# Creating and Processing HTML Forms

**Topics in This Chapter**

- Data submission from forms
- Text controls
- Push buttons
- Check boxes and radio buttons
- Combo boxes and list boxes
- File upload controls
- Server-side image maps
- Hidden fields
- Groups of controls
- Tab ordering
- A Web server for debugging forms

# Chapter 19



HTML forms provide a simple and reliable user interface to collect data from the user and transmit the data to a servlet or other server-side program for processing. In this chapter we present the standard form controls defined by the HTML 4.0 specification. However, before covering each control, we first explain how the form data is transmitted to the server when a `GET` or `POST` request is made.

We also present a mini Web server that is useful for understanding and debugging the data sent by your HTML forms. The server simply reads all the HTTP data sent to it by the browser, then returns a Web page with those lines embedded within a `PRE` element. We use this server throughout the examples in this chapter to show the form control data that is sent to the server when the HTML form is submitted.

To use forms, you'll need to remember where to place regular HTML files to make them accessible to the Web server. This location varies from server to server, as discussed in Chapter 2 and the Appendix. Below, we review the location for HTML files in the default Web application for Tomcat, JRun, and Resin.

## Default Web Application: Tomcat

- **Main Location.**
  *install_dir*/webapps/ROOT
- **Corresponding URL.**
  http://*host*/*SomeFile*.html
- **More Specific Location (Arbitrary Subdirectory).**
  *install_dir*/webapps/ROOT/*SomeDirectory*

- **Corresponding URL.**
  http://*host*/*SomeDirectory*/*SomeFile*.html

## Default Web Application: JRun

- **Main Location.**
  *install_dir*/servers/default/default-ear/default-war
- **Corresponding URL.**
  http://*host*/*SomeFile*.html
- **More Specific Location (Arbitrary Subdirectory).**
  *install_dir*/servers/default/default-ear/default-war/*SomeDirectory*
- **Corresponding URL.**
  http://*host*/*SomeDirectory*/*SomeFile*.html

## Default Web Application: Resin

- **Main Location.**
  *install_dir*/doc
- **Corresponding URL.**
  http://*host*/*SomeFile*.html
- **More Specific Location (Arbitrary Subdirectory).**
  *install_dir*/doc/*SomeDirectory*
- **Corresponding URLs.**
  http://*host*/*SomeDirectory*/*SomeFile*.html

The server's default Web application is useful for practice and learning, but when you deploy real-life applications, you will almost certainly use custom Web applications; see Section 2.11 for details.

# 19.1 How HTML Forms Transmit Data

HTML forms let you create a variety of user interface controls to collect input in a Web page. Each of the controls typically has a name and a value, where the name is specified in the HTML and the value comes either from user input or from a default value in the HTML. The entire form is associated with the URL of a program that will process the data, and when the user submits the form (usually by pressing a button), the names and values of the controls are sent to the designated URL as a string of the form

```
name1=value1&name2=value2...&nameN=valueN
```

This string can be sent to the designated program in one of two ways: GET or POST. The first method, an HTTP GET request, appends the form data to the end of the specified URL after a question mark. The second method, HTTP POST, sends the data after the HTTP request headers and a blank line. In the following examples, we show explicitly how the data is sent to the server for both GET and POST requests.

For example, Listing 19.1 (HTML code) and Figure 19–1 (typical result) show a simple form with two textfields. The HTML elements that make up this form are discussed in detail in the rest of this chapter, but for now note a couple of things. First, observe that one textfield has a name of firstName and the other has a name of lastName. Second, note that the GUI controls are considered text-level (inline) elements, so you need to use explicit HTML formatting to make sure that the controls appear next to the text describing them. Finally, notice that the FORM element designates http://localhost:8088/SomeProgram as the URL to which the data will be sent.

Before submitting the form, we started a server program called EchoServer on port 8088 of our local machine. EchoServer, shown in Section 19.12, is a mini Web server used for debugging. No matter what URL is specified and what data is sent to EchoServer, it merely returns a Web page showing all the HTTP information sent by the browser. As shown in Figure 19–2, when the form is submitted with Joe in the first textfield and Hacker in the second, the browser simply requests the URL http://localhost:8088/SomeProgram?firstName=Joe&lastName=Hacker.

**Listing 19.1** GetForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>A Sample Form Using GET</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>A Sample Form Using GET</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
  First name:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
  Last name:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT"> <!-- Press this button to submit form -->
</FORM>
</CENTER>
</BODY></HTML>
```

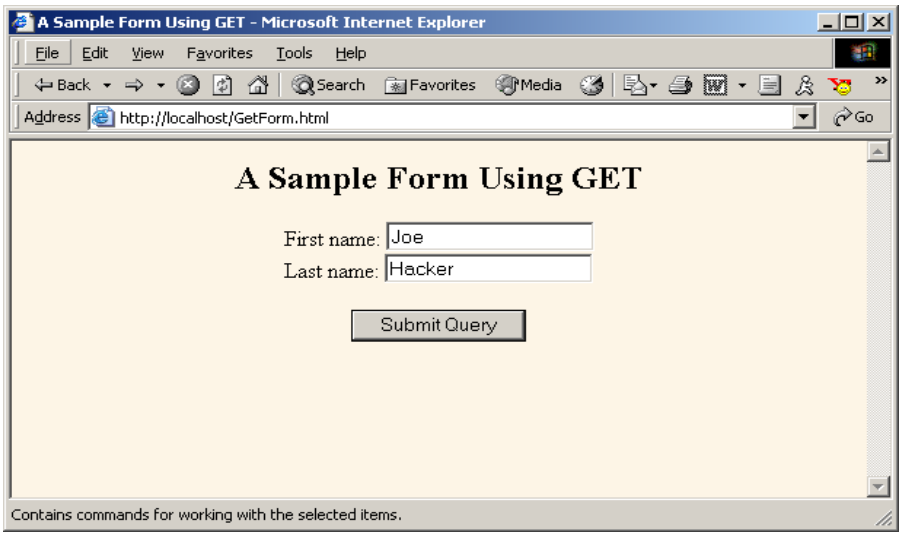


**Figure 19–1** Initial result of GetForm.html.

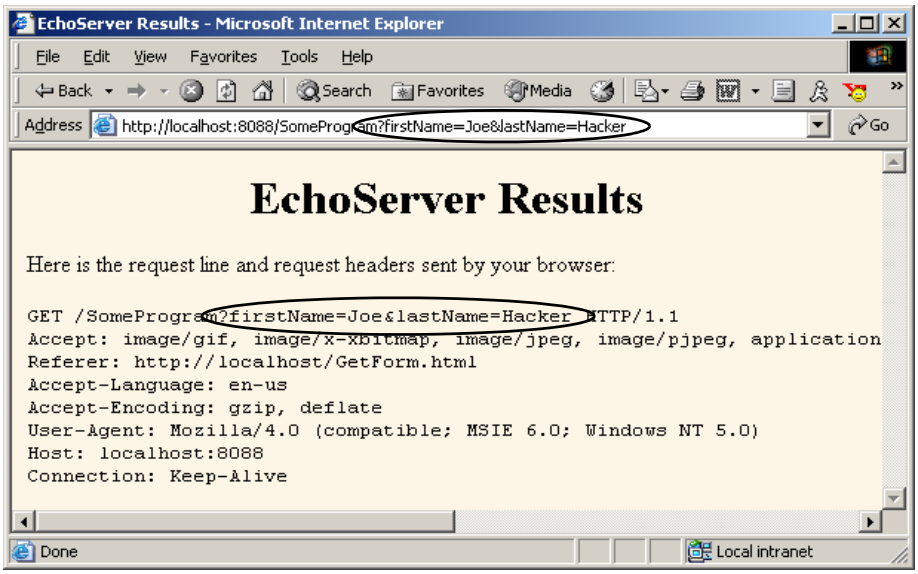


**Figure 19–2** HTTP request sent by Internet Explorer 6.0 when submitting GetForm.html.

Listing 19.2 (HTML code) and Figure 19–3 (typical result) show a variation that uses `POST` instead of `GET`. As shown in Figure 19–4, submitting the form with textfield values of `Joe` and `Hacker` results in the line `firstName=Joe&lastName=Hacker` being sent to the browser on a separate line after the HTTP request headers and a blank line.

That's the general idea behind HTML forms: GUI controls gather data from the user, each control has a name and a value, and a string containing all the name/value pairs is sent to the server when the form is submitted. Extracting the names and values on the server is straightforward in servlets: that subject is covered in Chapter 4 (Handling the Client Request: Form Data). The commonly used form controls are covered in the following sections.

**Listing 19.2** PostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>A Sample Form Using POST</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>A Sample Form Using POST</H2>
<FORM ACTION="http://localhost:8088/SomeProgram"
      METHOD="POST">
  First name:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
  Last name:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT">
</FORM>
</CENTER>
</BODY></HTML>
```

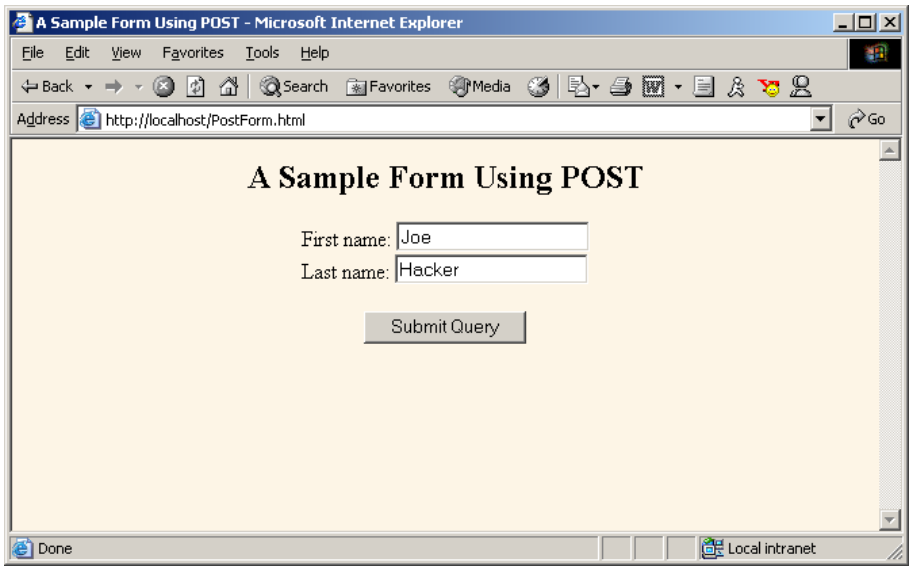


**Figure 19–3** Initial result of PostForm.html.

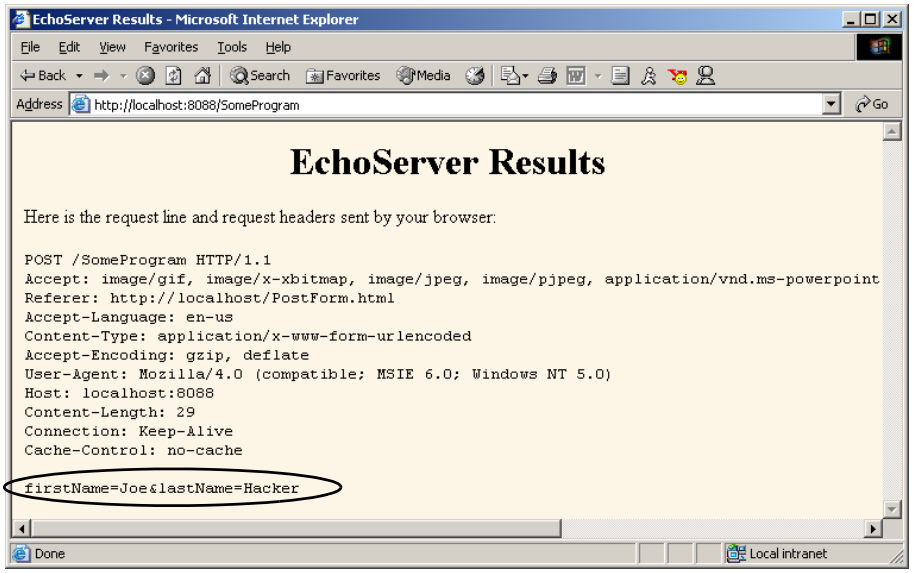


**Figure 19–4** HTTP request sent by Internet Explorer 6.0 when submitting PostForm.html.

# 19.2 The FORM Element

HTML forms allow you to create a set of data input elements associated with a particular URL. Each of these elements is typically given a name in the HTML source code, and each has a value based on the original HTML or user input. When the form is submitted, the names and values of all active elements are collected into a string with = between each name and value and with & between each name/value pair. This string is then transmitted to the URL designated by the FORM element. The string is either appended to the URL after a question mark or sent on a separate line after the HTTP request headers and a blank line, depending on whether GET or POST is used as the submission method. This section covers the FORM element itself, used primarily to designate the URL and to choose the submission method. The following sections cover the various user interface controls that can be used within forms.

### HTML Element: `<FORM ACTION="..." ...> ... </FORM>`

**Attributes:** ACTION, METHOD, ENCTYPE, TARGET, ONSUBMIT, ONRESET, ACCEPT, ACCEPT-CHARSET

The FORM element creates an area for data input elements and designates the URL to which any collected data will be transmitted. For example:

```
<FORM ACTION="http://some.isp.com/someWebApp/SomeServlet">
  FORM input elements and regular HTML
</FORM>
```

The rest of this section explains the attributes that apply to the FORM element: ACTION, METHOD, ENCTYPE, TARGET, ONSUBMIT, ONRESET, ACCEPT, and ACCEPT-CHARSET. Note that we do not discuss attributes like STYLE, CLASS, and LANG that apply to general HTML elements, but only those that are specific to the FORM element.

#### ACTION

The ACTION attribute specifies the URL of the server-side program that will process the FORM data (e.g., http://www.whitehouse.gov/servlet/schedule-fund-raiser). If the server-side program is located on the same server from which the HTML form was obtained, we recommend using a relative URL instead of an absolute URL for the action. This approach lets you move both the form and the servlet to a different host without editing either. This is an important consideration since you typically develop and test on one machine and then deploy on another. For example,

```
ACTION="/servlet/schedule-fund-raiser"
```

### Core Approach

*If the servlet or JSP page is located on the same server as the HTML form, use a relative URL in the* `ACTION` *attribute.*

In addition, you can specify an email address to which the `FORM` data will be sent (e.g., mailto:audit@irs.gov). Some ISPs do not allow ordinary users to create server-side programs, or they charge extra for this privilege. In such a case, sending the data by email is a convenient option when you create pages that need to collect data but not return results (e.g., for accepting orders for products). You must use the `POST` method (see `METHOD` in the following subsection) when using a mailto URL.

Also, note that the `ACTION` attribute is not required for the `FORM` element. If you omit `ACTION`, the form data is sent to the same URL as the form itself. See Section 4.8 for an example of the use of this self-submission approach.

#### METHOD

The `METHOD` attribute specifies how the data will be transmitted to the HTTP server. When `GET` is used, the data is appended to the end of the designated URL after a question mark. For an example, see Section 19.1 (How HTML Forms Transmit Data). `GET` is the default and is also the method that is used when the user types a URL into the address bar or clicks on a hypertext link. When `POST` is used, the data is sent on a separate line. Either `GET` or `POST` could be preferable, depending on the situation.

Since `GET` data is part of the URL, the advantages of `GET` are that you can do the following:

- **Save the results of a form submission.** For example, you can submit data and bookmark the resultant URL, send it to a colleague by email, or put it in a normal hypertext link. The ability to bookmark the results page is the main reason google.com, yahoo.com, and other search engines use `GET`.
- **Type data in by hand.** You can test servlets or JSP pages that use `GET` simply by entering a URL with the appropriate data attached. This ability is convenient during initial development.

Since `POST` data is *not* part of the URL, the advantages of `POST` are that you can do the following:

- **Transmit large amounts of data.** Many browsers limit URLs to a few thousand characters, making `GET` inappropriate when your form must send a large amount of data. Since HTML forms let you upload files from the client machine (see Section 19.7), sending multiple megabytes of data is quite commonplace. Only `POST` can be used for this task.
- **Send binary data.** Spaces, carriage returns, tabs, and many other characters are illegal in URLs. If you upload a large binary file, it would be a time-consuming process to encode all the characters before transmission and decode them on the other end.
- **Keep the data private from someone looking over the user's shoulder.** HTML forms let you create password fields in which the data is replaced by asterisks on the screen. However, using a password field is pointless if the data is displayed in clear text in the URL, letting snoopers read it by peering over the user's shoulder or by scrolling through the browser's history list when the user leaves the computer unattended. Note, however, that `POST` alone provides no protection from someone using a packet sniffer on the network connection. To protect against this type of attack, use SSL (`https://` connections) to encrypt the network traffic. For more information on using SSL in Web applications, see the chapters on Web application security in Volume 2 of this book.

To read `GET` or `POST` data from a servlet, you call

```
request.getParameter("name")
```

where `name` is the value of the `NAME` attribute of the input element in the HTML form. For additional details, see Chapter 4 (Handling the Client Request: Form Data). Note that, if needed, you can also use `request.getInputStream` to read the `POST` data directly, as below.

```
int length = request.getContentLength();
if (length > SOME_MAXSIZE) {
  throw new IOException("Possible denial of service attack");
}
byte[] data = new byte[length];
ServletInputStream inputStream = request.getInputStream();
int read = inputStream.readLine(data, 0, length);
```

### ENCTYPE

This attribute specifies the way in which the data will be encoded before being transmitted. The default is `application/x-www-form-urlencoded`. The encoding, as specified by the World Wide Web Consortium, is UTF-8, except that the client converts each space into a plus sign (+) and each other non-

alphanumeric character into a percent sign (%) followed by the two hexadecimal digits representing that character in the browser's character set. These transformations are in addition to placing an equal sign (=) between entry names and values and an ampersand (&) between the pairs.

For example, Figure 19–5 shows a version of GetForm.html (Listing 19.1) where "`Larry (Java Hacker?)`" is entered for the first name. As can be seen in Figure 19–6, this entry is sent as "`Larry+%28Java+Hacker%3F%29`". That's because spaces become plus signs, 28 is the ASCII value (in hex) for a left parenthesis, 3F is the ASCII value of a question mark, and 29 is a right parenthesis.

Note that, unless otherwise specified, `POST` data is also encoded as `application/x-www-form-urlencoded`.

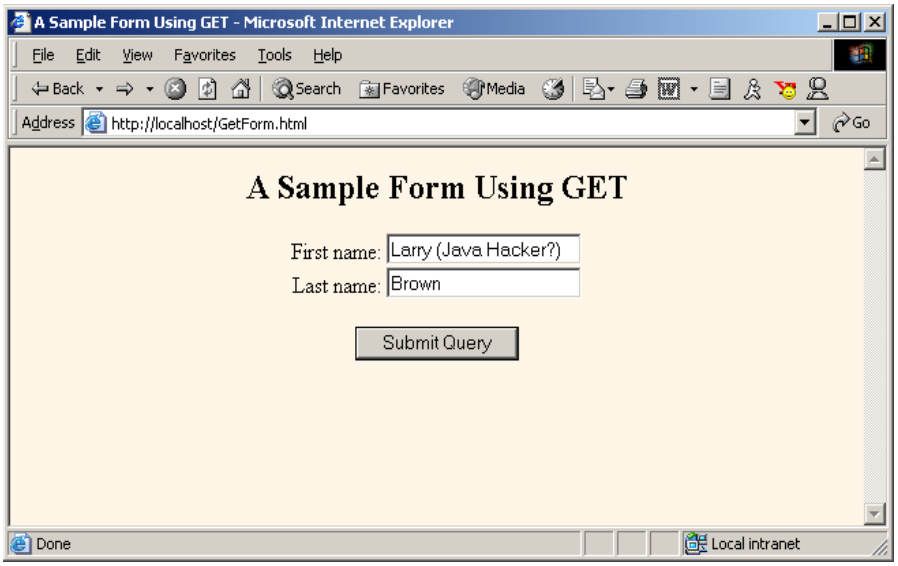


**Figure 19–5** Customized result of GetForm.html.

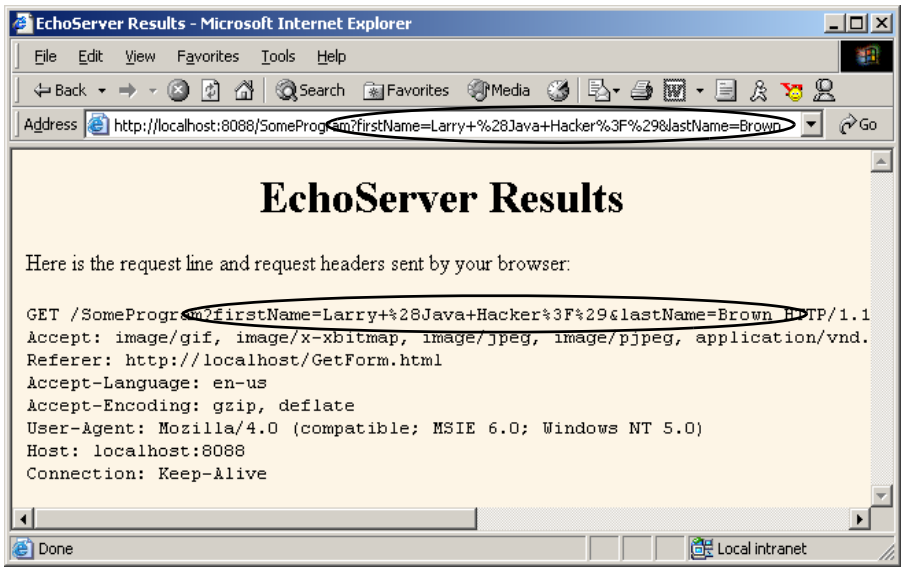


**Figure 19–6** HTTP request sent by Internet Explorer 6.0 when submitting GetForm.html with the data shown in Figure 19–5.

Most recent browsers support an additional `ENCTYPE` of `multipart/form-data`. This encoding transmits each field as a separate part of a MIME-compatible document. To use this `ENCTYPE`, you must specify `POST` as the method type. This encoding sometimes makes it easier for the server-side program to handle complex data, and it is required when you are using file upload controls to send entire documents (see Section 19.7). For example, Listing 19.3 shows a form that differs from GetForm.html (Listing 19.1) only in that

```
<FORM ACTION="http://localhost:8088/SomeProgram">
```

has been changed to

```
<FORM ACTION="http://localhost:8088/SomeProgram"
      ENCTYPE="multipart/form-data" METHOD="POST">
```

Figures 19–7 and 19–8 show the results.

**Listing 19.3** MultipartForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Using ENCTYPE="multipart/form-data"</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>Using ENCTYPE="multipart/form-data"</H2>
<FORM ACTION="http://localhost:8088/SomeProgram"
      ENCTYPE="multipart/form-data" METHOD="POST">
  First name:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
  Last name:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT">
</FORM>
</CENTER>
</BODY></HTML>
```

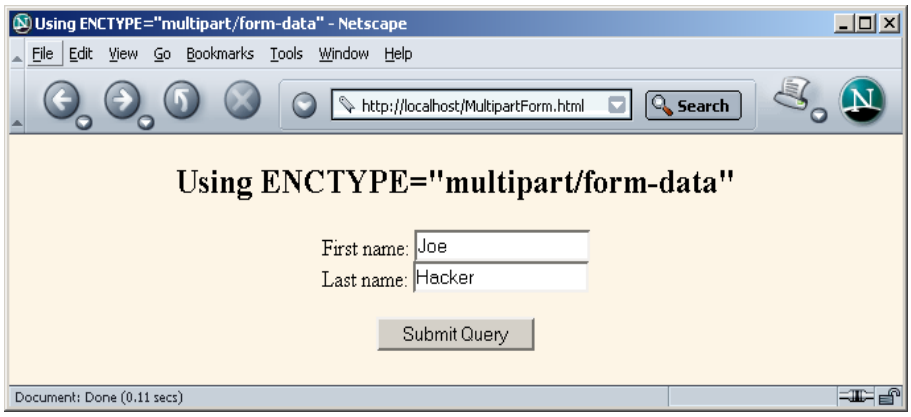


**Figure 19–7** Initial result of MultipartForm.html.

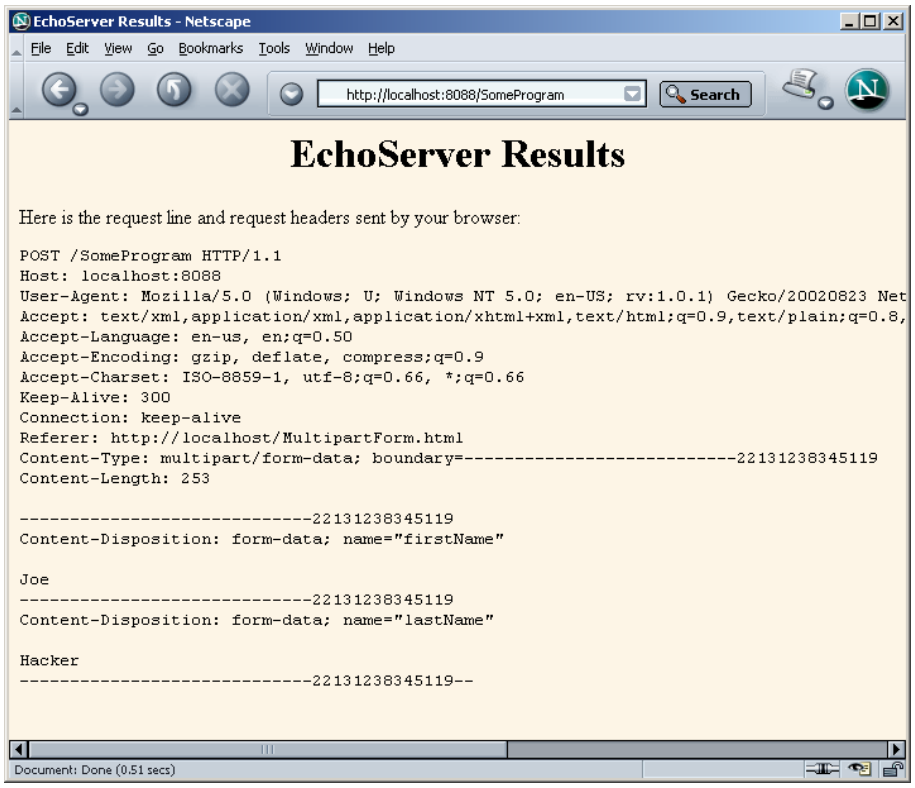


**Figure 19–8** HTTP request sent by Netscape 7.0 when submitting MultipartForm.html.

### TARGET

The `TARGET` attribute is used by frame-capable browsers to determine which frame cell should be used to display the results of the servlet, JSP page, or other program handling the form submission. The default is to display the results in whatever frame cell contains the form being submitted.

### ONSUBMIT and ONRESET

These attributes are used by JavaScript to attach code that should be evaluated when the form is submitted or reset. For `ONSUBMIT`, if the expression evaluates to `false`, the form is not submitted. This case lets you invoke JavaScript code on the client that checks the format of the form field values before they are submitted, prompting the user for missing or illegal entries.

### ACCEPT and ACCEPT-CHARSET

These attributes are new in HTML 4.0 and specify the MIME types (`ACCEPT`) and character encodings (`ACCEPT-CHARSET`) that must be accepted by the servlet or other program processing the form data. The MIME types listed in `ACCEPT` can also be used by the client to limit which file types are displayed to the user for file upload elements.

## 19.3 Text Controls

HTML supports three types of text-input elements: textfields, password fields, and text areas. Each is given a name, and the value is taken from the content of the control. The name and value are sent to the server when the form is submitted, which is typically done by means of a submit button (see Section 19.4).

### Textfields

**HTML Element:** `<INPUT TYPE="TEXT" NAME="..." ...>`
**(No End Tag)**

**Attributes:** `NAME` (required), `VALUE`, `SIZE`, `MAXLENGTH`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR`, `ONKEYDOWN`, `ONKEYPRESS`, `ONKEYUP`

This element creates a single-line input field in which the user can enter text, as illustrated earlier in Listings 19.1, 19.2, and 19.3. For multiline fields, see `TEXTAREA` in the following subsection. `TEXT` is the default `TYPE` in `INPUT` forms, although it is recommended for clarity that `TEXT` be supplied explicitly. You should remember that the normal browser word-wrapping applies inside `FORM` elements, so use appropriate HTML markup to ensure that the browser will not separate the descriptive text from the associated textfield.

**Core Approach**

*Use explicit HTML constructs to group textfields with their descriptive text.*

Netscape 7.0 and Internet Explorer 6.0 submit the form when the user presses Enter while the cursor is in a textfield and the form has a SUBMIT button (see Section 19.4 for details on a SUBMIT button). However, this behavior is not dictated by the HTML specification, and other browsers behave differently.

### Core Warning

*Don't rely on the browser submitting the form when the user presses Enter while in a textfield. Always include a button or image map that submits the form explicitly.*

To prevent the form from being submitted when the user presses Enter in a textfield, use a BUTTON input with an onClick event handler instead of a SUBMIT button. For example, you may want to use

```
<INPUT TYPE="BUTTON" VALUE="Check Values"
       onClick="submit()">
```

instead of

```
<INPUT TYPE="SUBMIT">
```

to submit your form.

The following subsections describe the attributes that apply specifically to textfields. Attributes that apply to general HTML elements (e.g., STYLE, CLASS, ID) are not discussed. The TABINDEX attribute, which applies to *all* form elements, is discussed in Section 19.11 (Tab Order Control).

#### NAME

The NAME attribute identifies the textfield when the form is submitted. In standard HTML, the attribute is required. Because data is always sent to the server in the form of name/value pairs, no data is sent for form controls that have no NAME.

#### VALUE

A VALUE attribute, if supplied, specifies the *initial* contents of the textfield. When the form is submitted, the *current* contents are sent; these can reflect user input. If the textfield is empty when the form is submitted, the form data simply consists of the name and an equal sign (e.g., name1=value1&**textfieldname=**&name3=value3).

### SIZE

This attribute specifies the width of the textfield, based on the average character width of the font being used. If text beyond this size is entered, the textfield scrolls to accommodate it. This could happen if the user enters more characters than `SIZE` number or enters `SIZE` number of wide characters (e.g., capital W) when a proportional-width font is being used. Netscape and Internet Explorer 6.0 automatically use a proportional font in textfields. Unfortunately, you cannot change the font by embedding the `INPUT` element in a `FONT` or `CODE` element. However, you can use cascading style sheets to change the font of input elements. For example, the following style sheet (placed in the `HEAD` section of the HTML page) will display the text for all `INPUT` elements as 12pt Futura (assuming that the Futura font is installed on the client machine).

```
<style type="text/css">
INPUT {
 font-size : 12pt;
 font-family : Futura;
}
</style>
```

**Core Approach**

*By default, Netscape and Internet Explorer display `INPUT` elements in a proportional font. To change the font for `INPUT` elements, use style sheets.*

### MAXLENGTH

`MAXLENGTH` gives the maximum number of *allowable* characters. This number is in contrast to the number of *visible* characters, which is specified through `SIZE`. However, note that users can always override this; for `GET` requests, they can type data directly in the URL and for `POST` requests they can write their own HTML form. So, the server-side program should not rely on the request containing the appropriate amount of data.

### ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONDBLDOWN, ONKEYPRESS, and ONKEYUP

These attributes are used only by browsers that support JavaScript. They specify the action to take when the mouse leaves the textfield after a change has occurred, when the user selects text in the textfield, when the textfield gets the input focus, when the textfield loses the input focus, and when individual keys are pressed or released, respectively.

## Password Fields

**HTML Element:** `<INPUT TYPE="PASSWORD" NAME="..." ...>`
**(No End Tag)**

**Attributes:** `NAME` (required), `VALUE`, `SIZE`, `MAXLENGTH`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR`, `ONKEYDOWN`, `ONKEYPRESS`, `ONKEYUP`

Password fields are created and used just like textfields, except that when the user enters text, the input is not echoed; instead, some obscuring character, usually an asterisk, is displayed (see Figure 19–9). Obscured input is useful for collecting data such as credit card numbers or passwords that the user would not want shown to people who may be near his computer. The regular, unobscured text (clear text) is transmitted as the value of the field when the form is submitted.

Since `GET` data is appended to the URL after a question mark, you should always use `POST` with a password field so that a bystander cannot read the unobscured password from the URL display at the top of the browser. In addition, for security during transmission of data, you should consider using SSL, which encrypts the data. For more information on using SSL, see the chapters on Web application security in Volume 2 of this book.

### Core Approach

*To protect the user's privacy, always use* `POST` *when creating forms that contain password fields. For additional security, transmit the data by using* `https://`*, which uses SSL to encrypt the data.*

### NAME, VALUE, SIZE, MAXLENGTH, ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, and ONKEYUP

Attributes for password fields are used in exactly the same manner as with textfields.

Enter Password: ••••••••

**Figure 19–9** A password field created by means of `<INPUT TYPE="PASSWORD" ...>`.

## Text Areas

**HTML Element:** `<TEXTAREA NAME="..." ROWS=xxx COLS=yyy> ... </TEXTAREA>`

**Attributes:** `NAME` (required), `ROWS` (required), `COLS` (required), `WRAP` (nonstandard), `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR`, `ONKEYDOWN`, `ONKEYPRESS`, `ONKEYUP`

The `TEXTAREA` element creates a multiline text area; see Figure 19–10. The element has no `VALUE` attribute; instead, text between the start and end tags is used as the initial content of the text area. The initial text between `<TEXTAREA ...>` and `</TEXTAREA>` is treated similarly to text inside the now-obsolete `XMP` element. That is, white space in this initial text is maintained, and HTML markup between the start and end tags is taken literally, except for character entities such as `&lt;`, `&copy;`, and so forth, which are interpreted normally. Unless a custom `ENCTYPE` is used in the form (see Section 19.2, "The FORM Element"), characters, including those generated from character entities, are URL-encoded before being transmitted. That is, spaces become plus signs and other nonalphanumeric characters become `%`*XX*, where *XX* is the numeric value of the character in hex.

### NAME

This attribute specifies the name that will be sent to the server.

### ROWS

`ROWS` specifies the number of visible lines of text. If more lines of text are entered, a vertical scrollbar will be added to the text area.

### COLS

`COLS` specifies the visible width of the text area, based on the average width of characters in the font being used. In Netscape 7.0 and Internet Explorer 6.0, if the text on a single line contains more characters than the specified width allows, the text is wrapped to the next line. However, if a single word has more characters than the specified width, Internet Explorer 6.0 wraps the word to the next line and Netscape 7.0 adds a horizontal scrollbar to keep the word on one line. Other browsers may behave differently.

### ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, and ONKEYUP

These attributes apply only to browsers that support JavaScript; they specify code to be executed when certain conditions arise. `ONCHANGE` handles the

situation in which the input focus leaves the text area after it has changed, ONSELECT describes what to do when text in the text area is selected by the user, ONFOCUS and ONBLUR specify what to do when the text area acquires or loses the input focus, and the remaining attributes determine what to do when individual keys are typed.

Listing 19.4 creates a text area with 5 visible rows that can hold about 30 characters per row. The result is shown in Figure 19–10.

**Listing 19.4** Example of a TEXTAREA form control

```
<CENTER>
<P>
Enter some HTML:<BR>
<TEXTAREA NAME="HTML" ROWS=5 COLS=30>
Delete this text and replace
with some HTML to validate.
</TEXTAREA>
<CENTER>
```

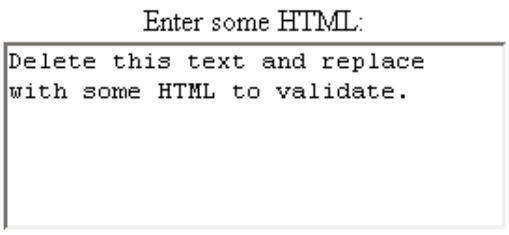


**Figure 19–10** A text area in Netscape 7.0.

# 19.4 Push Buttons

Push buttons are used for two main purposes in HTML forms: to submit forms and to reset the controls to the values specified in the original HTML. Browsers that use JavaScript can also use buttons for a third purpose: to trigger arbitrary JavaScript code.

Traditionally, buttons have been created by the INPUT element used with a TYPE attribute of SUBMIT, RESET, or BUTTON. In HTML 4.0, the BUTTON element was

introduced and is supported by Internet Explorer 6.0 and Netscape 7.0. This new element lets you create buttons with multiline labels, images, font changes, and the like. However, earlier browsers may not support the `BUTTON` element.

## Submit Buttons

### HTML Element: `<INPUT TYPE="SUBMIT" ...>` (No End Tag)

**Attributes:** `NAME`, `VALUE`, `ONCLICK`, `ONDBLCLICK`, `ONFOCUS`, `ONBLUR`

When a submit button is clicked, the form is sent to the servlet or other server-side program designated by the `ACTION` parameter of the `FORM`. Although the action can be triggered in other ways, such as the user clicking on an image map, most forms have at least one submit button. Submit buttons, like other form controls, adopt the look and feel of the client operating system, so will look slightly different on different platforms. Figure 19–11 shows a submit button on Windows 2000 Professional, created by

```
<INPUT TYPE="SUBMIT">
```


**Figure 19–11** A submit button with the default label.

#### NAME and VALUE

Most input elements have a name and an associated value. When the form is submitted, the names and values of active elements are concatenated to form the data string. If a submit button is used simply to initiate the submission of the form, the button's name can be omitted and it does not contribute to the data string that is sent. If a name *is* supplied, then only the name and value of the button that was actually clicked are sent. This capability lets you use more than one button and detect which one is pressed. The label is used as the value that is transmitted. Supplying an explicit `VALUE` will change the default label.

For instance, Listing 19.5 creates a textfield and two submit buttons, shown in Figure 19–12. If, for example, the first button is selected, the data string sent to the server would be `Item=256MB+SIMM&`**`Add`**`=Add+Item+to+Cart`.

**Listing 19.5** Example of SUBMIT input controls

```
<CENTER>
Item:
<INPUT TYPE="TEXT" NAME="Item" VALUE="256MB SIMM"><BR>
<INPUT TYPE="SUBMIT" NAME="Add"
       VALUE="Add Item to Cart">
<INPUT TYPE="SUBMIT" NAME="Delete"
       VALUE="Delete Item from Cart">
</CENTER>
```

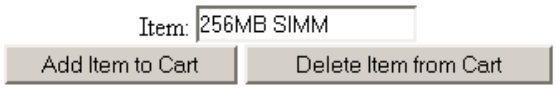


**Figure 19–12** Submit buttons with user-defined labels.

Note that when the form data is submitted to a servlet, `request.getParameter` returns `null` for buttons that were not pressed. So, you could use a simple check for `null`, as below, to determine which button was selected.

```
if (request.getParameter("Add") != null) {
  doCartAdditionOperation(...);
} else if (request.getParameter("Delete") != null) {
  doCartDeletionOperation(...);
}
```

### ONCLICK, ONDBLCLICK, ONFOCUS, and ONBLUR

These nonstandard attributes are used by JavaScript-capable browsers to associate JavaScript code with the button. The `ONCLICK` and `ONDBLCLICK` code is executed when the button is pressed, the `ONFOCUS` code when the button gets the input focus, and the `ONBLUR` code when the button loses the focus. If the code attached to a button returns `false`, the submission of the form is suppressed. HTML attributes are not case sensitive, and these attributes are traditionally called `onClick`, `onDblClick`, `onFocus`, and `onBlur` by JavaScript programmers.

### HTML Element: `<BUTTON TYPE="SUBMIT" ...>` `HTML Markup` `</BUTTON>`

**Attributes:** NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

This alternative way of creating submit buttons lets you use arbitrary HTML markup for the content of the button. This element lets you have multiline button labels, button labels with font changes, image buttons, and so forth. Listing 19.6 gives a few examples, with results shown in Figure 19–13.

#### NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, and ONBLUR

These attributes are used in the same way as with `<INPUT TYPE="SUBMIT" ...>`.

**Listing 19.6** ButtonElement.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>The BUTTON Element</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE">
<CENTER>
<H2>The BUTTON Element</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
<BUTTON TYPE="SUBMIT">Single-line Label</BUTTON>
  
<BUTTON TYPE="SUBMIT">Multi-line<BR>label</BUTTON>
<P>
<BUTTON TYPE="SUBMIT">
<B>Label</B> with <I>font</I> changes.
</BUTTON>
<P>
<BUTTON TYPE="SUBMIT">
<IMG SRC="images/Java-Logo.gif" WIDTH="110" HEIGHT="101"
     ALIGN="LEFT" ALT="Java Cup Logo">
Label<BR>with image
</BUTTON>
</FORM>
</CENTER>
</BODY></HTML>
```

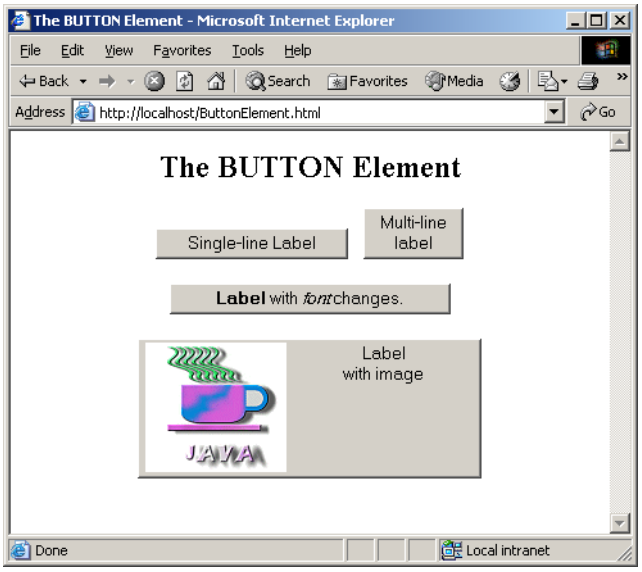


**Figure 19–13** Submit buttons created with the `BUTTON` element.

## Reset Buttons

**HTML Element:** `<INPUT TYPE="RESET" ...>` **(No End Tag)**
**Attributes:** `VALUE`, `NAME`, `ONCLICK`, `ONDBLCLICK`, `ONFOCUS`, `ONBLUR`

Reset buttons serve to reset the values of all items in the form to those specified in the original `VALUE` parameters. Their value is never transmitted as part of the form's contents.

### VALUE

The `VALUE` attribute specifies the button label; "Reset" is the default.

### NAME

Because reset buttons do not contribute to the data string transmitted when the form is submitted, they are not named in standard HTML. However, JavaScript permits a `NAME` attribute to be used to simplify reference to the element.

### ONCLICK, ONDBLCLICK, ONFOCUS, and ONBLUR

These nonstandard attributes are used by JavaScript-capable browsers to associate JavaScript code with the button. The `ONCLICK` and `ONDBLCLICK` code is executed when the button is pressed, the `ONFOCUS` code when the button gets

the input focus, and the ONBLUR code when it loses the focus. HTML attributes are not case sensitive, and these attributes are traditionally called onClick, onDblClick, onFocus, and onBlur by JavaScript programmers.

**HTML Element:** `<BUTTON TYPE="RESET" ...>` `HTML Markup` `</BUTTON>`

**Attributes:** VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

This alternative way of creating reset buttons lets you use arbitrary HTML markup for the content of the button. All attributes are used identically to those in <INPUT TYPE="RESET" ...>.

## JavaScript Buttons

**HTML Element:** `<INPUT TYPE="BUTTON" ...>` **(No End Tag)**

**Attributes:** NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

This element is recognized only by browsers that support JavaScript. It creates a button with the same visual appearance as a SUBMIT or RESET button and allows the author to attach JavaScript code to the ONCLICK, ONDBLCLICK, ONFOCUS, or ONBLUR attributes. The name/value pair associated with a JavaScript button is not transmitted as part of the data when the form is submitted. Arbitrary code can be associated with the button, but one of the most common uses is to verify that all input elements are in the proper format before the form is submitted to the server. For instance, the following creates a button that, when activated, calls the validateForm function.

```
<INPUT TYPE="BUTTON" VALUE="Check Values"
       onClick="validateForm()">
```

**HTML Element:** `<BUTTON TYPE="BUTTON" ...>` `HTML Markup` `</BUTTON>`

**Attributes:** NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

This alternative way of creating JavaScript buttons lets you use arbitrary HTML markup for the content of the button. All attributes are used identically to those in <INPUT TYPE="BUTTON" ...>.

# 19.5 Check Boxes and Radio Buttons

Check boxes and radio buttons are useful controls for allowing the user to select among a set of predefined choices. Although check boxes can be selected or deselected individually, radio buttons can be grouped so that only a single member of the group can be selected at a time.

## Check Boxes

**HTML Element:** `<INPUT TYPE="CHECKBOX" NAME="..." ...>` **(No End Tag)**

**Attributes:** `NAME` (required), `VALUE`, `CHECKED`, `ONCLICK`, `ONFOCUS`, `ONBLUR`

This input element creates a check box whose name/value pair is transmitted *only* if the check box is checked when the form is submitted. For instance, the following code results in the check box shown in Figure 19–14.

```
<P>
<INPUT TYPE="CHECKBOX" NAME="noEmail" CHECKED>
Check here if you do <I>not</I> want to
get our email newsletter
```

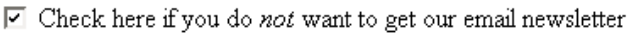


**Figure 19–14** An HTML check box.

Note that the descriptive text associated with the check box is normal HTML, and care should be taken to guarantee that it appears next to the check box. Thus, the `<P>` in the preceding example ensures that the check box isn't part of the previous paragraph.

### Core Approach

*Paragraphs inside a `FORM` are filled and wrapped just like regular paragraphs. So, be sure to insert explicit HTML markup to keep input elements with the text that describes them.*

### NAME

This attribute supplies the name that is sent to the server. The `NAME` attribute is required for standard HTML check boxes but is optional when used with JavaScript.

### VALUE

The `VALUE` attribute is optional and defaults to `on`. Recall that the name and value are sent to the server only if the check box is checked when the form is submitted. For instance, in the preceding example, `noEmail=on` would be added to the data string since the box is checked, but nothing would be added if the box was unchecked. As a result, servlets, JSP pages, or other server-side programs often check only for the existence of the check box name (e.g., that `request.getParameter` returns non-`null`), ignoring its value.

### CHECKED

If the `CHECKED` attribute is supplied, then the check box is initially checked when the associated Web page is loaded. Otherwise, it is initially unchecked.

### ONCLICK, ONFOCUS, and ONBLUR

These attributes supply JavaScript code to be executed when the button is clicked, receives the input focus, and loses the focus, respectively.

## Radio Buttons

**HTML Element:** `<INPUT TYPE="RADIO" NAME="..." VALUE="..." ...>` **(No End Tag)**

**Attributes:** `NAME` (required), `VALUE` (required), `CHECKED`, `ONCLICK`, `ONFOCUS`, `ONBLUR`

Radio buttons differ from check boxes in that only a single radio button in a given group can be selected at any one time. You indicate a group of radio buttons by providing all of them with the same `NAME`. Only one button in a group can be depressed at a time; selecting a new button when one is already selected results in the previous choice becoming deselected. The value of the one selected is sent when the form is submitted. Although radio buttons technically need not appear near to each other, this proximity is almost always recommended.

An example of a radio button is shown in Listing 19.7. Because input elements are wrapped as part of normal paragraphs, a `DL` list is used to make sure that the buttons appear under each other in the resultant page and are indented from the heading above them. Figure 19–15 shows the result. In this case, `creditCard=java` would get sent as part of the form data when the form is submitted.

**Listing 19.7** Example of a radio button group

```
<DL>
  <DT>Credit Card:
  <DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="visa">
      Visa
  <DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="mastercard">
      Master Card
  <DD><INPUT TYPE="RADIO" NAME="creditCard"
              VALUE="java" CHECKED>
      Java Smart Card
  <DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="amex">
      American Express
  <DD><INPUT TYPE="RADIO" NAME="creditCard" VALUE="discover">
      Discover
</DL>
```

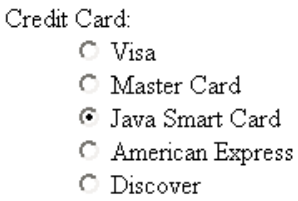


**Figure 19–15** Radio buttons in HTML.

### NAME

Unlike the `NAME` attribute of most input elements, this `NAME` attribute is shared by multiple elements. All radio buttons associated with the same name are grouped logically so that no more than one can be selected at any given time. Note that attribute values are case sensitive, so the following would result in two radio buttons that are *not* in the same group.

```
<INPUT TYPE="RADIO" NAME="Foo" VALUE="Value1">
<INPUT TYPE="RADIO" NAME="FOO" VALUE="Value2">
```

### Core Warning

*Be sure the `NAME` attribute of each radio button in a logical group matches that of the other group members exactly, including case.*

### VALUE

The VALUE attribute supplies the value that gets transmitted with NAME when the form is submitted. It doesn't affect the appearance of the radio button. Instead, normal text and HTML markup are placed around the radio button, just as with check boxes.

### CHECKED

If the CHECKED attribute is supplied, then the radio button is initially checked when the associated Web page is loaded. Otherwise, it is initially unchecked.

### ONCLICK, ONFOCUS, and ONBLUR

These attributes specify JavaScript code to be executed when the button is clicked, receives the input focus, and loses the focus, respectively.

## 19.6 Combo Boxes and List Boxes

A SELECT element presents a set of options to the user. If only a single entry can be selected and no visible size has been specified, the options are presented in a combo box (drop-down menu); list boxes are used when multiple selections are permitted or a specific visible size has been specified. The choices themselves are specified by OPTION entries embedded in the SELECT element. The typical format is as follows:

```
<SELECT NAME="Name" ...>
  <OPTION VALUE="Value1">Choice 1 Text
  <OPTION VALUE="Value2">Choice 2 Text
  ...
  <OPTION VALUE="ValueN">Choice N Text
</SELECT>
```

The HTML 4.0 specification also defines OPTGROUP (with a single attribute of LABEL) to enclose OPTION elements to create cascading menus.

**HTML Element:** `<SELECT NAME="..." ...> ... </SELECT>`
**Attributes:** NAME (required), SIZE, MULTIPLE, ONCLICK, ONFOCUS, ONBLUR, ONCHANGE

SELECT creates a combo box or list box for selecting among choices. You specify each choice with an OPTION element enclosed between <SELECT ...> and </SELECT>.

### NAME
NAME identifies the form to the servlet, JSP page, or other server-side program.

### SIZE
SIZE gives the number of visible rows. If SIZE is used, the SELECT menu is usually represented as a list box instead of a combo box. A combo box is the normal representation when neither SIZE nor MULTIPLE is supplied.

### MULTIPLE
The MULTIPLE attribute specifies that multiple entries can be selected simultaneously. If MULTIPLE is omitted, only a single selection is permitted. From a servlet, you would use request.getParameterValues to obtain an array of the entries selected in the list. For example, the code

```
String[] listValues = request.getParameterValues("language");
if (listValues != null) {
  for(int i=0; i<listValues.length; i++) {
    String value = listValues[i];
    ...
  }
}
```

would allow you to process all values selected in a list named language (see Listing 19.8). Be aware that the order of the values in the returned array may not correspond to the order of the values displayed in the list.

#### Core Approach

*If multiple selections are possible in a SELECT list, then use request.getParameterValues to obtain an array of all selected items.*

### ONCLICK, ONFOCUS, ONBLUR, and ONCHANGE
These nonstandard attributes are supported by browsers that understand JavaScript. They indicate code to be executed when the entry is clicked, gains the input focus, loses the input focus, and loses the focus after having been changed, respectively.

## HTML Element: <OPTION ...> (End Tag Optional)
**Attributes:** SELECTED, VALUE

This element specifies the menu choices; it is valid only inside a SELECT element.

### VALUE

VALUE gives the value to be transmitted with the NAME of the SELECT menu if the current option is selected. This is *not* the text that is displayed to the user; that is specified by separate HTML markup listed after the OPTION tag.

### SELECTED

If present, SELECTED specifies that the particular menu item shown is selected when the page is first loaded.

Listing 19.8 creates a menu of programming language choices. Because only a single selection is allowed and no visible SIZE is specified, it is displayed as a combo box. Figures 19–16 and 19–17 show the initial appearance and the appearance after the user activates the menu by clicking on it. If the entry Java is active when the form is submitted, then language=java is sent to the server-side program. Notice that it is the VALUE attribute, not the descriptive text, that is transmitted.

**Listing 19.8** Example of a SELECT menu

```
Favorite language:
<SELECT NAME="language">
  <OPTION VALUE="c">C
  <OPTION VALUE="c++">C++
  <OPTION VALUE="java" SELECTED>Java
  <OPTION VALUE="lisp">Lisp
  <OPTION VALUE="perl">Perl
  <OPTION VALUE="smalltalk">Smalltalk
</SELECT>
```

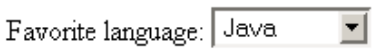


**Figure 19–16** A SELECT element displayed as a combo box (drop-down menu).

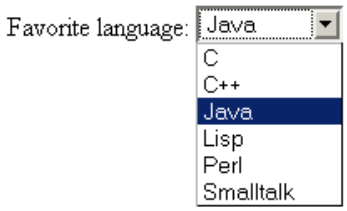


**Figure 19–17** Choosing options from a SELECT menu.

The second example shows a SELECT element rendered as a list box. If more than one entry is active when the form is submitted, then more than one value is sent, listed as separate entries (repeating NAME). For instance, in the example shown in Listing 19.9 (Figure 19–18), language=java&language=perl gets added to the data being sent to the server. Multiple entries that share the same name is the reason servlet authors need to be familiar with the getParameterValues method of HttpServletRequest in addition to the more common getParameter method. See Chapter 4 (Handling the Client Request: Form Data) for details.

**Listing 19.9** Example of a SELECT menu that permits selection of multiple options

```
Languages you know:<BR>
<SELECT NAME="language" MULTIPLE>
  <OPTION VALUE="c">C
  <OPTION VALUE="c++">C++
  <OPTION VALUE="java" SELECTED>Java
  <OPTION VALUE="lisp">Lisp
  <OPTION VALUE="perl" SELECTED>Perl
  <OPTION VALUE="smalltalk">Smalltalk
</SELECT>
```

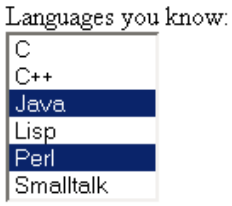


**Figure 19–18** A SELECT element that specifies MULTIPLE or SIZE results in a list box.

### HTML Element: `<OPTGROUP ...> ... </OPTGROUP>`

**Attributes:** `LABEL` (required)

This element, supported by Netscape 7.0 and Internet Explorer 6.0, permits grouping of menu choices. It is valid only inside a `SELECT` element.

#### LABEL

`LABEL` gives the text to display for the group of menu choices. Netscape and Internet Explorer display the label in bold text, using an oblique font.

Listing 19.10 creates a menu of server-side language choices. Here the menu choices are categorized into two groups by the `OPTGROUP` element. The first group is Common Servlet Languages, and the second group is Common CGI Languages. Figure 19–19 shows the displayed menu with `Java` selected for the common CGI language. For this selection the request data sent to the server is `language=java`. Be aware that no additional information is sent to the server by Netscape and Internet Explorer to indicate which `OPTGROUP` choice was selected. As a result, you should use unique values for all the menu choices, regardless of group.

#### Core Approach

*When using multiple `OPTGROUP` elements, ensure that all `OPTION`s use a unique name for the `VALUE`.*

**Listing 19.10** Example of a `SELECT` menu categorized into two groups with the `OPTGROUP` element

```
Server-side Languages:
<SELECT NAME="language">
  <OPTGROUP LABEL="Common Servlet Languages">
    <OPTION VALUE="java1">Java
  </OPTGROUP>
  <OPTGROUP LABEL="Common CGI Languages">
    <OPTION VALUE="c">C
    <OPTION VALUE="c++">C++
    <OPTION VALUE="java2">Java
    <OPTION VALUE="perl">Perl
    <OPTION VALUE="vb">Visual Basic
</OPTGROUP>
</SELECT>
```

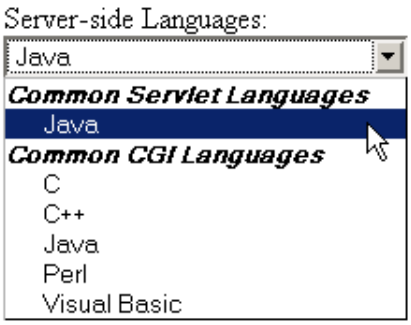


**Figure 19–19** A `SELECT` element using `OPTGROUP` to group menu choices in Netscape 7.0.

# 19.7 File Upload Controls

**HTML Element:** `<INPUT TYPE="FILE" ...>` **(No End Tag)**
**Attributes:** `NAME` (required), `VALUE` (ignored), `SIZE`, `MAXLENGTH`, `ACCEPT`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR` (nonstandard)

This element results in a filename textfield next to a Browse button. Users can enter a path directly in the textfield or click on the button to bring up a file selection dialog that lets them interactively choose the path to a file. When the form is submitted, the *contents* of the file are transmitted as long as an `ENCTYPE` of `multipart/form-data` was specified in the initial `FORM` declaration. For multipart data, you also need to specify `POST` as the method type. This element provides a convenient way to make user-support pages, with which the user sends a description of a problem along with any associated data or configuration files.

### Core Approach

*Always specify `ENCTYPE="multipart/form-data"` and `METHOD="POST"` in forms with file upload controls.*

Unfortunately, the servlet API provides no high-level tools to read uploaded files; you have to call `request.getInputStream` and parse the request yourself. Fortunately, numerous third-party libraries are available for this task. One of the most popular is from the Jakarta Commons library; for details, see http://jakarta.apache.org/commons/fileupload/.

### NAME

The NAME attribute identifies the textfield to the server-side program.

### VALUE

For security reasons, this attribute is ignored; only the end user can specify a filename. Otherwise, a malicious HTML author could steal client files by specifying a filename and then using JavaScript to automatically submit the form when the page is loaded.

### SIZE and MAXLENGTH

The SIZE and MAXLENGTH attributes are used the same way as in textfields, specifying the number of visible and maximum allowable characters, respectively.

### ACCEPT

The ACCEPT attribute is intended to be a comma-separated list of MIME types used to restrict the available filenames. However, very few browsers support this attribute.

### ONCHANGE, ONSELECT, ONFOCUS, and ONBLUR

These attributes are used by browsers that support JavaScript to specify the action to take when the mouse leaves the textfield after a change has occurred, when the user selects text in the textfield, when the textfield gets the input focus, and when the textfield loses the input focus, respectively.

For example, the code in Listing 19.11 creates a file upload control. Figure 19–20 shows the initial result, and Figure 19–21 shows a typical pop-up window that results when the Browse button is activated.

**Listing 19.11** Example of a file upload control

```
<FORM ACTION="http://localhost:8088/SomeProgram"
      ENCTYPE="multipart/form-data" METHOD="POST">
Enter data file below:<BR>
<INPUT TYPE="FILE" NAME="fileName">
</FORM>
```

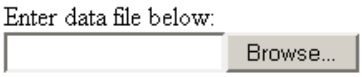


**Figure 19–20** Initial look of a file upload control.

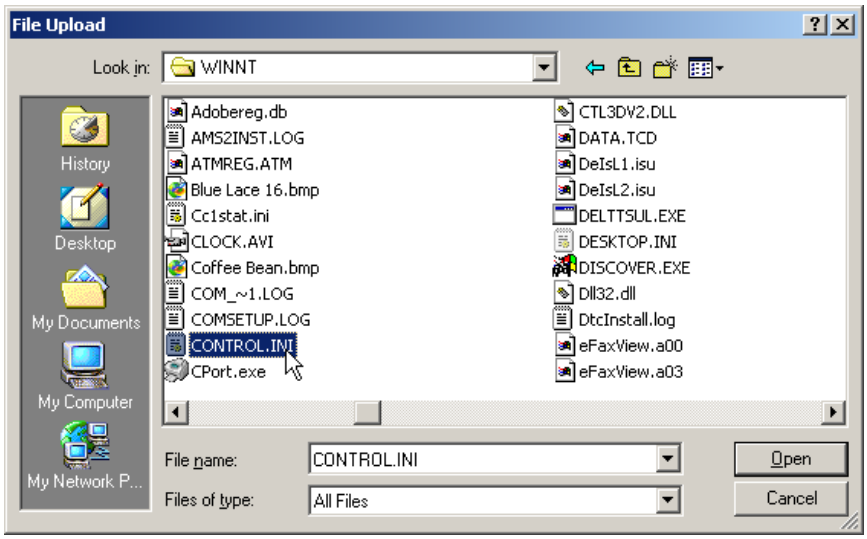


**Figure 19–21** A file chooser resulting from the user clicking on Browse in a file upload control on Windows 2000 Professional.

# 19.8 Server-Side Image Maps

In standard HTML, an element called `MAP` lets you associate URLs with various regions of an image; then, when the image is clicked in one of the designated regions, the browser loads the appropriate URL. This form of mapping is known as a *client-side image map*, since the determination of which URL to contact is made on the client and no server-side program is involved. HTML also supports *server-side image maps* that can be used within HTML forms. With such maps, an image is drawn; when the user clicks on the image, the coordinates of the click are sent to a server-side program.

Client-side image maps are simpler and more efficient than server-side ones and should be used when all you want to do is associate a fixed set of URLs with some predefined image regions. However, server-side image maps are appropriate if the URL needs to be computed (e.g,. for weather maps), the regions change frequently, or other form data needs to be included with the request. This section discusses two approaches to server-side image maps.

## IMAGE—Standard Server-Side Image Maps

The usual way to create server-side image maps is by means of an `<INPUT TYPE="IMAGE" ...>` element inside a form.

### HTML Element: `<INPUT TYPE="IMAGE" ...>` (No End Tag)

**Attributes:** `NAME` (required), `SRC`, `ALIGN`

This element displays an image that, when clicked, sends the form to the servlet or other server-side program specified by the enclosing form's `ACTION`. The name itself is not sent; instead, *`name`*`.x=`*`xpos`* and *`name`*`.y=`*`ypos`* are transmitted, where *`xpos`* and *`ypos`* are the coordinates of the mouse click relative to the upper-left corner of the image.

#### NAME

The `NAME` attribute identifies the textfield when the form is submitted.

#### SRC

`SRC` designates the URL of the associated image.

#### ALIGN

The `ALIGN` attribute has the same options (`TOP`, `MIDDLE`, `BOTTOM`, `LEFT`, `RIGHT`) and default (`BOTTOM`) as the `ALIGN` attribute of the `IMG` element and is used in the same way.

Listing 19.12 shows a simple example in which the form's `ACTION` specifies the `EchoServer` developed in Section 19.12. Figures 19–22 and 19–23 show the results before and after the image is clicked.

**Listing 19.12** ImageMap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>The IMAGE Input Control</TITLE>
</HEAD>
<BODY>
<H1 ALIGN="CENTER">The IMAGE Input Control</H1>
Which island is Java? Click and see if you are correct.
<FORM ACTION="http://localhost:8088/GeographyTester">
  <INPUT TYPE="IMAGE" NAME="map" SRC="images/indonesia.gif">
</FORM>
Of course, image maps can be implemented <B>in</B>
Java as well. :-)
</BODY></HTML>
```

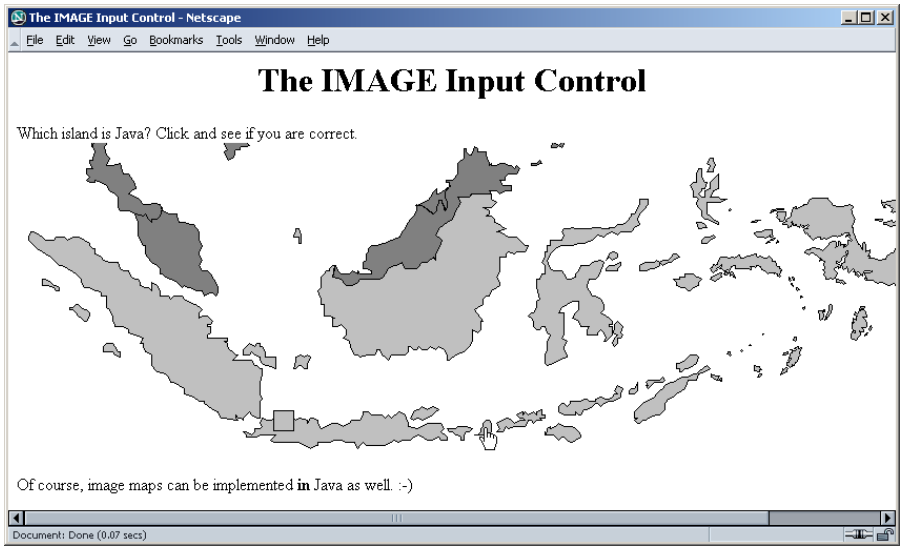


**Figure 19–22** An `IMAGE` input control with `NAME="map"`.

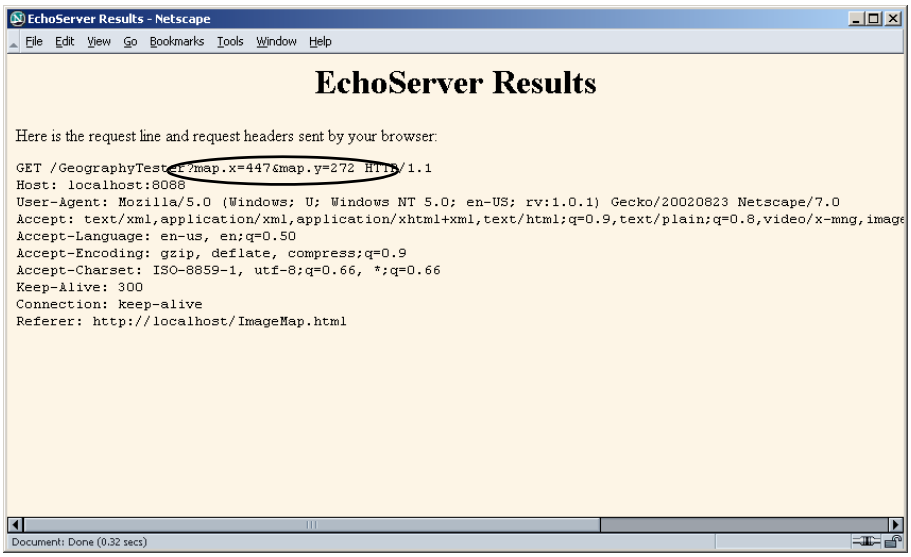


**Figure 19–23** Clicking on the image at (447, 272) submits the form and adds `map.x=447&map.y=272` to the form data.

# ISMAP—Alternative Server-Side Image Maps

`ISMAP` is an optional attribute of the `IMG` element and can be used in a manner similar to the `<INPUT TYPE="IMAGE" ...>` `FORM` entry. `ISMAP` is not actually a `FORM` element at all, but it can still be used for simple connections to servlets or other server-side programs. If an image with `ISMAP` is inside a hypertext link, then clicking on the image results in the coordinates of the click being sent to the specified URL. Coordinates are separated by commas and are specified in pixels relative to the top-left corner of the image.

For instance, Listing 19.13 embeds an image that uses the `ISMAP` attribute inside a hypertext link to http://localhost:8088/ChipTester, which is answered by the mini HTTP server developed in Section 19.12. Figure 19–24 shows the initial result, which is identical to what would have been shown had the `ISMAP` attribute been omitted. However, when the mouse button is pressed 270 pixels to the right and 189 pixels below the top-left corner of the image, the browser requests the URL http://localhost:8088/ChipTester?270,189 (as is shown in Figure 19–25).

If a server-side image map is used simply to select among a static set of destination URLs, then a client-side `MAP` element is a much better option because the server doesn't have to be contacted just to decide which URL applies. If the image map is intended to be mixed with other input elements, then the `IMAGE` input type

is preferred instead. However, for a stand-alone image map in which the URL associated with a region changes frequently or requires calculation, an image with `ISMAP` is a reasonable choice.

**Listing 19.13** IsMap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>The ISMAP Attribute</TITLE>
</HEAD>
<BODY>
<H1 ALIGN="CENTER">The ISMAP Attribute</H1>
<H2>Select a pin:</H2>
<A HREF="http://localhost:8088/ChipTester">
<IMG SRC="images/chip.gif" WIDTH=495 HEIGHT=200 ALT="Chip"
     BORDER=0 ISMAP></A>
</BODY></HTML>
```

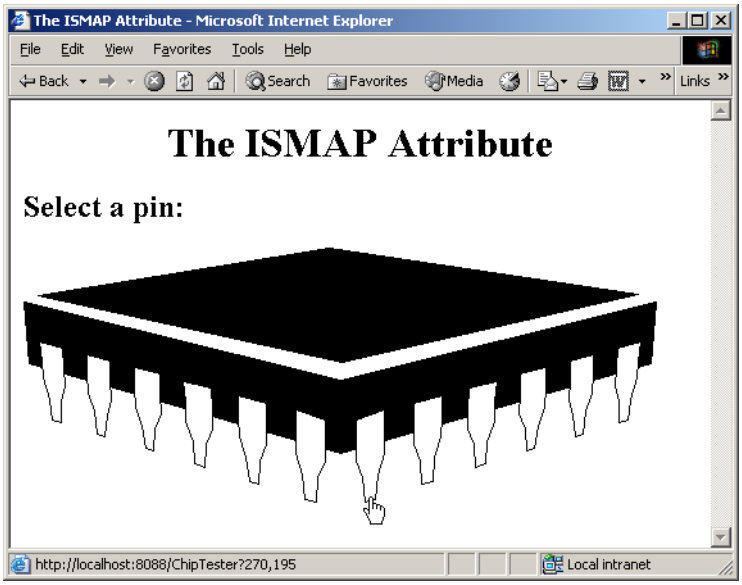


**Figure 19–24** Setting the `ISMAP` attribute of an `IMG` element inside a hypertext link changes what happens when the image is selected.

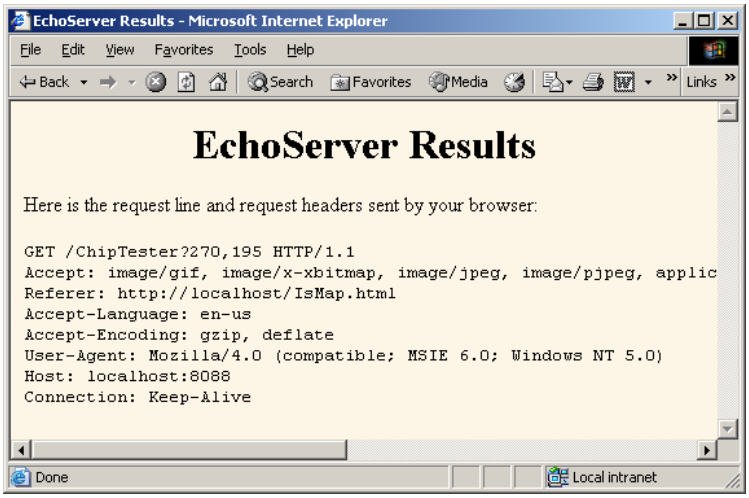


**Figure 19–25** When an `ISMAP` image is selected, the coordinates of the selection are transmitted with the URL.

# 19.9 Hidden Fields

Hidden fields do not affect the appearance of the page that is presented to the user. Instead, they store fixed names and values that are sent unchanged to the server, regardless of user input. Hidden fields are typically used for three purposes:

- **Tracking the user.** As the user moves around within the site, user IDs in hidden fields can be used to track which pages the user has visited or to indicate the selections made by the user. In practice, servlet authors typically rely on the servlet session tracking API rather than attempting to implement session tracking at this low level. For details on session tracking, see Chapter 9.
- **Providing predefined input to a server-side program.** When a variety of static HTML pages act as front ends to the same program on the server, predefined hidden fields can help provide information about the requesting source page. For example, an online store might pay commissions to people who refer customers to their site. In this scenario, the referring page could let visitors search the store's catalog by means of a form, but embed a hidden field giving its referral ID.

- **Storing contextual information in pages that are dynamically generated.** For example, in a table listing the items in a shopping cart, you can place a hidden field in each row to identify the particular item ID. In this manner, the user can modify the number of items ordered and, when submitted to the server-side program, the hidden field will identify the item being modified. The user never needs to see the item ID on the HTML page.

Note that the term "hidden" does not mean that the field cannot be discovered by the user, since it is clearly visible in the HTML source. Because there is no reliable way to "hide" the HTML that generates a page, authors are cautioned not to use hidden fields to embed passwords or other sensitive information.

### HTML Element: `<INPUT TYPE="HIDDEN" NAME="..." VALUE="...">` (No End Tag)

**Attributes:** `NAME` (required), `VALUE`

This element stores a name and a value, but no graphical element is created in the browser. The name/value pair is added to the form data when the form is submitted. For instance, with the following example, `itemID=brown001` will always get sent with the form data.

```
<INPUT TYPE="HIDDEN" NAME="itemID" VALUE="brown001">
```

## 19.10 Groups of Controls

HTML 4.0 defines the `FIELDSET` element, with an associated `LEGEND`, that can be used to visually group controls within a form. Note that the `FIELDSET` element works only in Netscape 6 and later and Internet Explorer 6 and later.

### HTML Element: `<FIELDSET> ... </FIELDSET>`

**Attributes:** None.

This element is used as a container to enclose controls and, optionally, a `LEGEND` element. It has no attributes beyond the universal ones for style sheets, language, and so forth. Listing 19.14 gives an example, with the result shown in Figure 19–26.

**Listing 19.14** Fieldset.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Grouping Controls</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Grouping Controls</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
<FIELDSET>
<LEGEND>Group One</LEGEND>
Field 1A: <INPUT TYPE="TEXT" NAME="field1A" VALUE="Field A"><BR>
Field 1B: <INPUT TYPE="TEXT" NAME="field1B" VALUE="Field B"><BR>
Field 1C: <INPUT TYPE="TEXT" NAME="field1C" VALUE="Field C"><BR>
</FIELDSET>
<FIELDSET>
<LEGEND ALIGN="RIGHT">Group Two</LEGEND>
Field 2A: <INPUT TYPE="TEXT" NAME="field2A" VALUE="Field A"><BR>
Field 2B: <INPUT TYPE="TEXT" NAME="field2B" VALUE="Field B"><BR>
Field 2C: <INPUT TYPE="TEXT" NAME="field2C" VALUE="Field C"><BR>
</FIELDSET>
</FORM>
</BODY></HTML>
```

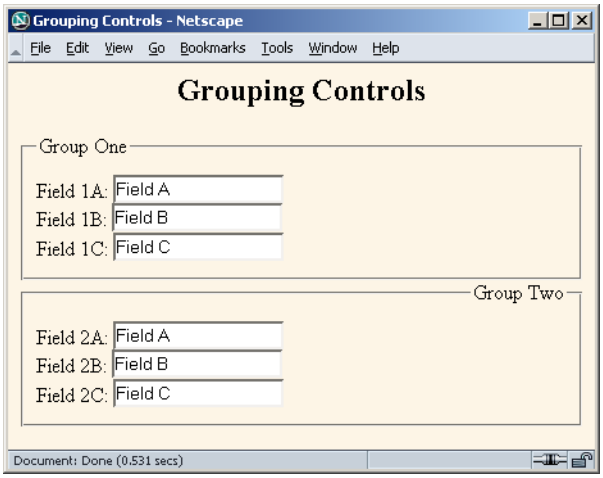


**Figure 19–26** The `FIELDSET` element lets you visually group related controls shown in Netscape 7.0.

**HTML Element:** `<LEGEND> ... </LEGEND>`
**Attributes:** `ALIGN`

This element places a label on the etched border that is drawn around the group of controls; it is legal only within an enclosing `FIELDSET`.

**ALIGN**
This attribute controls the position of the label. Legal values are `TOP`, `BOTTOM`, `LEFT`, and `RIGHT`, with `TOP` being the default. In Figure 19–26, the first group has the default legend alignment, and the second group stipulates `ALIGN="RIGHT"`. In HTML, style sheets are often a better way to control element alignment since they permit a single change to be propagated to multiple pages.

# 19.11 Tab Order Control

HTML 4.0 defines a `TABINDEX` attribute that can be used in any of the visual HTML elements. The `TABINDEX` value is an integer, and it controls the order in which elements receive the input focus when the TAB key is pressed.

In Listing 19.15 we present three textfields, `field1`, `field2`, and `field3`. Here, the `TABINDEX` attribute is set such that the tab order is from `field1` to `field3`, and then finally to `field2`. The HTML page is displayed in Figure 19–27.

Typically, the implied tab order used by the browser is top-to-bottom, left-to-right. If a nonstandard order is important in your application, you should explicitly declare a tab order.

**Listing 19.15** Tabindex.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Controlling TAB Order</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Controlling TAB Order</H2>
<FORM ACTION="http://localhost:8088/SomeProgram">
Field 1 (first tab selection):
<INPUT TYPE="TEXT" NAME="field1" TABINDEX=1><BR>
Field 2 (third tab selection):
<INPUT TYPE="TEXT" NAME="field2" TABINDEX=3><BR>
Field 3 (second tab selection):
<INPUT TYPE="TEXT" NAME="field3" TABINDEX=2><BR>
</FORM>
</BODY></HTML>
```

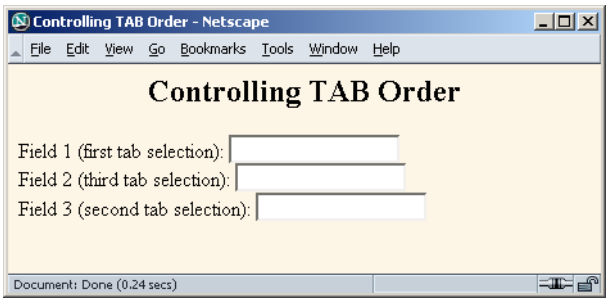


**Figure 19–27** Repeatedly pressing the TAB key cycles the input focus among the first, third, and second text fields, in that order (as dictated by `TABINDEX`).

# 19.12 A Debugging Web Server

This section presents a mini "Web server" that is useful when you are trying to understand the behavior of HTML forms. We used it for many of the examples earlier in the chapter. The server simply reads all the HTTP data sent to it by the browser, then returns a Web page with those lines embedded within a `PRE` element.

This server is also useful for debugging servlets. When something goes wrong, the first task is to determine if the problem lies in the way in which you collect data or the way in which you process it. The `WebClient` program of Section 3.6 lets you see the raw data resulting from the server-side program; `EchoServer` lets you see the raw data transmitted by the client form.

Starting the `EchoServer` on, say, port 8088 of your local machine, then changing your forms to specify `http://localhost:8088/` lets you see if the data being collected is in the format you expect. In addition to seeing the sent form data, `EchoServer` also shows the HTTP request headers sent from the browser.

## EchoServer

Listing 19.16 presents the top-level server code. You typically run `EchoServer` from the command line, specifying a port to listen on, or accepting the default port of 8088. `EchoServer` then accepts repeated HTTP requests from clients, packaging all HTTP data sent to it inside a Web page that is returned to the client. In most cases, the server reads until a blank line is received, indicating the end of `GET`, `HEAD`, or most other types of HTTP requests. In the case of `POST`, however, the server checks the `Content-Length` request header and reads that many bytes beyond the blank line.

Listings 19.17 and 19.18 present some utility classes that simplify networking. The `EchoServer` is built on top of them.

**Listing 19.16** EchoServer.java

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

/** A simple HTTP server that generates a Web page showing all
 *  the data that it received from the Web client (usually
 *  a browser). To use this server, start it on the system of
 *  your choice, supplying a port number if you want something
 *  other than port 8088. Call this system server.com. Next,
 *  start a Web browser on the same or a different system, and
 *  connect to http://server.com:8088/whatever. The resultant
 *  Web page will show the data that your browser sent. For
 *  debugging in a servlet or other server-side program, specify
 *  http://server.com:8088/whatever as the ACTION of your HTML
 *  form. You can send GET or POST data; either way, the
 *  resultant page will show what your browser sent.
 */

public class EchoServer extends NetworkServer {
  protected int maxRequestLines = 50;
  protected String serverName = "EchoServer";

  /** Supply a port number as a command-line
   *  argument. Otherwise, use port 8088.
   */

  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {}
    }
    new EchoServer(port, 0);
  }

  public EchoServer(int port, int maxConnections) {
    super(port, maxConnections);
    listen();
  }
```

**Listing 19.16** EchoServer.java *(continued)*

```
/** Overrides the NetworkServer handleConnection method to
 *  read each line of data received, save it into an array
 *  of strings, then send it back embedded inside a PRE
 *  element in an HTML page.
 */

public void handleConnection(Socket server)
    throws IOException{
  System.out.println
      (serverName + ": got connection from " +
       server.getInetAddress().getHostName());
  BufferedReader in = SocketUtil.getReader(server);
  PrintWriter out = SocketUtil.getWriter(server);
  String[] inputLines = new String[maxRequestLines];
  int i;
  for (i=0; i<maxRequestLines; i++) {
    inputLines[i] = in.readLine();
    if (inputLines[i] == null) // Client closed connection.
      break;
    if (inputLines[i].length() == 0) { // Blank line.
      if (usingPost(inputLines)) {
        readPostData(inputLines, i, in);
        i = i + 2;
      }
      break;
    }
  }
  printHeader(out);
  for (int j=0; j<i; j++) {
    out.println(inputLines[j]);
  }
  printTrailer(out);
  server.close();
}

// Send standard HTTP response and top of a standard Web page.
// Use HTTP 1.0 for compatibility with all clients.

private void printHeader(PrintWriter out) {
  out.println
    ("HTTP/1.0 200 OK\r\n" +
     "Server: " + serverName + "\r\n" +
     "Content-Type: text/html\r\n" +
     "\r\n" +
     "<!DOCTYPE HTML PUBLIC " +
       "\"-//W3C//DTD HTML 4.0 Transitional//EN\">\n" +
```

**Listing 19.16** EchoServer.java *(continued)*

```
      "<HTML>\n" +
      "<HEAD>\n" +
      "  <TITLE>" + serverName + " Results</TITLE>\n" +
      "</HEAD>\n" +
      "\n" +
      "<BODY BGCOLOR=\"#FDF5E6\">\n" +
      "<H1 ALIGN=\"CENTER\">" + serverName +
        " Results</H1>\n" +
      "Here is the request line and request headers\n" +
      "sent by your browser:\n" +
      "<PRE>");
}

// Print bottom of a standard Web page.

private void printTrailer(PrintWriter out) {
  out.println
    ("</PRE>\n" +
     "</BODY>\n" +
     "</HTML>\n");
}

// Normal Web page requests use GET, so this server can simply
// read a line at a time. However, HTML forms can also use
// POST, in which case we have to determine the number of POST
// bytes that are sent so we know how much extra data to read
// after the standard HTTP headers.

private boolean usingPost(String[] inputs) {
  return(inputs[0].toUpperCase().startsWith("POST"));
}

private void readPostData(String[] inputs, int i,
                          BufferedReader in)
    throws IOException {
  int contentLength = contentLength(inputs);
  char[] postData = new char[contentLength];
  in.read(postData, 0, contentLength);
  inputs[++i] = new String(postData, 0, contentLength);
}

// Given a line that starts with Content-Length,
// this returns the integer value specified.
```

**Listing 19.16** EchoServer.java *(continued)*

```
  private int contentLength(String[] inputs) {
    String input;
    for (int i=0; i<inputs.length; i++) {
      if (inputs[i].length() == 0)
        break;
      input = inputs[i].toUpperCase();
      if (input.startsWith("CONTENT-LENGTH"))
        return(getLength(input));
    }
    return(0);
  }

  private int getLength(String length) {
    StringTokenizer tok = new StringTokenizer(length);
    tok.nextToken();
    return(Integer.parseInt(tok.nextToken()));
  }
}
```

**Listing 19.17** NetworkServer.java

```
import java.net.*;
import java.io.*;

/** A starting point for network servers. You'll need to
 *  override handleConnection, but in many cases listen can
 *  remain unchanged. NetworkServer uses SocketUtil to simplify
 *  the creation of the PrintWriter and BufferedReader.
 */

public class NetworkServer {
  private int port, maxConnections;

  /** Build a server on specified port. It will continue to
   *  accept connections, passing each to handleConnection until
   *  an explicit exit command is sent (e.g., System.exit) or
   *  the maximum number of connections is reached. Specify
   *  0 for maxConnections if you want the server to run
   *  indefinitely.
   */
```

**Listing 19.17** NetworkServer.java *(continued)*

```
public NetworkServer(int port, int maxConnections) {
  setPort(port);
  setMaxConnections(maxConnections);
}

/** Monitor a port for connections. Each time one is
 *  established, pass resulting Socket to handleConnection.
 */

public void listen() {
  int i=0;
  try {
    ServerSocket listener = new ServerSocket(port);
    Socket server;
    while((i++ < maxConnections) || (maxConnections == 0)) {
      server = listener.accept();
      handleConnection(server);
    }
  } catch (IOException ioe) {
    System.out.println("IOException: " + ioe);
    ioe.printStackTrace();
  }
}

/** This is the method that provides the behavior to the
 *  server, since it determines what is done with the
 *  resulting socket. <B>Override this method in servers
 *  you write.</B>
 *  <P>
 *  This generic version simply reports the host that made
 *  the connection, shows the first line the client sent,
 *  and sends a single line in response.
 */

protected void handleConnection(Socket server)
    throws IOException{
  BufferedReader in = SocketUtil.getReader(server);
  PrintWriter out = SocketUtil.getWriter(server);
  System.out.println
    ("Generic Network Server: got connection from " +
     server.getInetAddress().getHostName() + "\n" +
     "with first line '" + in.readLine() + "'");
  out.println("Generic Network Server");
  server.close();
}
```

**Listing 19.17** NetworkServer.java *(continued)*

```
  /** Gets the max connections server will handle before
   *  exiting. A value of 0 indicates that server should run
   *  until explicitly killed.
   */

  public int getMaxConnections() {
    return(maxConnections);
  }

  /** Sets max connections. A value of 0 indicates that server
   *  should run indefinitely (until explicitly killed).
   */

  public void setMaxConnections(int maxConnections) {
    this.maxConnections = maxConnections;
  }

  /** Gets port on which server is listening. */

  public int getPort() {
    return(port);
  }

  /** Sets port. <B>You can only do before "connect" is
   *  called.</B> That usually happens in the constructor.
   */

  protected void setPort(int port) {
    this.port = port;
  }
}
```

**Listing 19.18** SocketUtil.java

```
import java.net.*;
import java.io.*;

/** A shorthand way to create BufferedReaders and
 *  PrintWriters associated with a Socket.
 */

public class SocketUtil {
  /** Make a BufferedReader to get incoming data. */
```

**Listing 19.18** SocketUtil.java *(continued)*

```
  public static BufferedReader getReader(Socket s)
      throws IOException {
    return(new BufferedReader(
       new InputStreamReader(s.getInputStream())));
  }

  /** Make a PrintWriter to send outgoing data.
   *  This PrintWriter will automatically flush stream
   *  when println is called.
   */

  public static PrintWriter getWriter(Socket s)
      throws IOException {
    // Second argument of true means autoflush.
    return(new PrintWriter(s.getOutputStream(), true));
  }
}
```