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.