


SESSION TRACKING

Topics in This Chapter

- 
- The purpose of session tracking
 - The servlet session tracking API
 - A servlet that uses sessions to show per-client access counts
 - A reusable shopping cart class
 - An on-line store that uses session tracking, shopping carts, and pages automatically built from catalog entries

Online version of this first edition of *Core Servlets and JavaServer Pages* is free for personal use. For more information, please see:

- **Second edition of the book:**
<http://www.coreservlets.com>.
- **Sequel:**
<http://www.moreservlets.com>.
- **Servlet and JSP training courses from the author:**
<http://courses.coreservlets.com>.

Chapter

9

This chapter shows you how to use the servlet session tracking API to keep track of visitors as they move around at your site.

9.1 The Need for Session Tracking

HTTP is a “stateless” protocol: each time a client retrieves a Web page, it opens a separate connection to the Web server, and the server does not automatically maintain contextual information about a client. Even with servers that support persistent (keep-alive) HTTP connections and keep a socket open for multiple client requests that occur close together in time (see Section 7.4), 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 on-line store add an item to their shopping carts, how does the server know what’s already in them? Similarly, when clients decide to proceed to checkout, how can the server determine which previously created shopping carts are theirs?

There are three typical solutions to this problem: cookies, URL-rewriting, and hidden form fields.

Cookies

You can use HTTP cookies to store information about a shopping session, and each subsequent connection can look up the current session and then extract information about that session from some location on the server machine. For example, a servlet could do something like the following:

```
String sessionID = makeUniqueString();
Hashtable sessionInfo = new Hashtable();
Hashtable 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 data associated with that particular session. This is an excellent solution and is the most widely used approach for session handling. Still, it would be nice to have a higher-level API that handles some of these details. Even though servlets have a high-level and easy-to-use interface to cookies (see Chapter 8), a number of relatively tedious details still need to be handled in this case:

- Extracting the cookie that stores the session identifier from the other cookies (there may be many cookies, after all)
- Setting an appropriate expiration time for the cookie (sessions that are inactive for 24 hours probably should be reset)
- Associating the hash tables with each request
- Generating the unique session identifiers

Besides, due to real and perceived privacy concerns over cookies (see Section 8.2), some users disable them. So, it would be nice to have alternative implementation approaches in addition to a higher-level protocol.

URL-Rewriting

With this approach, the client appends some extra data on the end of each URL that identifies the session, and the server associates that identifier with data it has stored about that session. For example, with `http://host/path/file.html;jsessionid=1234`, the session information is attached as `jsessionid=1234`. This is also an excellent solution, and even has the advantage that it works when browsers don't support cookies or when

the user has disabled them. However, it has most of the same problems as cookies, namely, that the server-side program has a lot of straightforward but tedious processing to do. In addition, 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. And, if the user leaves the session and comes back via a bookmark or link, the session information can be lost.

Hidden Form Fields

HTML forms can have an entry that looks like the following:

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

This entry means that, when the form is submitted, the specified name and value are included in the GET or POST data. For details, see Section 16.9 (Hidden Fields). This hidden field can be used to store information about the session but it has the major disadvantage that it only works if every page is dynamically generated.

Session Tracking in Servlets

Servlets provide an outstanding technical solution: the `HttpSession` API. This high-level interface is built on top of cookies or URL-rewriting. In fact, most servers use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled. But, the servlet author doesn't need to bother with many of the 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 The Session Tracking API

Using sessions in servlets is straightforward and involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions. Finally, if you return any URLs to the clients that reference your site and URL-rewriting is being used, you need to attach the session information to the URLs.

Looking Up the HttpSession Object Associated with the Current Request

You look up the `HttpSession` object by calling the `getSession` method of `HttpServletRequest`. Behind the scenes, the system extracts a user ID from a cookie or attached URL data, then uses that 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 `getSession` returns `null`, this means that the user is not already participating in a session, so you can create a new session. Creating a new session in this case is so commonly done that there is an option to automatically create a new session if one doesn't already exist. Just pass `true` to `getSession`. Thus, your first step usually looks like this:

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

If you care whether the session existed previously or is newly created, you can use `isNew` to check.

Looking Up Information Associated with a Session

`HttpSession` objects live on the server; 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 that lets you store any number of keys and associated values. In version 2.1 and earlier of the servlet API, you use `session.getValue("attribute")` to look up a previously stored value. The return type is `Object`, so you have to do a type-cast 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.

In version 2.2 of the servlet API, `getValue` is deprecated in favor of `getAttribute` because of the better naming match with `setAttribute` (in version 2.1 the match for `getValue` is `putValue`, not `setValue`). Nevertheless, since not all commercial servlet engines yet support version 2.2, I'll use `getValue` in my examples.

Here's a representative example, assuming `ShoppingCart` is some class you've defined to store information on items being purchased (for an implementation, see Section 9.4 (An On-Line Store Using a Shopping Cart and Session Tracking)).

```

HttpSession session = request.getSession(true);
ShoppingCart cart =
    (ShoppingCart) session.getValue("shoppingCart");
if (cart == null) { // No cart already in session
    cart = new ShoppingCart();
    session.putValue("shoppingCart", cart);
}
doSomethingWith(cart);

```

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 `getValueNames`, which returns an array of strings. This method is your only option for finding attribute names in version 2.1, but in servlet engines supporting version 2.2 of the servlet specification, you can use `getAttributeNames`. That method is more consistent in that it returns an `Enumeration`, just like the `getHeaderNames` and `getParameterNames` methods of `HttpServletRequest`.

Although the data that was explicitly associated with a session is the part you care most about, there are some other pieces of information that are sometimes useful as well. Here is a summary of the methods available in the `HttpSession` class.

public Object `getValue(String name)`

public Object `getAttribute(String name)`

These methods extract a previously stored value from a session object. They return `null` if there is no value associated with the given name. Use `getValue` in version 2.1 of the servlet API. Version 2.2 supports both methods, but `getAttribute` is preferred and `getValue` is deprecated.

public void `putValue(String name, Object value)`

public void `setAttribute(String name, Object value)`

These methods associate a value with a name. Use `putValue` with version 2.1 servlets and either `setAttribute` (preferred) or `putValue` (deprecated) with version 2.2 servlets. If the object supplied to `putValue` or `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 `removeValue(String name)`

public void `removeAttribute(String name)`

These methods remove any values associated with the designated name. If the value being removed implements `HttpSessionBindingListener`,

tener, its `valueUnbound` method is called. With version 2.1 servlets, use `removeValue`. In version 2.2, `removeAttribute` is preferred, but `removeValue` is still supported (albeit deprecated) for backward compatibility.

public String[] getValueNames()

public Enumeration getAttributeNames()

These methods return the names of all attributes in the session. Use `getValueNames` in version 2.1 of the servlet specification. In version 2.2, `getValueNames` is supported but deprecated; use `getAttributeNames` instead.

public String getId()

This method returns the unique identifier generated for each session. It is sometimes used as the key name when only a single value is associated with a session, or when information about sessions is being logged.

public boolean isNew()

This method returns `true` if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns `false` for preexisting sessions.

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 out, 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 sent from the client.

public int getMaxInactiveInterval()

public void setMaxInactiveInterval(int seconds)

These methods get or set the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never time out. Note that the time out is maintained on the server and is *not* the same as the cookie expiration date, which is sent to the client.

public void invalidate()

This method invalidates the session and unbinds all objects associated with it.

Associating Information with a Session

As discussed in the previous section, you *read* information associated with a session by using `getValue` (in version 2.1 of the servlet specification) or `getAttribute` (in version 2.2). To *specify* information in version 2.1 servlets, you use `putValue`, supplying a key and a value. Use `setAttribute` in version 2.2. This is a more consistent name because it uses the `get/set` notation of JavaBeans. 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. Now, every time `putValue` or `setAttribute` is called on one of those objects, its `valueBound` method is called immediately afterward.

Be aware that `putValue` and `setAttribute` replace any previous values; if you want to remove a value without supplying a replacement, use `removeValue` in version 2.1 and `removeAttribute` in version 2.2. These methods trigger the `valueUnbound` method of any values that implement `HttpSessionBindingListener`. Sometimes you just want to replace previous values; see the `referringPage` entry in the example below for an example. Other times, you want to retrieve a previous value and augment it; for an example, see the `previousItems` entry below. This example assumes a `ShoppingCart` class with an `addItem` method to store items being ordered, and a `Catalog` class with a static `getItem` method that returns an item, given an item identifier. For an implementation, see Section 9.4 (An On-Line Store Using a Shopping Cart and Session Tracking).

```
HttpSession session = request.getSession(true);
session.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart cart =
    (ShoppingCart)session.getValue("previousItems");
if (cart == null) { // No cart already in session
    cart = new ShoppingCart();
    session.putValue("previousItems", cart);
}
String itemID = request.getParameter("itemID");
if (itemID != null) {
    cart.addItem(Catalog.getItem(itemID));
}
```

Terminating Sessions

Sessions will automatically become inactive when the amount of time between client accesses exceeds the interval specified by `getMaxInactiveInterval`. When this happens, any objects bound to the `HttpSession` object automatically get unbound. When this happens, your attached objects will automatically be notified if they implement the `HttpSessionBindingListener` interface.

Rather than waiting for sessions to time out, you can explicitly deactivate a session through the use of the session's `invalidate` method.

Encoding URLs Sent to the Client

If you are using URL-rewriting for session tracking and you send a URL that references your site to the client, you need to explicitly add on the session data. Since the servlet will automatically switch to URL-rewriting when cookies aren't supported by the client, you should routinely encode *all* URLs that reference your site. There are two possible places you might use URLs that refer to your own site. The first is when 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 will determine if URL-rewriting is currently in use and append 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 place 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, there are different rules for determining if 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; // Relative URL OK in version 2.2
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

Since you often don't know if your servlet will later become part of a series of pages that use session tracking, it is good practice for servlets to plan ahead and encode URLs that reference their site.

Core Approach

Plan ahead: pass URLs that refer to your own site through `response.encodeURL` or `response.encodeRedirectURL`, regardless of whether your servlet is using session tracking.



9.3 A Servlet Showing 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(true)` to either retrieve the existing session or, if there was no session, to create a new one. The servlet then looks for an attribute of type `Integer` called `accessCount`. 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 `putValue`. Finally, the servlet prints a small HTML table showing information about the session. 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

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

/** Simple example of session tracking. See the shopping
 *  cart example for a more detailed one.
 */

public class ShowSession extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Session Tracking Example";
        HttpSession session = request.getSession(true);
        String heading;
        // Use getAttribute instead of getValue in version 2.2.
        Integer accessCount =
```

Listing 9.1 ShowSession.java (continued)

```

        (Integer)session.getValue("accessCount");
    if (accessCount == null) {
        accessCount = new Integer(0);
        heading = "Welcome, Newcomer";
    } else {
        heading = "Welcome Back";
        accessCount = new Integer(accessCount.intValue() + 1);
    }
    // Use setAttribute instead of putValue in version 2.2.
    session.putValue("accessCount", accessCount);

    out.println(ServletUtilities.headWithTitle(title) +
        "<BODY BGCOLOR=\"#FDF5E6\">\n" +
        "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
        "<H2>Information on Your Session:</H2>\n" +
        "<TABLE BORDER=1 ALIGN=\"CENTER\">\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" +
        "</BODY></HTML>");
}

/** Handle GET and POST requests identically. */

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

```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

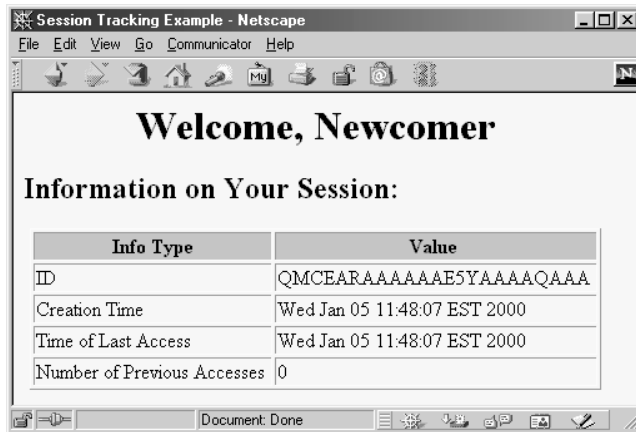


Figure 9-1 First visit to ShowSession servlet.

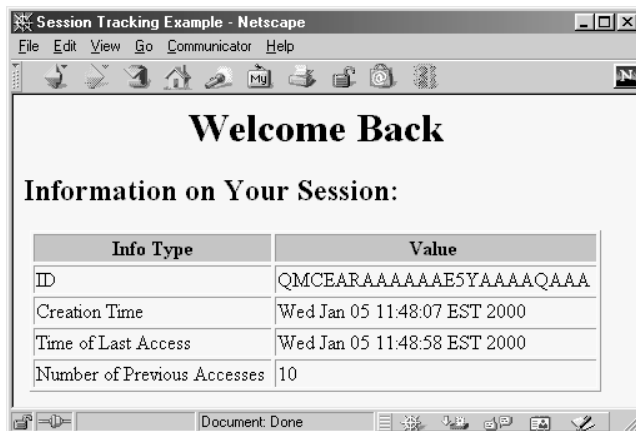
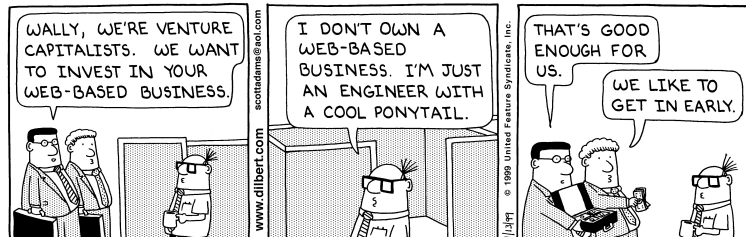


Figure 9-2 Eleventh visit to ShowSession servlet.

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

This section gives an extended example of how you might build an on-line 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 upon 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.



DILBERT reprinted by permission of United Syndicate, Inc.

Building the Front End

Listing 9.2 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 9.3 (with the result shown in Figure 9–3) and Listing 9.4 (with the result shown in Figure 9–4) show how easy it is to build actual pages with this parent class.

Listing 9.2 CatalogPage.java

```
package coreservlets;

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

/** 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
```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

Listing 9.2 CatalogPage.java (continued)

```

*   is done by putting calls to setItems and setTitle
*   in init.
*/

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

    /** Given an array of item IDs, look them up in the
     *   Catalog and put their corresponding Item entry
     *   into the items array. The Item 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 Item[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>

```

Listing 9.2 CatalogPage.java (continued)

```

* To see the HTML that results from this method, do
* "View Source" on KidsBooksPage or TechBooksPage, two
* concrete classes that extend this abstract class.
*/

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    if (items == null) {
        response.sendError(response.SC_NOT_FOUND,
                           "Missing Items.");
        return;
    }
    PrintWriter out = response.getWriter();
    out.println(ServletUtilities.headWithTitle(title) +
               "<BODY BGCOLOR=\"#FDF5E6\">\n" +
               "<H1 ALIGN=\"CENTER\">" + title + "</H1>");

    Item 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>");
}

```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

Listing 9.2 CatalogPage.java (continued)

```
/** POST and GET requests handled identically. */

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

Listing 9.3 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 9.4 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");
    }
}
```



Figure 9-3 Result of the KidsBooksPage servlet.

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

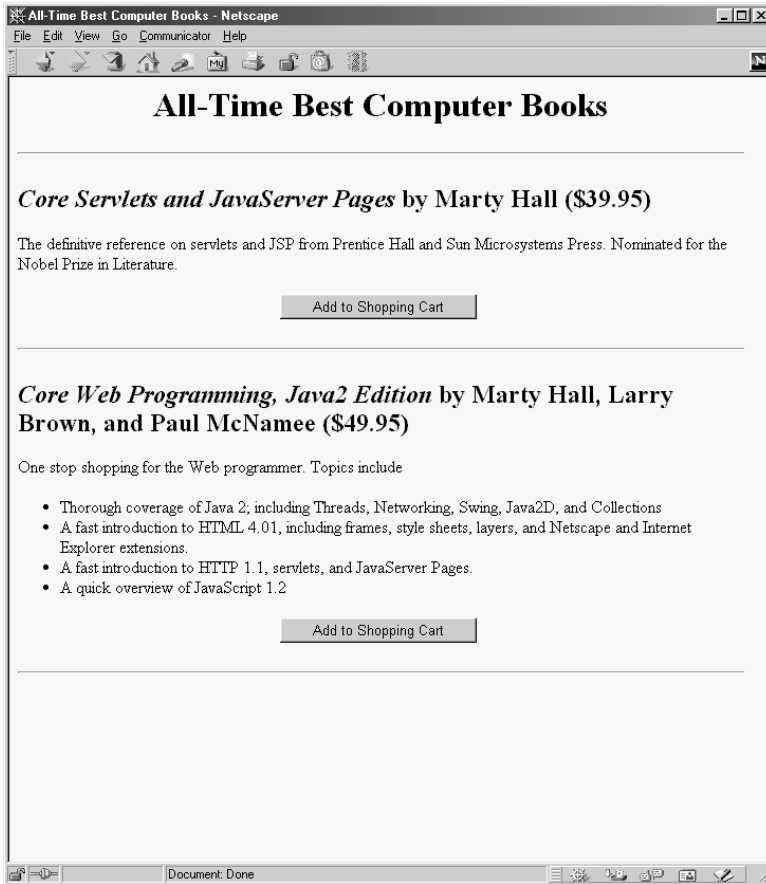


Figure 9-4 Result of the TechBooksPage servlet.

Handling the Orders

Listing 9.5 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 where concurrent access could occur, such as when a single user has multiple browser windows open and sends updates from more than one very close together in time. So, just to be safe, the code synchronizes access based upon the session object. This prevents other threads that

use the same session from accessing the data concurrently, while still allowing simultaneous requests from different users to proceed. Figures 9–5 and 9–6 show some typical results.

Listing 9.5 OrderPage.java

```
package coreservlets;

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

/** 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(true);
        ShoppingCart cart;
        synchronized(session) {
            cart = (ShoppingCart)session.getValue("shoppingCart");
            // New visitors get a fresh shopping cart.
            // Previous visitors keep using their existing cart.
            if (cart == null) {
                cart = new ShoppingCart();
                session.putValue("shoppingCart", cart);
            }
            String itemID = request.getParameter("itemID");
            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
```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

Listing 9.5 OrderPage.java (continued)

```

        // 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";
out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
synchronized(session) {
    Vector 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
            ("<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();

        String formURL =
            "/servlet/coreservlets.OrderPage";
        // Pass URLs that reference own site through encodeURL.
        formURL = response.encodeURL(formURL);

        // For each entry in shopping cart, make
        // table row showing ID, description, per-item
        // cost, number ordered, and total cost.

```

Listing 9.5 OrderPage.java (continued)

```

// 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.elementAt(i);
    out.println
    ("<TR>\n" +
     " <TD>" + order.getItemID() + "\n" +
     " <TD>" + order.getShortDescription() + "\n" +
     " <TD>" +
     formatter.format(order.getUnitCost()) + "\n" +
     " <TD>" +
     "<FORM ACTION=\"" + formURL + "\">\n" +
     "<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
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>");
}
}

/** POST and GET requests handled identically. */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

219

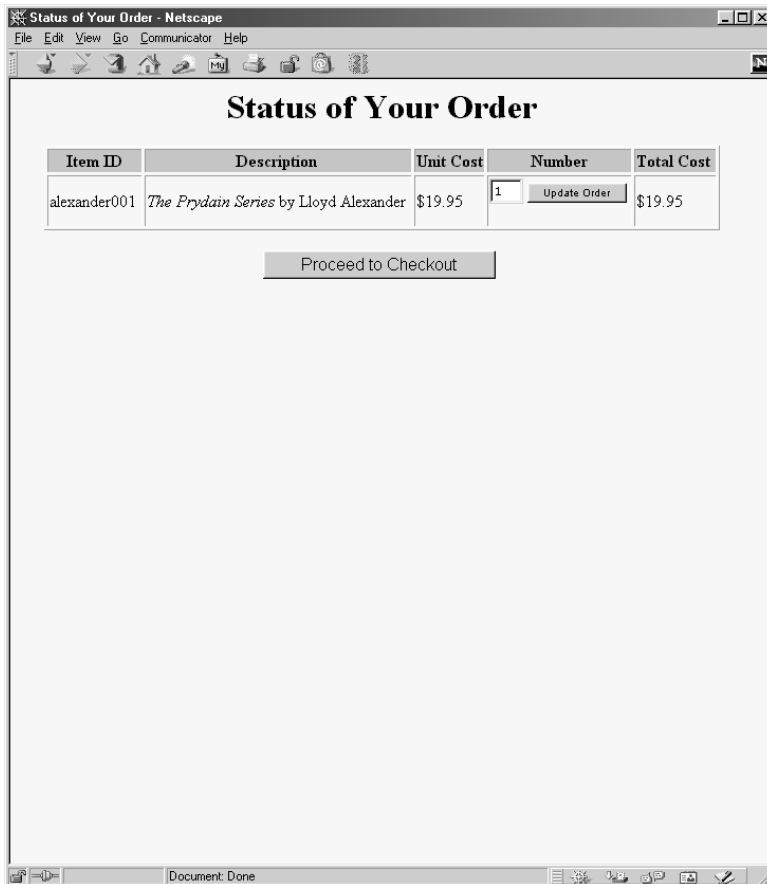


Figure 9-5 Result of OrderPage servlet after user clicks on “Add to Shopping Cart” in KidsBooksPage.



Figure 9-6 Result of `OrderPage` servlet after several additions and changes to the order.

Behind the Scenes: Implementing the Shopping Cart and Catalog Items

Listing 9.6 gives the shopping cart implementation. It simply maintains a `Vector` of orders, with methods to add and update these orders. Listing 9.7 shows the code for the individual catalog item, Listing 9.8 presents the class representing the order status of a particular item, and Listing 9.9 gives the catalog implementation.

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

Listing 9.6 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 Vector itemsOrdered;

    /** Builds an empty shopping cart. */

    public ShoppingCart() {
        itemsOrdered = new Vector();
    }

    /** Returns Vector of ItemOrder entries giving
     * Item and number ordered.
     */

    public Vector 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.elementAt(i);
            if (order.getItemID().equals(itemID)) {
                order.incrementNumItems();
                return;
            }
        }
        ItemOrder newOrder = new ItemOrder(Catalog.getItem(itemID));
        itemsOrdered.addElement(newOrder);
    }

    /** Looks through cart to find order entry corresponding
     * to item ID listed. If the designated number
```

Listing 9.6 ShoppingCart.java (continued)

```

    * 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) {
        ItemOrder order;
        for(int i=0; i<itemsOrdered.size(); i++) {
            order = (ItemOrder)itemsOrdered.elementAt(i);
            if (order.getItemID().equals(itemID)) {
                if (numOrdered <= 0) {
                    itemsOrdered.removeElementAt(i);
                } else {
                    order.setNumItems(numOrdered);
                }
            }
            return;
        }
        ItemOrder newOrder =
            new ItemOrder(Catalog.getItem(itemID));
        itemsOrdered.addElement(newOrder);
    }
}

```

Listing 9.7 Item.java

```

package coreservlets;

/** Describes a catalog item for on-line 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 Item {
    private String itemID;
    private String shortDescription;
    private String longDescription;
    private double cost;
}

```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

Listing 9.7 Item.java (continued)

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

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 9.8 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
 *  Item data without first extracting the Item separately.
 */

public class ItemOrder {
    private Item item;
    private int numItems;

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

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

    protected void setItem(Item 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;
    }
}

```

9.4 An On-Line Store Using a Shopping Cart and Session Tracking

Listing 9.8 ItemOrder.java (continued)

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

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

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

Listing 9.9 Catalog.java

```
package coreservlets;

/** A catalog listing the items available in inventory. */

public class Catalog {
    // This would come from a database in real life
    private static Item[] items =
        { new Item("hall001",
            "<I>Core Servlets and JavaServer Pages</I> " +
            "by Marty Hall",
            "The definitive reference on servlets " +
            "and JSP from Prentice Hall and \n" +
            "Sun Microsystems Press. Nominated for " +
            "the Nobel Prize in Literature.",
            39.95),
        new Item("hall002",
            "<I>Core Web Programming, Java2 Edition</I> " +
            "by Marty Hall, Larry Brown, and " +
            "Paul McNamee",
            "One stop shopping for the Web programmer. " +
            "Topics include \n" +
            "<UL><LI>Thorough coverage of Java 2; " +
            "including Threads, Networking, Swing, \n" +
            "Java2D, and Collections\n" +
            "<LI>A fast introduction to HTML 4.01, " +
            "including frames, style sheets, layers,\n" +
            "and Netscape and Internet Explorer " +
            "extensions.\n" +
            "<LI>A fast introduction to HTTP 1.1, " +
            "servlets, and JavaServer Pages.\n" +
```

Listing 9.9 Catalog.java (continued)

```

        "<LI>A quick overview of JavaScript 1.2\n" +
        "</UL>",
        49.95),
    new Item("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 Item("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 Annvin. 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 Item("rowling001",
        "<I>The Harry Potter Trilogy</I> by " +
        "J.K. Rowling",
        "The first three of the popular stories " +
        "about wizard-in-training Harry Potter\n" +
        "topped both the adult and children's " +
        "best-seller lists. Series includes\n" +
        "<I>Harry Potter and the " +
        "Sorcerer's Stone</I>,\n" +
        "<I>Harry Potter and the "

```

Listing 9.9 Catalog.java (continued)

```
                "Chamber of Secrets</I>, and\n" +
                "<I>Harry Potter and the " +
                "Prisoner of Azkaban</I>.",
                25.95)
    };

    public static Item getItem(String itemID) {
        Item 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);
    }
}
```
