done
Local intranet

All-Time Best Computer Books

Core Servlets and JavaServer Pages 2nd Edition (Volume 1) by Marty Hall and Larry Brown ($39.95)

The definitive reference on servlets and JSP from Prentice Hall and Sun Microsystems Press.

Nominated for the Nobel Prize in Literature.

Add to Shopping Cart

Core Web Programming, 2nd Edition by Marty Hall and Larry Brown ($49.99)

One stop shopping for the Web programmer. Topics include

- Thorough coverage of Java 2; including Threads, Networking, Swing, Java 2D, RMI, JDBC, and Collections
- A fast introduction to HTML 4.01, including frames, style sheets, and layers.
- A fast introduction to HTTP 1.1, servlets, and JavaServer Pages.
- A quick overview of JavaScript 1.2

Add to Shopping Cart

Figure 9–7 Result of the `TechBooksPage` servlet.

Handling the Orders

Listing 9.7 shows the servlet that handles the orders coming from the various catalog pages shown in the previous subsection. It uses session tracking to associate a shopping cart with each user. Since each user has a separate session, it is unlikely that multiple threads will be accessing the same shopping cart simultaneously. However, if you were paranoid, you could conceive of a few circumstances in which concurrent access could occur, such as when a single user has multiple browser windows open and sends updates from more than one in quick succession. So, just to be safe, the code synchronizes access based upon the session object. This synchronization prevents other threads that use the same session from accessing the data concurrently, while still allowing simultaneous requests from different users to proceed. Figures 9–8 and 9–9 show some typical results.

Listing 9.7 OrderPage.java

```
package coreservlets;

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

/** Shows all items currently in ShoppingCart. Clients
 *  have their own session that keeps track of which
 *  ShoppingCart is theirs. If this is their first visit
 *  to the order page, a new shopping cart is created.
 *  Usually, people come to this page by way of a page
 *  showing catalog entries, so this page adds an additional
 *  item to the shopping cart. But users can also
 *  bookmark this page, access it from their history list,
 *  or be sent back to it by clicking on the "Update Order"
 *  button after changing the number of items ordered.
 */

public class OrderPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    ShoppingCart cart;
    synchronized(session) {
      cart = (ShoppingCart)session.getAttribute("shoppingCart");
      // New visitors get a fresh shopping cart.
      // Previous visitors keep using their existing cart.
      if (cart == null) {
        cart = new ShoppingCart();
        session.setAttribute("shoppingCart", cart);
      }
```

Listing 9.7 OrderPage.java *(continued)*

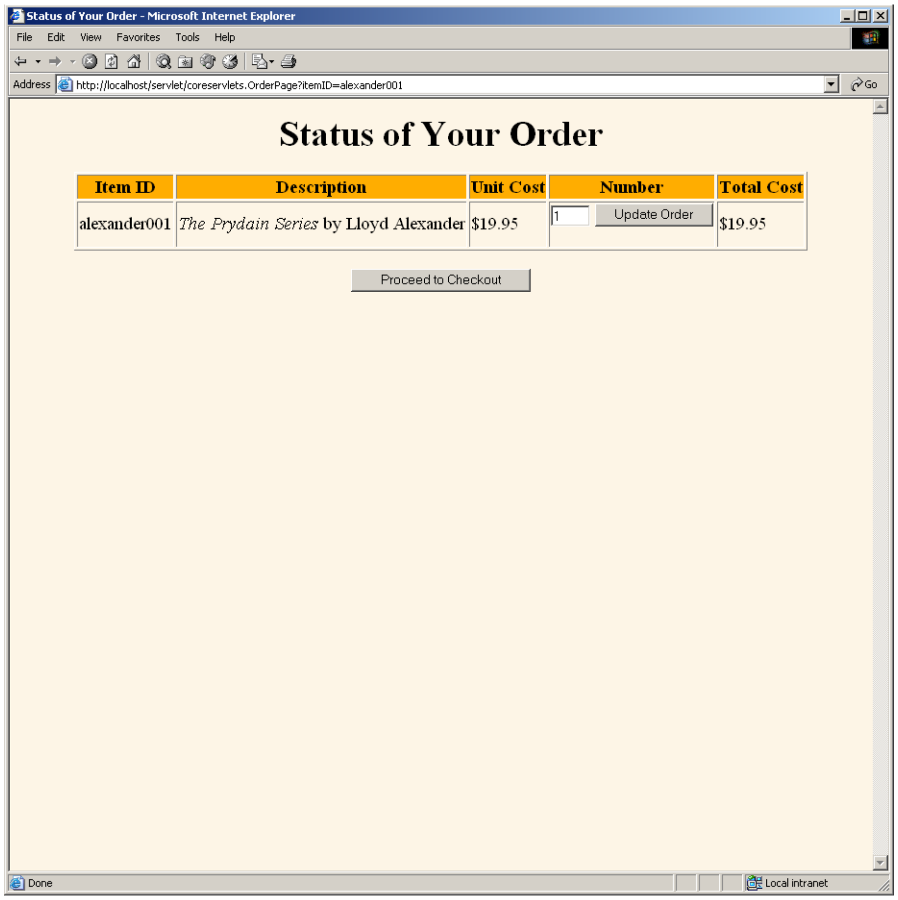
```
      if (itemID != null) {
        String numItemsString =
          request.getParameter("numItems");
        if (numItemsString == null) {
          // If request specified an ID but no number,
          // then customers came here via an "Add Item to Cart"
          // button on a catalog page.
          cart.addItem(itemID);
        } else {
          // If request specified an ID and number, then
          // customers came here via an "Update Order" button
          // after changing the number of items in order.
          // Note that specifying a number of 0 results
          // in item being deleted from cart.
          int numItems;
          try {
            numItems = Integer.parseInt(numItemsString);
          } catch(NumberFormatException nfe) {
            numItems = 1;
          }
          cart.setNumOrdered(itemID, numItems);
        }
      }
    }
    // Whether or not the customer changed the order, show
    // order status.
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Status of Your Order";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
    synchronized(session) {
      List itemsOrdered = cart.getItemsOrdered();
      if (itemsOrdered.size() == 0) {
        out.println("<H2><I>No items in your cart...</I></H2>");
      } else {
        // If there is at least one item in cart, show table
        // of items ordered.
        out.println
```

Listing 9.7 OrderPage.java *(continued)*

```
String itemID = request.getParameter("itemID");
    ("<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
     "<TR BGCOLOR=\"#FFAD00\">\n" +
     "  <TH>Item ID<TH>Description\n" +
     "  <TH>Unit Cost<TH>Number<TH>Total Cost");
  ItemOrder order;
  // Rounds to two decimal places, inserts dollar
  // sign (or other currency symbol), etc., as
  // appropriate in current Locale.
  NumberFormat formatter =
    NumberFormat.getCurrencyInstance();
  // For each entry in shopping cart, make
  // table row showing ID, description, per-item
  // cost, number ordered, and total cost.
  // Put number ordered in textfield that user
  // can change, with "Update Order" button next
  // to it, which resubmits to this same page
  // but specifying a different number of items.
  for(int i=0; i<itemsOrdered.size(); i++) {
    order = (ItemOrder)itemsOrdered.get(i);
    out.println
      ("<TR>\n" +
       "  <TD>" + order.getItemID() + "\n" +
       "  <TD>" + order.getShortDescription() + "\n" +
       "  <TD>" +
       formatter.format(order.getUnitCost()) + "\n" +
       "  <TD>" +
       "<FORM>\n" +  // Submit to current URL
       "<INPUT TYPE=\"HIDDEN\" NAME=\"itemID\"\n" +
       "       VALUE=\"" + order.getItemID() + "\">\n" +
       "<INPUT TYPE=\"TEXT\" NAME=\"numItems\"\n" +
       "       SIZE=3 VALUE=\"" +
       order.getNumItems() + "\">\n" +
       "<SMALL>\n" +
       "<INPUT TYPE=\"SUBMIT\"\n "+
       "       VALUE=\"Update Order\">\n" +
       "</SMALL>\n" +
       "</FORM>\n" +
       "  <TD>" +
       formatter.format(order.getTotalCost()));
  }
  String checkoutURL =
    response.encodeURL("../Checkout.html");
  // "Proceed to Checkout" button below table
```

Listing 9.7 OrderPage.java *(continued)*

```
        out.println
          ("</TABLE>\n" +
           "<FORM ACTION=\"" + checkoutURL + "\">\n" +
           "<BIG><CENTER>\n" +
           "<INPUT TYPE=\"SUBMIT\"\n" +
           "       VALUE=\"Proceed to Checkout\">\n" +
           "</CENTER></BIG></FORM>");
      }
      out.println("</BODY></HTML>");
    }
  }
}
```

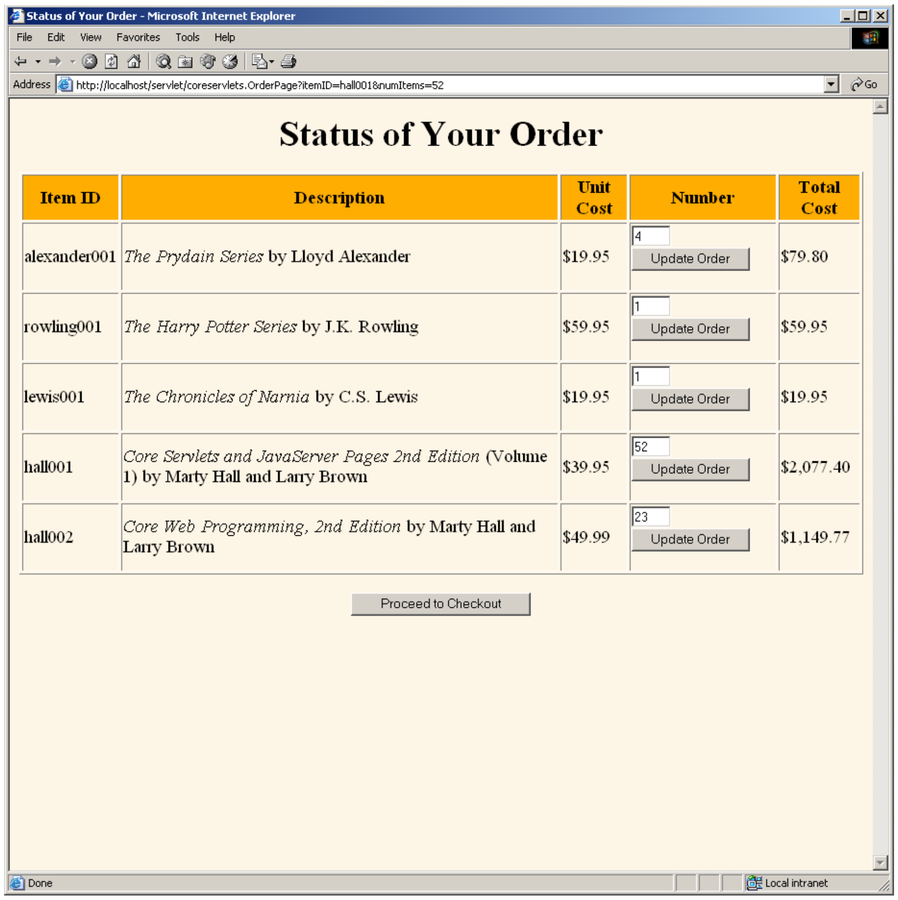


Status of Your Order

Item ID	Description	Unit Cost	Number	Total Cost
alexander001	*The Prydain Series* by Lloyd Alexander	$19.95	1 Update Order	$19.95

Status of Your Order - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.OrderPage?itemID=alexander001
Go
Proceed to Checkout
Done
Local intranet

Figure 9–8 Result of `OrderPage` servlet after user clicks on “Add to Shopping Cart” in `KidsBooksPage`.


Status of Your Order - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.OrderPage?itemID=hall001&numItems=52
Go

Status of Your Order

Item ID	Description	Unit Cost	Number	Total Cost
alexander001	*The Prydain Series* by Lloyd Alexander	$19.95	4 Update Order	$79.80
rowling001	*The Harry Potter Series* by J.K. Rowling	$59.95	1 Update Order	$59.95
lewis001	*The Chronicles of Narnia* by C.S. Lewis	$19.95	1 Update Order	$19.95
hall001	*Core Servlets and JavaServer Pages 2nd Edition* (Volume 1) by Marty Hall and Larry Brown	$39.95	52 Update Order	$2,077.40
hall002	*Core Web Programming, 2nd Edition* by Marty Hall and Larry Brown	$49.99	23 Update Order	$1,149.77

Figure 9–9 Result of `OrderPage` servlet after several additions and changes to the order.

Listing 9.8 Checkout.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Checking Out</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Checking Out</H1>
```

Listing 9.8 Checkout.html *(continued)*

```
We are sorry, but our electronic credit-card-processing
system is currently out of order. Please send a check
to:
<PRE>
  Marty Hall
  coreservlets.com, Inc.
  300 Red Brook Blvd., Suite 400
  Owings Mills, MD 21117
</PRE>
Since we have not yet calculated shipping charges, please
sign the check but do not fill in the amount. We will
generously do that for you.
</BODY></HTML>
```

Behind the Scenes: Implementing the Shopping Cart and Catalog Items

Listing 9.9 gives the shopping cart implementation. It simply maintains a `List` of orders, with methods to add and update these orders. Listing 9.10 shows the code for an individual catalog item, Listing 9.11 presents the class representing the order status of a particular item, and Listing 9.12 gives the catalog implementation.

Listing 9.9 ShoppingCart.java

```
package coreservlets;

import java.util.*;

/** A shopping cart data structure used to track orders.
 *  The OrderPage servlet associates one of these carts
 *  with each user session.
 */

public class ShoppingCart {
  private ArrayList itemsOrdered;

  /** Builds an empty shopping cart. */
```

Listing 9.9 ShoppingCart.java *(continued)*

```
public ShoppingCart() {
  itemsOrdered = new ArrayList();
}

/** Returns List of ItemOrder entries giving
 *  Item and number ordered. Declared as List instead
 *  of ArrayList so that underlying implementation
 *  can be changed later.
 */

public List getItemsOrdered() {
  return(itemsOrdered);
}

/** Looks through cart to see if it already contains
 *  an order entry corresponding to item ID. If it does,
 *  increments the number ordered. If not, looks up
 *  Item in catalog and adds an order entry for it.
 */

public synchronized void addItem(String itemID) {
  ItemOrder order;
  for(int i=0; i<itemsOrdered.size(); i++) {
    order = (ItemOrder)itemsOrdered.get(i);
    if (order.getItemID().equals(itemID)) {
      order.incrementNumItems();
      return;
    }
  }
  ItemOrder newOrder = new ItemOrder(Catalog.getItem(itemID));
  itemsOrdered.add(newOrder);
}

/** Looks through cart to find order entry corresponding
 *  to item ID listed. If the designated number
 *  is positive, sets it. If designated number is 0
 *  (or, negative due to a user input error), deletes
 *  item from cart.
 */

public synchronized void setNumOrdered(String itemID,
                                       int numOrdered) {
```

Listing 9.9 ShoppingCart.java *(continued)*

```
    ItemOrder order;
    for(int i=0; i<itemsOrdered.size(); i++) {
      order = (ItemOrder)itemsOrdered.get(i);
      if (order.getItemID().equals(itemID)) {
        if (numOrdered <= 0) {
          itemsOrdered.remove(i);
        } else {
          order.setNumItems(numOrdered);
        }
        return;
      }
    }
    ItemOrder newOrder =
      new ItemOrder(Catalog.getItem(itemID));
    itemsOrdered.add(newOrder);
  }
}
```

Listing 9.10 CatalogItem.java

```
package coreservlets;

/** Describes a catalog item for an online store. The itemID
 *  uniquely identifies the item, the short description
 *  gives brief info like the book title and author,
 *  the long description describes the item in a couple
 *  of sentences, and the cost gives the current per-item price.
 *  Both the short and long descriptions can contain HTML
 *  markup.
 */

public class CatalogItem {
  private String itemID;
  private String shortDescription;
  private String longDescription;
  private double cost;

  public CatalogItem(String itemID, String shortDescription,
                     String longDescription, double cost) {
    setItemID(itemID);
    setShortDescription(shortDescription);
    setLongDescription(longDescription);
    setCost(cost);
  }
```

Listing 9.10 CatalogItem.java *(continued)*

```
  public String getItemID() {
    return(itemID);
  }

  protected void setItemID(String itemID) {
    this.itemID = itemID;
  }

  public String getShortDescription() {
    return(shortDescription);
  }

  protected void setShortDescription(String shortDescription) {
    this.shortDescription = shortDescription;
  }

  public String getLongDescription() {
    return(longDescription);
  }

  protected void setLongDescription(String longDescription) {
    this.longDescription = longDescription;
  }

  public double getCost() {
    return(cost);
  }

  protected void setCost(double cost) {
    this.cost = cost;
  }
}
```

Listing 9.11 ItemOrder.java

```
package coreservlets;

/** Associates a catalog Item with a specific order by
 *  keeping track of the number ordered and the total price.
 *  Also provides some convenience methods to get at the
 *  CatalogItem data without extracting the CatalogItem
 *  separately.
 */
```

Listing 9.11 ItemOrder.java *(continued)*

```
public class ItemOrder {
  private CatalogItem item;
  private int numItems;

  public ItemOrder(CatalogItem item) {
    setItem(item);
    setNumItems(1);
  }

  public CatalogItem getItem() {
    return(item);
  }

  protected void setItem(CatalogItem item) {
    this.item = item;
  }

  public String getItemID() {
    return(getItem().getItemID());
  }

  public String getShortDescription() {
    return(getItem().getShortDescription());
  }

  public String getLongDescription() {
    return(getItem().getLongDescription());
  }

  public double getUnitCost() {
    return(getItem().getCost());
  }

  public int getNumItems() {
    return(numItems);
  }

  public void setNumItems(int n) {
    this.numItems = n;
  }

  public void incrementNumItems() {
    setNumItems(getNumItems() + 1);
  }
```

Listing 9.11 ItemOrder.java *(continued)*

```
  public void cancelOrder() {
    setNumItems(0);
  }

  public double getTotalCost() {
    return(getNumItems() * getUnitCost());
  }
}
```

Listing 9.12 Catalog.java

```
package coreservlets;

/** A catalog that lists the items available in inventory. */

public class Catalog {
  // This would come from a database in real life.
  // We use a static table for ease of testing and deployment.
  // See JDBC chapters for info on using databases in
  // servlets and JSP pages.
  private static CatalogItem[] items =
    { new CatalogItem
        ("hall001",
         "<I>Core Servlets and JavaServer Pages " +
           "2nd Edition</I> (Volume 1)" +
           " by Marty Hall and Larry Brown",
         "The definitive reference on servlets " +
           "and JSP from Prentice Hall and \n" +
           "Sun Microsystems Press.<P>Nominated for " +
           "the Nobel Prize in Literature.",
         39.95),
      new CatalogItem
          ("hall002",
           "<I>Core Web Programming, 2nd Edition</I> " +
             "by Marty Hall and Larry Brown",
           "One stop shopping for the Web programmer. " +
             "Topics include \n" +
             "<UL><LI>Thorough coverage of Java 2; " +
             "including Threads, Networking, Swing, \n" +
             "Java 2D, RMI, JDBC, and Collections\n" +
             "<LI>A fast introduction to HTML 4.01, " +
             "including frames, style sheets, and layers.\n" +
```

Listing 9.12 Catalog.java *(continued)*

```
          "<LI>A fast introduction to HTTP 1.1, " +
          "servlets, and JavaServer Pages.\n" +
          "<LI>A quick overview of JavaScript 1.2\n" +
          "</UL>",
        49.99),
  new CatalogItem
      ("lewis001",
        "<I>The Chronicles of Narnia</I> by C.S. Lewis",
          "The classic children's adventure pitting " +
          "Aslan the Great Lion and his followers\n" +
          "against the White Witch and the forces " +
          "of evil. Dragons, magicians, quests, \n" +
          "and talking animals wound around a deep " +
          "spiritual allegory. Series includes\n" +
          "<I>The Magician's Nephew</I>,\n" +
          "<I>The Lion, the Witch and the Wardrobe</I>,\n" +
          "<I>The Horse and His Boy</I>,\n" +
          "<I>Prince Caspian</I>,\n" +
          "<I>The Voyage of the Dawn Treader</I>,\n" +
          "<I>The Silver Chair</I>, and \n" +
          "<I>The Last Battle</I>.",
        19.95),
  new CatalogItem
      ("alexander001",
        "<I>The Prydain Series</I> by Lloyd Alexander",
          "Humble pig-keeper Taran joins mighty " +
          "Lord Gwydion in his battle against\n" +
          "Arawn the Lord of Annuvin. Joined by " +
          "his loyal friends the beautiful princess\n" +
          "Eilonwy, wannabe bard Fflewddur Fflam," +
          "and furry half-man Gurgi, Taran discovers " +
          "courage, nobility, and other values along\n" +
          "the way. Series includes\n" +
          "<I>The Book of Three</I>,\n" +
          "<I>The Black Cauldron</I>,\n" +
          "<I>The Castle of Llyr</I>,\n" +
          "<I>Taran Wanderer</I>, and\n" +
          "<I>The High King</I>.",
        19.95),
  new CatalogItem
      ("rowling001",
        "<I>The Harry Potter Series</I> by J.K. Rowling",
        "The first five of the popular stories " +
          "about wizard-in-training Harry Potter\n" +
          "topped both the adult and children's " +
```

Listing 9.12 Catalog.java *(continued)*

```
            "best-seller lists. Series includes\n" +
            "<I>Harry Potter and the Sorcerer's Stone</I>,\n" +
            "<I>Harry Potter and the Chamber of Secrets</I>,\n" +
            "<I>Harry Potter and the " +
            "Prisoner of Azkaban</I>,\n" +
            "<I>Harry Potter and the Goblet of Fire</I>, and\n" +
            "<I>Harry Potter and the "+
            "Order of the Phoenix</I>.\n",
          59.95)
        };

  public static CatalogItem getItem(String itemID) {
    CatalogItem item;
    if (itemID == null) {
      return(null);
    }
    for(int i=0; i<items.length; i++) {
      item = items[i];
      if (itemID.equals(item.getItemID())) {
        return(item);
      }
    }
    return(null);
  }
}
```