The Form Object

CHAPTER 21

In This Chapter

The form object as container of form elements

How to submit forms via e-mail

Processing form validations

The majority of scripting in an HTML document takes place in and around forms. Because forms tend to be the primary user interface elements of HTML documents, this fact shouldn't be surprising. The challenge of scripting forms and form elements often comes from getting object references just right: The references can get pretty long by the time you start pointing to the property of a form element (which is part of a form, which is part of a document, which is part of a window or frame).

The Form in the Object Hierarchy

Take another look at the JavaScript object hierarchy in Navigator 4 (refer back to Figure 16-1). The form object can contain a wide variety of form element objects, which are covered in Chapters 22 through 24. In this chapter, however, the focus is strictly on the container.

The good news on the compatibility front is that with the exception of differences when submitting forms to CGI programs on the server, much of the client-side scripting works on all scriptable browsers. Where differences exist, I point out the real gotchas.

Form Object

Properties	*Methods*	*Event Handlers*
action	handleEvent()	onReset=
elements[]	reset()	onSubmit=
encoding	submit()	
length		
method		
name		
target		

Syntax

Creating a form:

```
<FORM
    [NAME="formName"]
    [TARGET="windowName"]
    [ACTION="serverURL"]
    [METHOD=GET | POST]
    [ENCTYPE="MIMEType"]
    [onReset="handlerTextOrFunction"]
    [onSubmit="handlerTextOrFunction"] >
</FORM>
```

Accessing form properties or methods:

```
[window.] document.formName.property | method([parameters])
[window.] document.forms[index].property | method([parameters])
```

About this object

Forms and their elements are the primary, two-way gateways between users and JavaScript scripts. A form element provides the only way that users can enter textual information or make a selection from a predetermined set of choices, whether those choices appear in the form of an on/off checkbox, one of a set of mutually exclusive radio buttons, or a selection from a list.

As you have seen in many Web sites, the form is the avenue for the user to enter information that gets sent to the server housing the Web files. Just what the server can do with this information depends on the CGI (Common Gateway Interface) programs running on the server. If your Web site runs on a server directly under your control (that is, it is "in-house" or "hosted" by a service), you have the freedom to set up all kinds of data-gathering or database search programs to interact with the user. But if you rely on an Internet service provider (ISP) to house your HTML files, you're limited to a usually plain set of CGI programs available to all customers of the service. Custom databases or transactional services are rarely provided for this kind of dial-up Internet service — popular with individuals and small businesses who cannot justify the cost of maintaining their own servers.

Regardless of your Internet server status, you can find plenty of uses for JavaScript scripts in documents. For instance, rather than using data exchanges (and Internet bandwidth) to gather raw user input and report any errors, a JavaScript-enhanced document can preprocess the information to make sure that it uses the format that is most easily received by your back-end database or other programs. All corrective interaction takes place in the browser, without one extra bit flowing across the Net. I devote all of Chapter 37 to form data-validation techniques.

How you define a form object (independent of the user interface elements, described later in this chapter) depends a great deal on how you plan to use the

information from the form's elements. If you're using the form completely for JavaScript purposes (that is, no queries or postings will be going to the server), you do not need to use the ACTION, TARGET, and METHOD attributes. But if your Web page will be feeding information or queries back to a server, you need to specify at least the ACTION and METHOD attribute; you need to also specify the TARGET attribute if the resulting data from the server is to be displayed in a window other than the calling window and the ENCTYPE attribute if your form's scripts fashion the server-bound data in a MIME type other than a plain ASCII stream.

References to form elements

For most client-side scripting, user interaction comes from the elements within a form; the form object becomes merely a repository for the various elements. If your scripts will be performing any data validation checks on user entries prior to submission or other calculations, many statements will have the form object as part of the reference to the element.

A complex HTML document can have multiple form objects. Each <FORM>...</FORM> tag pair defines one form. You won't receive any penalties (except for potential confusion on the part of someone reading your script) if you reuse a name for an element in each of a document's forms. For example, if each of three forms has a grouping of radio buttons with the name "choice," the object reference to each button ensures that JavaScript won't confuse them. The reference to the first button of each of those button groups is as follows:

```
document.forms[0].choice[0]
document.forms[1].choice[0]

document.forms[2].choice[0]
```

Remember, too, that you can create forms (or any HTML object for that matter) on the fly when you assemble HTML strings for writing into other windows or frames. Therefore, you can determine various attributes of a form from settings in an existing document.

Passing forms and elements to functions

When a form or form element contains an event handler that calls a function defined elsewhere in the document, a couple shortcuts are available that you can use to simplify the task of addressing the objects in the function. Failure to grasp this concept not only causes you to write more code than you have to, but also hopelessly loses you when you try to trace somebody else's code in his or her JavaScripted document. The watchword in event handler parameters is

```
this
```

which represents the current object that contains the event handler attribute. For example, consider the function and form definition in Listing 21-1. The entire user interface for this listing consists of form elements, as shown in Figure 21-1.

Listing 21-1: **Passing the Form Object as a Parameter**

```
<HTML>
<HEAD>
<TITLE>Beatle Picker</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function processData(form) {
        for (var i = 0; i < form.Beatles.length; i++) {
           if (form.Beatles[i].checked) {
              break
           }
        }
        var chosenBeatle = form.Beatles[i].value
        var chosenSong = form.song.value
        alert("Looking to see if " + chosenSong + " was written by " +
chosenBeatle + "...")
}

function checkSong(songTitle) {
        var enteredSong = songTitle.value
        alert("Making sure that " + enteredSong + " was recorded by the
Beatles.")
}
</SCRIPT>
</HEAD>

<BODY>
<FORM NAME="Abbey Road">
Choose your favorite Beatle:
<INPUT TYPE="radio" NAME="Beatles" VALUE="John Lennon"
CHECKED="true">John
<INPUT TYPE="radio" NAME="Beatles" VALUE="Paul McCartney">Paul
<INPUT TYPE="radio" NAME="Beatles" VALUE="George Harrison">George
<INPUT TYPE="radio" NAME="Beatles" VALUE="Ringo Starr">Ringo<P>

Enter the name of your favorite Beatles song:<BR>
<INPUT TYPE="text" NAME="song" VALUE="Eleanor Rigby"
onChange="checkSong(this)"><P>
<INPUT TYPE="button" NAME="process" VALUE="Process Request..."
onClick="processData(this.form)">
</FORM>
</BODY>
</HTML>
```

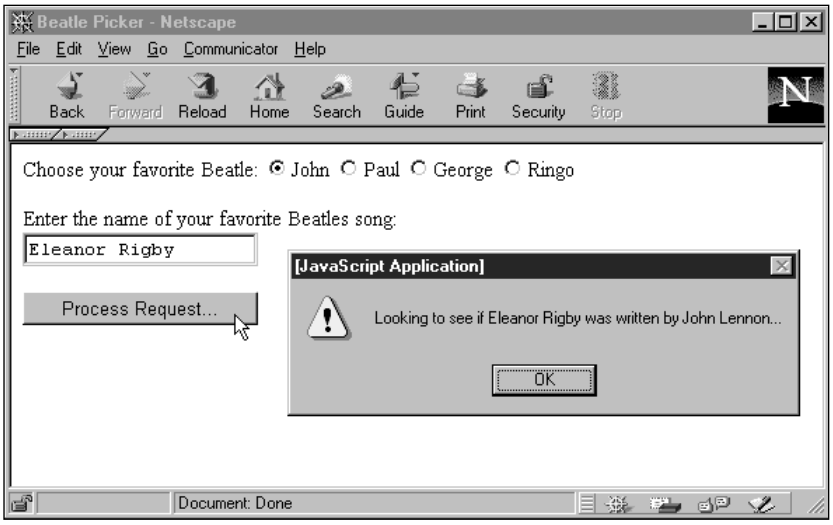

Figure 21-1: A variety of elements compose the form of Listing 21-1.

If you want to summon any properties of the form elements to work on them inside the `processData()` function, you can go about it in two different ways. One is to have the `onClick=` event handler (in the button element at the bottom of the document) call the `processData()` function and not pass any parameters. Inside the function, all references to objects (such as the radio buttons or the song field) must be complete references, such as

```
document.forms[0].song.value
```

to retrieve the value entered into the "song" field.

A more efficient way is to send the form object as a parameter with the call to the function (as shown in Listing 21-1). By specifying `this.form` as the parameter, you tell JavaScript to send along everything it knows about the form from which the function is being called. At the function, that form object is assigned to a variable name (arbitrarily set to `form` here) that appears in parentheses after the function name. I've used the parameter variable name `form` here because it represents an entire form. But you can use any valid variable name you like.

Part of the information that comes along with that form is its address among all JavaScript objects loaded with the document. That means as long as statements refer to that form object (by its variable name) the full address is automatically part of the reference. Thus, here I can use `form` to take the place of `document.forms[0]` in any address. To get the value of the song field, the reference is

```
form.song.value
```

Had I assigned the form object to a parameter variable called `sylvester`, the reference would have been

```
sylvester.song.value
```

This referencing methodology works for retrieving or setting properties and calling an object's methods. Another version of the `this` parameter passing style is simply to use the word `this` as the parameter. Unlike `this.form`, which passes a reference to the entire form connected to a particular element, `this` passes only a reference to that one element. In Listing 21-1, you could add an event handler to the song field to do some validation of the entry (to make sure that the entry appears in a database array of Beatles' songs created elsewhere in the document). Therefore, you want to send only the field object to the function for analysis:

```
<INPUT TYPE="text" NAME="song" onChange="checkSong(this)"><P>
```

You then have to create a function to catch this call:

```
function checkSong(songTitle) {
       var enteredSong = songTitle.value
       alert("Making sure that " + enteredSong + " was recorded by the
Beatles.")
}
```

Within this function, you can go straight to the heart — the `value` property of the field element without a long address. The entire field object came along for the ride with its complete address.

One further extension of this methodology is to pass only a single property of a form element as a parameter. In the last example, since the `checkSong()` function needs only the `value` property of the field, the event handler could have passed `this.value` as a parameter. Because `this` refers to the very object in which the event handler appears, `this.`*`propertyName`* syntax lets you extract and pass along a single property:

```
<INPUT TYPE="text" NAME="song" onChange="checkSong(this.value)"><P>
```

A benefit of this way of passing form element data is that the function doesn't have to do as much work:

```
function checkSong(songTitle) {
       alert("Making sure that " + songTitle + " was recorded by the
Beatles.")
}
```

Therefore, I suggest passing the entire form element (`this`) when the function needs to make multiple accesses to that element (perhaps reading one property and then writing back to that very element's `value` property, such as converting a text field to all uppercase letters). But if only one property is needed in the function, pass only that one property (`this.`*`propertyName`*). Lastly, if the function will access multiple elements in a form (for example, a button click means that the function must retrieve a field's content), pass the entire form (`this.form`). Also be aware that you can submit multiple parameters (for example, `onClick="someFunction(this.form, this.name)"`) or even an entirely different object from the same form (for example, `onClick="someFunction(this.form.emailAddr.value)"`). Simply adjust your function's incoming parameters accordingly (see Chapter 34 for more details about custom functions).

E-mailing forms

A common request among scripters is the ability to send a form via e-mail back to the page's author. This includes the occasional desire to send "secret" e-mail back to the author whenever someone visits the Web site. Let me address the privacy issue first.

A site visitor's e-mail address is valuable personal information that should not be retrieved from the user without his or her permission or knowledge. That's one reason why Netscape plugged a privacy hole in Navigator 2 that allowed submitting a form to a `mailto:` URL without requesting permission from the user. Some workarounds for this could be used in Navigator 3, but I do not condone surreptitiously lifting e-mail addresses, so I choose not to publicize those workarounds here.

Microsoft, on the other hand, goes too far in preventing forms e-mailing. While Netscape's browsers reveal to the user in an alert that an e-mail message bearing the user's e-mail address (as stored in the browser's preferences) will be sent upon approval, Internet Explorer 3 does not send form content via e-mail at all; Internet Explorer 4 sends form content as an attachment, but only after displaying a mail composition window to the user. In all cases, the mail composition window appears to the user.

Many ISPs that host Web sites provide standard CGIs for forwarding forms to an e-mail address of your choice. This manner of capturing form data, however, does not also capture the visitor's e-mail address unless your form has a field where the visitor voluntarily enters that information.

Back to Navigator, if you want to have forms submitted as e-mail messages, you must attend to three `<FORM>` tag attributes. The first is the `METHOD` attribute. It must be set to `POST`. Next comes the `ACTION` attribute. Normally the spot for a URL to another file or server CGI, you substitute the special mailto: URL followed by an optional parameter for the subject. Unlike the more fully featured `mailto:` URLs possible with the location and link objects (see Chapters 15 and 17), a form's `mailto:` URL can include at most the subject for the message. Here is a sample:

```
ACTION="mailto:prez@whitehouse.gov?subject=Opinion Poll"
```

The last attribute of note is `ENCTYPE`. If you omit this attribute, Navigator sends the form data as an attachment consisting of escaped name-value pairs, as in this example:

```
name=Danny+Goodman&rank=Scripter+First+Class&serialNumber=042
```

But if you set the `ENCTYPE` attribute to `"text/plain"`, the form name-value pairs are placed in the body of the mail message in a more human-readable format:

```
name=Danny Goodman
rank=Scripter First Class
serialNumber=042
```

To sum up, here would be the complete `<FORM>` tag for e-mailing the form in Navigator:

```
<FORM NAME="entry"
        METHOD=POST
        ACTION="mailto:prez@whitehouse.gov?subject=Opinion Poll"
        ENCTYPE="text/plain">
```

Changing form attributes

Navigator exposes all form attributes as modifiable properties. Therefore, you can change, say, the action of a form via a script in response to user interaction on your page. For example, you might have two different CGI programs on your server depending on whether a form's checkbox is checked.

Form attribute properties cannot be changed on the fly in Internet Explorer 3. They can be modified in Internet Explorer 4.

Buttons in forms

A common mistake that newcomers to scripting make is defining all clickable buttons as the submit type of input object (`<INPUT TYPE="submit">`). The submit style of button does exactly what it says: It submits the form. If you don't set any `METHOD` or `ACTION` attributes of the `<FORM>` tag, the browser inserts its default values for you: `METHOD=GET and ACTION=<pageURL>`. When a form with these attributes is submitted, the page reloads itself and resets all field values to their initial values.

Use a submit button only when you want the button to actually submit the form. If you want a button for other types of action, use the button style (`<INPUT TYPE="button">`).

Redirection after submission

All of us have submitted a form to a site and seen a "Thank You" page come back from the server to verify that our submission was accepted. This is warm and fuzzy, if not logical, feedback for the submission action. It is not surprising that you would want to re-create that effect even if the submission is to a `mailto:` URL. Unfortunately, a problem gets in the way.

A commonsense approach to the situation would call for a script to perform the submission (via the `form.submit()` method) and then navigate to another page that does the "Thank You." Here would be such a scenario from inside a function triggered by a click of a link surrounding a nice graphical Submit button:

```
function doSubmit() {
        document.forms[0].submit()
        location = "thanks.html"
}
```

The problem is that when another statement executes after the `form.submit()` method, the submission is canceled. In other words, the script does not wait for the submission to complete itself and verify to the browser that all is well (even though the browser appears to know how to track that information, given the status bar feedback during submission). The point is, because JavaScript does not provide an event that is triggered by a successful submission, there is no sure-fire way to display your own "Thank You" page.

Don't be tempted by the `window.setTimeout()` method to change the location after some number of milliseconds following the `form.submit()` method. You cannot predict how fast the network and/or server will be for every visitor. If the submission does not fully complete before the timeout ends, then the submission is still canceled, even if it is partially completed.

It's too bad we don't have this power at our disposal yet. Perhaps a future version of the document object model will provide an event that lets us do something only after a successful submission.

Form element arrays

Since the first implementation of JavaScript in Navigator 2, Netscape's document object model has provided a feature beneficial to a lot of scripters. If you create a series of like-named objects, they automatically become an array of objects accessible via array syntax (see Chapter 7). This is particularly helpful if you are creating forms with columns and rows of fields, such as in an order form. By assigning the same name to all fields in a column, you can use `for` loops to cycle through each row using the loop index as an array index.

As an example, the following code shows a typical function that calculates the total for an order form row (and calls another custom function to format the value):

```
function extendRows(form) {
        for (var i = 0; i < Qty.length; i++) {
            var rowSum = form.Qty[i].value * form.Price[i].value
            form.Total[i].value = formatNum(rowSum,2)
        }
}
```

All fields in the Qty column are named Qty. The item in the first row has the array index value of zero and is addressed as `form.Qty[i]`.

Unfortunately, Internet Explorer 3 does not turn like-named fields into an array of references, although Internet Explorer 4 does. But you can still script repetitive moves through an organized set of fields. The key is to assign names to the fields that include their index numbers: `Qty0`, `Qty1`, `Qty2`, and so on. You can even assign these names in a `for` loop that generates the table:

```
for (var i = 0; i <= 10; i++) {
        ...
        document.write("<INPUT TYPE='text' NAME='Qty" + i + "'>")
        ...
}
```

Later, when it comes time to work with the fields, you can use the indexing scheme to address the fields:

```
for (var i = 0; i < Qty.length; i++) {
        var rowSum = form["Qty" + i].value * form["Price" + i].value
        form["Total" + i].value = formatNum(rowSum,2)
}
```

In other words, construct names for each item, and use those names as array index names. This solution is backward and forward compatible.

Properties

action

Value: URL **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

The `action` property (along with the `method` and `target` properties) is primarily for HTML authors whose pages communicate with server-based CGI scripts. This property is the same as the value you assign to the `ACTION` attribute of a `<FORM>` definition. The value is typically a URL on the server where queries or postings are sent for submission.

User input may affect how you want your page to access a server. For example, a checked box in your document may set a form's `action` property so that a CGI script on one server handles all the input, whereas an unchecked box means the form data goes to a CGI script on an entirely different server. Or, one setting may direct the action to one `mailto:` address, whereas another setting sets the `action` property to a different `mailto:` address.

Although the specifications for all three related properties indicate that they can be set on the fly, such changes are ephemeral. A soft reload eradicates any settings you make to these properties, so at best you'd make changes only in the same scripts that submit the form (see `form.submit()`).

This value is not modifiable in Internet Explorer 3 but is in Internet Explorer 4.

Example

```
document.forms[0].action = "mailto:jdoe@giantco.com"
```

Related Items: `form.method` property; `form.target` property; `form.encoding` property.

elements

Value: Array of sub-objects **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

Elements include all the user interface elements defined for a form: text fields, buttons, radio buttons, checkboxes, selection lists, and more. Like some other

JavaScript object properties, the elements property is an array of all items defined within the current HTML document. For example, if a form defines three <INPUT> items, the elements property for that form is an array consisting of three entries, one for each item. Each entry is the full object specification for that element; so, to extract properties or call methods for those elements, your script must dig deeper in the reference. Therefore, if the first element of a form is a text field, and you want to extract the string currently showing in the field (a text element's value property), the reference looks like this:

```
document.forms[0].elements[0].value
```

Notice that this reference summons two array-oriented properties along the way: one for the document's forms property and, subsequently, one for the form's elements property.

You can access an element in other ways, too (see discussions of individual element objects later in this chapter). An advantage to using the elements property occurs when you have a form with many elements, each with a related or potentially confusing name. In such circumstances, references that point directly to an element's name may be more difficult to trace or read. On the other hand, the order of entries in an elements array depends entirely upon their order in the HTML document — the first <INPUT> item in a form is elements[0], the second is elements[1], and so on. If you redesign the physical layout of your form elements after writing scripts for them, the index values you originally had for referencing a specific form may no longer be valid. Referencing an element by name, however, works no matter how you move the form elements around in your HTML document.

To JavaScript, an element is an element, whether it is a radio button or textarea. If your script needs to loop through all elements of a form in search of particular kinds of elements, use the type property of every form object (Navigator 3+ and Internet Explorer 4+) to identify which kind of object it is. The type property consists of the same string used in the TYPE= attribute of an <INPUT> tag.

Overall, my personal preference is to generate meaningful names for each form element and use those names in references throughout my scripts. Just the same, if I have a script that must poll every element or contiguous range of elements for a particular property value, the indexed array of elements facilitates using a repeat loop to examine each one efficiently.

Example

The document in Listing 21-2 demonstrates a practical use of the elements property. A form contains four fields and some other elements mixed in between (Figure 21-2). The first part of the function that acts on these items repeats through all the elements in the form to find out which ones are text objects and which text objects are empty. Notice how I use the type property to separate objects from the rest, even when radio buttons appear amid the fields. If one field has nothing in it, I alert the user and use that same index value to place the insertion point at the field with the field's focus() method.

Listing 21-2: **Using the form.elements[] Array**

```
<HTML>
<HEAD>
<TITLE>Elements Array</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
function verifyIt() {
        var form = document.forms[0]
        for (i = 0; i < form.elements.length; i++) {
           if (form.elements[i].type == "text" &&
form.elements[i].value == ""){
              alert("Please fill out all fields.")
              form.elements[i].focus()
              break
           }
           // more tests
        }
        // more statements
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
Enter your first name:<INPUT TYPE="text" NAME="firstName"><P>
Enter your last name:<INPUT TYPE="text" NAME="lastName"><P>
<INPUT TYPE="radio" NAME="gender">Male
<INPUT TYPE="radio" NAME="gender">Female <P>
Enter your address:<INPUT TYPE="text" NAME="address"><P>
Enter your city:<INPUT TYPE="text" NAME="city"><P>
<INPUT TYPE="checkbox" NAME="retired">I am retired
</FORM>
<FORM>
<INPUT TYPE="button" NAME="act" VALUE="Verify" onClick="verifyIt()">
</FORM>
</BODY>
</HTML>
```

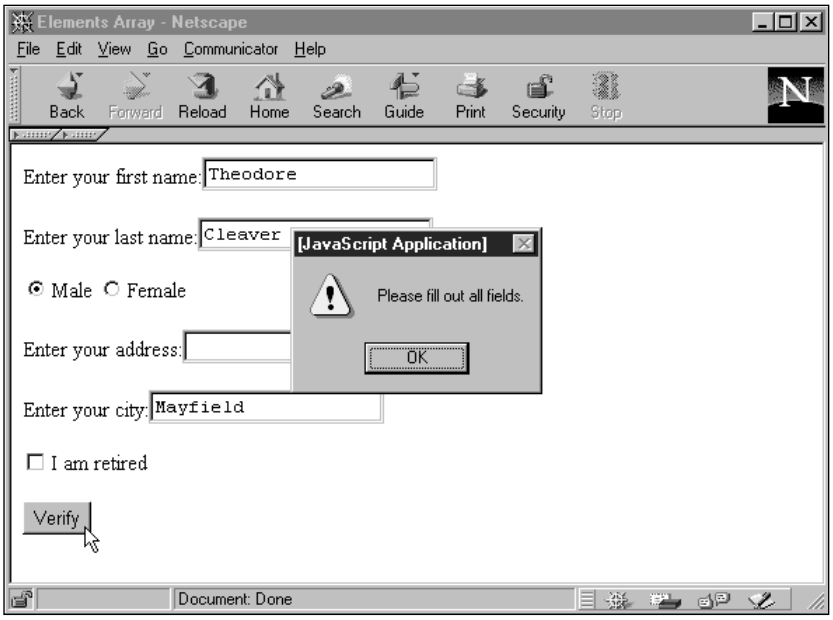

Figure 21-2: A document with mixed elements

Related Items: text, textarea, button, radio, checkbox, and select objects.

encoding

Value: MIMETypeString **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

You can define a form to alert a server that the data being submitted is in a MIME type. This property reflects the setting of the `ENCTYPE` attribute in the form definition. For `mailto:` URLs, I recommend setting this value (in the tag or via script) to `"text/plain"` to have the form contents placed in the mail message body. If the definition does not have an `ENCTYPE` attribute, this property is an empty string.

This value is not modifiable in Internet Explorer 3 but is in Internet Explorer 4.

Example

```
formMIME = document.forms[0].encoding
```

Related Items: `form.action` method; `form.method` method.

method

Value: "GET" or "POST" **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

A form's `method` property is either the `GET` or `POST` values assigned to the `METHOD` attribute in a `<FORM>` definition. Terminology overlaps here a bit, so be careful to distinguish a form's method of transferring its data to a server from the object-oriented method (action or function) that all JavaScript forms have.

Of primary importance to HTML documents that submit a form's data to a server-based CGI script is the `method` property, which determines the format used to convey this information. For example, to submit a form to a `mailto:` URL, the `method` property must be `POST`. Details of forms posting and CGI processing are beyond the scope of this book. Consult HTML or CGI documentation to determine which is the appropriate setting for this attribute in your Web server environment. If a form does not have a `METHOD` attribute explicitly defined for it, the default value is `GET`.

This value is not modifiable in Internet Explorer 3 but is in Internet Explorer 4.

Example

```
formMethod = document.forms[0].method
```

Related Items: `form.action` property; `form.target` property; `form.encoding` property.

name

Value: String **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

Assigning a name to a form via the `NAME` attribute is optional but highly recommended when your scripts need to reference a form or its elements. This attribute's value is retrievable as the `name` property of a form. You won't have much need to read this property unless you're inspecting another source's document for its form construction, as in

```
var formName = parent.frameName.document.forms[0].name
```

target

Value: WindowNameString **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

Whenever an HTML document submits a query to a server for processing, the server typically sends back an HTML page — whether it is a canned response or, more likely, a customized page based on the input provided by the user. You see this situation all the time when you perform a search at Web search sites, such as Yahoo!, Lycos, and AltaVista. In a multiframe or multiwindow environment, you may want to keep the form part of this transaction in view for the user, while leaving the responding page in a separate frame or window for viewing. The purpose of the TARGET attribute of a <FORM> definition is to enable you to specify where the output from the server's query should be displayed.

The value of the target property is the name of the window or frame. For instance, if you define a frameset with three frames and assign the names Frame1, Frame2, and Frame3 to them, you need to supply one of these names (as a quoted string) as the parameter of the TARGET attribute of the <FORM> definition. Navigator and compatible browsers also observe four special window names that you can use in the <FORM> definition: _top, _parent, _self, and _blank. To set the target as a separate subwindow opened via a script, be sure to use the window name from the window.open() method's second parameter, and not the window object reference that the method returns.

This value is not modifiable in Internet Explorer 3 but is in Internet Explorer 4.

Example

```
document.forms[0].target = "resultFrame"
```

Related Items: form.action property; form.method property; form.encoding property.

Methods

handleEvent(*event*)

Returns: Nothing.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			

See the discussion of the `window.handleEvent()` method in Chapter 14 and the event object in Chapter 33 for details on this ubiquitous form element method.

`reset()`

Returns: Nothing.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔			✔

A common practice, especially with a long form, is to provide a button that enables the user to return all the form elements to their default settings. The standard Reset button (a separate object type described in Chapter 23) does that task just fine. But if you want to clear the form using script control, you must do so by invoking the `reset()` method for the form. More than likely, such a call will come from outside the form, perhaps from a function or from a graphical button. In such a case, make sure that the reference to the `reset()` method includes the complete reference to the form you want to reset — even if the page only has one form defined for it.

Example

In Listing 21-3, the act of resetting the form is assigned to the `HREF` attribute of a link object (that is attached to a graphic called `reset.jpg`). I use the `javascript:` URL to invoke the `reset()` method for the form directly (in other words, without doing it via function).

Listing 21-3: form.reset() and form.submit() Methods

```
<HTML>
<HEAD>
<TITLE>Registration Form</TITLE>
</HEAD>
<BODY>
<FORM NAME="entries" METHOD=POST ACTION="http://www.u.edu/pub/cgi-
bin/register">
Enter your first name:<INPUT TYPE="text" NAME="firstName"><P>
Enter your last name:<INPUT TYPE="text" NAME="lastName"><P>
Enter your address:<INPUT TYPE="text" NAME="address"><P>
Enter your city:<INPUT TYPE="text" NAME="city"><P>
<INPUT TYPE="radio" NAME="gender" CHECKED>Male
<INPUT TYPE="radio" NAME="gender">Female <P>
<INPUT TYPE="checkbox" NAME="retired">I am retired
</FORM>
```

```
<P>
<A HREF="javascript:document.forms[0].submit()"><IMG SRC="submit.jpg"
HEIGHT=25 WIDTH=100 BORDER=0></A>
<A HREF="javascript:document.forms[0].reset()"><IMG SRC="reset.jpg"
HEIGHT=25 WIDTH=100 BORDER=0></A>
</BODY>
</HTML>
```

Related Items: `onReset=` event handler; `reset` object.

submit()

Returns: Nothing.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

The most common way to send a form's data to a server's CGI program for processing is to have a user click a Submit button. The standard HTML Submit button is designed to send data from all elements of a form according to the specifications listed in the `<FORM>` definition's attributes. But if you want to submit a form's data to a server automatically for a user, or want to use a graphical button for submission, you can accomplish the submission with the `form.submit()` method.

Invoking this method is almost the same as a user clicking a form's Submit button (except that the `onSubmit=` event handler is not triggered in Navigator). Therefore, you may have an image on your page that is a graphical submission button. If that image is associated with a link object, you can capture a mouse click on that image and trigger a function whose content includes a call to a form's `submit()` method (see Listing 21-3).

In a multiple-form HTML document, however, you must be sure to reference the proper form, either by name or according to its position in a `document.forms[]` array. Always make sure that the reference you specify in your script points to the desired form before submitting any data to a server.

As a security and privacy precaution for people visiting your site, JavaScript ignores all `submit()` methods whose associated form action is set to a `mailto:` URL. Many Web page designers would love to have secret e-mail addresses captured from visitors. Because such a capture could be considered an invasion of privacy, the power has been disabled since Navigator 2.02. You can, however, still use an explicit Submit button object to mail a form to you from Navigator browsers only (see "E-mailing forms" earlier).

Because the `form.submit()` method does not trigger the form's `onSubmit=` event handler, you must perform any presubmission processing and forms validation in the same script that ends with the `form.submit()` statement. You also do not want to interrupt the submission process after the script invokes the

`form.submit()` method. Script statements after one invoking `form.submit()` — especially those that navigate to other pages or attempt a second submission — will cause the first submission to cancel itself.

Example

Consult Listing 21-3 for an example of using the `submit()` method from outside of a form.

Related Items: `onSubmit=` event handler.

Event handlers

onReset=

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔			✔

Immediately before a Reset button returns a form to its default settings, JavaScript sends a reset event to the form. By including an `onReset=` event handler in the form definition, you can trap that event before the reset takes place.

A friendly way of using this feature is to provide a safety net for a user who accidentally clicks on the Reset button after filling out a form. The event handler can run a function that asks the user to confirm the action.

The `onReset=` event handler employs a technique that started surfacing with Navigator 3: The event handler must evaluate to `return true` for the event to continue to the browser. This may remind you of the way `onMouseOver=` and `onMouseOut=` event handlers work for links and image areas. This requirement is far more useful here because your function can control whether the reset operation ultimately proceeds to conclusion.

Example

Listing 21-4 demonstrates one way to prevent accidental form resets or submissions. Using standard Reset and Submit buttons as interface elements, the `<FORM>` object definition includes both event handlers. Each event handler calls its own function that offers a choice for users. Notice how each event handler includes the word `return` and takes advantage of the Boolean values that come back from the `confirm()` method dialog boxes in both functions.

Listing 21-4: The onReset= and onSubmit= Event Handlers

```
<HTML>
<HEAD>
<TITLE>Submit and Reset Confirmation</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function allowReset() {
        return window.confirm("Go ahead and clear the form?")
```

```
}
function allowSend() {
       return window.confirm("Go ahead and mail this info?")
}
</SCRIPT>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="mailto:trash3@dannyg.com" onReset="return
allowReset()" onSubmit="return allowSend()">
Enter your first name:<INPUT TYPE="text" NAME="firstName"><P>
Enter your last name:<INPUT TYPE="text" NAME="lastName"><P>
Enter your address:<INPUT TYPE="text" NAME="address"><P>
Enter your city:<INPUT TYPE="text" NAME="city"><P>
<INPUT TYPE="radio" NAME="gender" CHECKED>Male
<INPUT TYPE="radio" NAME="gender">Female <P>
<INPUT TYPE="checkbox" NAME="retired">I am retired<P>
<INPUT TYPE="reset">
<INPUT TYPE="submit">
</FORM>
</BODY>
</HTML>
```

onSubmit=

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

No matter how a form's data is actually submitted (by a user clicking a Submit button or by a script invoking the `form.submit()` method), you may want your JavaScript-enabled HTML document to perform some data validation on the user input, especially with text fields, before the submission heads for the server. You have the option of doing such validation while the user enters data (see Chapter 37) or in batch mode before sending the data to the server — or both. The place to trigger this last-ditch data validation is the form's `onSubmit=` event handler.

When you define an `onSubmit=` handler as an attribute of a `<FORM>` definition, JavaScript sends the submit event to the form just before it dashes off the data to the server. Therefore, any script or function that is the parameter of the `onSubmit=` attribute executes before the data is actually submitted. Note that this event handler fires only in response to a genuine Submit-style button, and not from a `form.submit()` method.

Any code executed for the `onSubmit=` event handler must evaluate to an expression consisting of the word `return` plus a Boolean value. If the Boolean value is true, the submission executes as usual; if the value is false, no submission is made. Therefore, if your script performs some validation prior to submitting data, make sure that the event handler calls that validation function as part of a `return` statement, as shown in Listing 21-4.

Even after your `onSubmit=` event handler traps a submission, JavaScript's security mechanism can present additional alerts to the user, depending on the server location of the HTML document and the destination of the submission.

Example

See Listing 21-4 for an example of trapping a submission via the `onSubmit=` event handler.

✦ ✦ ✦