Text-Related Objects

CHAPTER 22

In This Chapter

Capturing and modifying text field contents

Triggering action by entering text

Capturing individual keystroke events

The Netscape document object model for forms includes four text-related input objects — text, password, textarea, and hidden. All four of these objects are used for entry, display, or temporary storage of text data. While all of these objects can have text placed in them by default as the page loads, scripts can also modify the contents of these objects. Importantly, all but the hidden objects retain their user- or script-modified content during a soft reload (for example, clicking the Reload button), except in Internet Explorer 3. Hidden objects revert to their default values on all reloads in all browsers.

A more obvious difference between the hidden object and the rest is that its invisibility removes it from the realm of user events and actions. Therefore, only the text-style object properties apply to the hidden object.

The persistence of text and textarea object data through reloads (and window resizes) makes these objects prime targets for off-screen storage of data that might otherwise be stored temporarily in a cookie. If you create a frame with no size (for example, you set the `COLS` or `ROWS` values of a `<FRAMESET>` tag to let all visible frames occupy 100% of the space and assign the rest — * — to the hidden frame), you can populate it with fields that act as shopping cart information or other data holders. Therefore, if users have cookies turned off or don't usually respond affirmatively to cookie requests, your application can still make use of temporary client storage. The field contents may survive unloading of the page, but whether this happens and for how many navigations away from the page the contents last depends on the visitor's cache settings (or if the browser is Internet Explorer 3, in which case no values preserve the unloading of a document). If the user quits Navigator, the field entry is lost.

Text Object

Properties	*Methods*	*Event Handlers*
defaultValue	blur()	onBlur=
form	focus()	onChange=
name	handleEvent()	onFocus=
type	select()	onKeyDown=
value		onKeyPress=
		onKeyUp=
		onSelect=

Syntax

Creating a text object:

```
<FORM>
<INPUT
        TYPE="text"
        NAME="fieldName"
        [VALUE="contents"]
        [SIZE="characterCount"]
        [MAXLENGTH="maxCharactersAllowed"]
        [onBlur="handlerTextOrFunction"]
        [onChange="handlerTextOrFunction"]
        [onFocus="handlerTextOrFunction"]
        [onKeyDown="handlerTextOrFunction"]
        [onKeyPress="handlerTextOrFunction"]
        [onKeyUp="handlerTextOrFunction"]
        [onSelect="handlerTextOrFunction"]>
</FORM>
```

Accessing text object properties or methods:

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

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

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

About this object

The text object is the primary medium for capturing user-entered text. Browsers tend to display entered text in a monospaced font (usually Courier or a derivative), so you can easily specify the width (SIZE) of a field based on the anticipated number of characters that a user may put into the field. The font is a fixed size and always is left-aligned in the field. If your design requires multiple lines of text, use the textarea object that follows.

Due to the limitations scripts have in updating information on an existing HTML page (without assembling and rewriting an entire page in JavaScript), a common practice is to use text objects to display results of a script calculation or other processing. Such fields may stand alone on a page or be part of a table.

Unfortunately, these fields are not write-protected, so it's easy to understand how a novice user may become confused when he or she causes the text pointer or selection to activate a field used exclusively for output, simply by tabbing through a page. Of course, if such a script does not have an event handler attached to it, no harm can come in manually changing the contents of a results field — but the user may get mighty confused. A better choice for the scripter may be to attach an onChange= event handler to output fields so that if a user attempts to change the contents of a field, the calculation runs again or the previous result (if stored in the script as a global variable) is automatically reinserted when the user tabs or clicks out of that changed field.

Text object methods and event handlers use terminology that may be known to Windows users but not to Macintosh users. A field is said to have *focus* when the user clicks on or tabs into the field. When a field has focus, either the text insertion pointer flashes, or any text in the field may be selected. Only one text object on a page can have focus at a time. The inverse user action — clicking or tabbing away from a text object — is called a *blur*. Clicking another object, whether it is another field or a button of any kind, causes a field that currently has focus to blur.

If you don't want the contents of a field to be changed by the user, you can force the field to lose focus when a user tabs to or clicks on a field. Use the following event handler in such a field:

```
onFocus="this.blur()"
```

Focus and blur also interact with other possible user actions to a text object: selecting and changing. *Selecting* occurs when the user clicks and drags across any text in the field; *changing* occurs when the user makes any alteration to the content of the field and then either tabs or clicks away from that field.

When you design event handlers for fields, be aware that a user's interaction with a field may trigger more than one event with a single action. For instance, clicking a field to select text may trigger both a focus and select event. If you have conflicting actions in the onFocus= and onSelect= event handlers, your scripts can do some weird things to the user's experience with your page. Displaying alert dialog boxes, for instance, also triggers blur events, so a field that has both an onSelect= handler (which displays the alert) and an onBlur= handler will get a nasty interaction from the two.

As a result, you should be very judicious with the number of event handlers you specify in any text object definition. If possible, pick one user action you want to use to initiate some JavaScript code execution and deploy it consistently on the

page. Not all fields require event handlers — only those you want to perform some action as the result of user activity in that field.

Many newcomers also become confused by the behavior of the change event. To prevent this event from being sent to the field for every character the user types, any change to a field is determined only *after* the field loses focus by the user's clicking or tabbing away from it. At that point, instead of a blur event being sent to the field, only a change event is sent, triggering an `onChange=` event handler if one is defined for the field. This extra burden of having to click or tab away from a field may entice you to shift any `onChange=` event handler tasks to a separate button that the user must click to initiate action on the field contents.

Many scripters dream of possibilities for text objects that, alas, are simply not possible in the current implementation, including the capabilities to dim the text box to prevent entry, select a portion of the text, and change font and color attributes.

Some of these items may be scriptable in the future, but we will have to wait and see. In the meantime, some new functionality was added to the object in Navigator 4. The biggest news was the addition of keystroke events, making it possible to monitor keystrokes before they register their characters in the field.

Text objects (including the related textarea object) have one unique behavior that can be very important to some document and script designs. Even if a default value is specified for the content of a field (in the `VALUE` attribute), any text entered into a field by a user or script persists in that field as long as the document is cached in the browser's memory cache (but Internet Explorer 3 has no such persistence). Therefore, if users of your page enter values into some fields, or your scripts display results in a field, all that data will be there later, even if the user performs a soft reload of the page or navigates to dozens of other Web pages or sites. Navigating back via the Go or Bookmarks menu entries causes the browser to retrieve the cached version (with its field entries). To force the page to appear with its default text object values, use the Open Location or Open File selections in the File menu, or script the `location.reload()` method. These actions cause the browser to load the desired page from scratch, regardless of the content of the cache. When you quit and relaunch the browser, the first time it goes to the desired page, the browser loads the page from scratch — with its default values.

This level of persistence is not as reliable as the `document.cookie` property because a user can reopen a URL at any time, thus erasing whatever was temporarily stored in a text or textarea object. Still, this method of temporary data storage may suffice for some designs. Unfortunately, you cannot completely hide a text object in case the data you want to store is for use only by your scripts. The `TYPE="hidden"` form element is not an alternative here because script-induced changes to its value do not persist across soft reloads.

If you prefer to use a text or textarea object as a storage medium but don't want users to see it, design the page to display in a nonresizable frame of height or width zero. Use proper frame references to store or retrieve values from the fields. Carrying out this task requires a great deal of work. The `document.cookie` may not seem so complicated after all that.

To extract the current content of a text object, summon the `document.formName. fieldName.value` property. Once you have the string value, you can use JavaScript's string object methods to parse or otherwise massage that

text as needed for your script. If the field entry is a number and you need to pass that value to methods requiring numbers, you have to convert the text to a number with the help of the `parseInt()` or `parseFloat()` global functions.

Properties

`defaultValue`

Value: String **Gettable:** Yes **Settable:** No

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

Though your users and your scripts are free to muck with the contents of a text object by assigning strings to the `value` property, you can always extract (and thus restore, if necessary) the string assigned to the text object in its `<INPUT>` definition. The `defaultValue` property yields the string parameter of the `VALUE=` attribute.

Example

Listing 22-1 has a simple form with a single field that has a default value set in its tag. A function (`resetField()`) restores the contents of the page's lone field to the value assigned to it in the `<INPUT>` definition. For a single-field page such as this, defining a `TYPE="reset"` button or calling `form.reset()` works the same way because such buttons reestablish default values of all elements of a form. But if you want to reset only a subset of fields in a form, follow the example button and function in Listing 22-1.

Listing 22-1: Resetting a Text Object to Default Value

```
<HTML>
<HEAD>
<TITLE>Text Object DefaultValue</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function upperMe(field) {
        field.value = field.value.toUpperCase()
}
function resetField(form) {
        form.converter.value = form.converter.defaultValue
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter lowercase letters for conversion to uppercase: <INPUT TYPE="text"
NAME="converter" VALUE="sample" onChange="upperMe(this)">
<INPUT TYPE="button" VALUE="Reset Field"
```

(continued)

Listing 22-1 *(continued)*

```
onClick="resetField(this.form)">
</FORM>
</BODY>
</HTML>
```

Related Items: `value` property.

name

Value: String **Gettable:** Yes **Settable:** No

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

Text object names are important for two reasons. First, if your HTML page is used to submit information to CGI scripts, the input device passes the name of the text object along with the data to help the server program identify the data being supplied by the form. Second, you can use a text object's name in its reference within JavaScript coding. If you assign distinctive, meaningful names to your fields, these names will help you read and debug your JavaScript listings (and will help others follow your scripting tactics).

Be as descriptive about your text object names as you can. Borrowing text from the field's on-page label also helps you mentally map a scripted reference to a physical field on the page. Like all JavaScript object names, text object names must begin with a letter and be followed by any number of letters or numbers. Avoid punctuation symbols with the exception of the very safe underscore character.

Although I urge you to use distinctive names for all objects you define in a document, you can make a case for assigning the same name to a series of interrelated fields — and JavaScript is ready to help. Within a single form, any reused name for the same object type is placed in an indexed array for that name. For example, if you define three fields with the name `entry`, the following statements retrieve the `value` property for each field:

```
data = document.forms[0].entry[0].value
data = document.forms[0].entry[1].value
data = document.forms[0].entry[2].value
```

This construction may be useful if you want to cycle through all of a form's related fields to determine which ones are blank. Elsewhere, your script probably needs to know what kind of information each field is supposed to receive, so it can process the data intelligently. I don't often recommend reusing object names, but you should be aware of how JavaScript handles them in case you need this construction. Unfortunately, Internet Explorer 3 does not turn like-named text input objects into arrays. See "Form Element Arrays" in Chapter 21 for more details.

Example

Consult Listing 22-2, where I use the text object's name, `convertor`, as part of the reference when assigning a value to the field. To extract the name of a text object, you can use the property reference. Therefore, assuming that your script doesn't know the name of the first object in the first form of a document, the statement is

```
objectName = document.forms[0].elements[0].name
```

Related Items: `form.elements` property; all other form element objects' name property.

type

Value: String **Gettable:** Yes **Settable:** No

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

Use the `type` property to help you identify a text object from an unknown group of form elements. For a text input object, the value is `text`.

Related Items: `form.elements` property.

value

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

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

A text object's `value` property is the two-way gateway to the content of the field. A reference to an object's `value` property returns the string currently showing in the field. Note that all values coming from a text object are string values. If your field prompts a user to enter a number, your script may have to perform data conversion to the number-as-string value ("42" instead of plain old 42) before a user can perform any math operations on it. JavaScript tries to be as automatic about this data conversion as possible and follows some rules about it (Chapter 27). If you see an error message that says a value is not a number (for a math operation), the value is still a string.

Your script places text of its own into a field for display to the user by assigning a string to the `value` property of a text object. Use the simple assignment operator. For example

```
document.forms[0].ZIP.value = "90210"
```

JavaScript is more forgiving about data types when assigning values to a text object. JavaScript automatically converts any value to a string on its way to a text object display. Even Boolean values get converted to their string equivalents "true" or "false." Scripts can place numeric values into fields without a hitch. But remember that if a script later retrieves these values from the text object, they will come back as strings.

Storing arrays in a field does require special processing. You need to use the `array.join()` method to convert an array into a string. Each array entry is delimited by a character you establish in the `array.join()` method. Later you can use the `string.split()` method to turn this delimited string into an array.

Example

Important: Listings 22-2 and 22-3 feature a form with only one text input object. The rules of HTML forms say that such a form will submit itself if the user presses the Enter key whenever the field has focus. Such a submission to a form whose action is undefined causes the page to perform a hard reload. Moreover, the form's `onSubmit=` event handler is not triggered with this kind of submission, so you cannot set an event handler to prevent the submission. To see the results of either listing, enter a value and then either press the Tab key or click anywhere outside of the field.

As a demonstration of how to retrieve and assign values to a text object, Listing 22-2 shows how the action in an `onChange=` event handler is triggered. Enter any lowercase letters into the field and click out of the field. I pass a reference to the entire form object as a parameter to the event handler. The function extracts the value, converts it to uppercase (using one of the JavaScript string object methods), and assigns it back to the same field in that form.

Listing 22-2: **Getting and Setting a Text Object's Value**

```
<HTML>
<HEAD>
<TITLE>Text Object Value</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function upperMe(form) {
        inputStr = form.converter.value
        form.converter.value = inputStr.toUpperCase()
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter lowercase letters for conversion to uppercase: <INPUT TYPE="text"
NAME="converter" VALUE="sample" onChange="upperMe(this.form)">
</FORM>
</BODY>
</HTML>
```

I also show two other ways to accomplish the same task, each one more efficient than the previous example. Both utilize the shortcut object reference to get at the heart of the text object. Listing 22-3 passes the field object — contained in the `this` reference — to the function handler. Because that field object contains a complete reference to it (out of sight, but there just the same), you can access the `value` property of that object and assign a string to that object's `value` property in a simple assignment statement.

Listing 22-3: **Passing a Text Object (as this) to the Function**

```
<HTML>
<HEAD>
<TITLE>Text Object Value</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function upperMe(field) {
        field.value = field.value.toUpperCase()
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter lowercase letters for conversion to uppercase: <INPUT TYPE="text"
NAME="converter" VALUE="sample" onChange="upperMe(this)">
</FORM>
</BODY>
</HTML>
```

A more efficient way is to deal with the field values directly in an embedded event handler — instead of calling an external function (which might still be useful if other objects need this functionality). With the function removed from the document, the event handler attribute of the `<INPUT>` definition changes to do all the work:

```
<INPUT TYPE="text" NAME="converter" VALUE="sample"
 onChange="this.value = this.value.toUpperCase()">
```

The right-hand side of the assignment expression extracts the current contents of the field and (with the help of the `toUpperCase()` method of the string object) converts the original string to all uppercase letters. The result of this operation is assigned to the `value` property of the field.

The application of the `this` keyword in the previous examples may be confusing at first, but these examples represent the range of ways in which you can use such references effectively. Using `this` by itself as a parameter to an object's event handler refers only to that single object — a text object in Listing 22-3. If you want to pass along a broader scope of objects that contain the current object, use the `this` keyword along with the outer object layer you want. In Listing 22-2, I sent the entire form along by specifying `this.form` — meaning the form that contains "this" object, which is being defined in the line of HTML code.

At the other end of the scale, you can use similar-looking syntax to specify a particular property of the "this" object. Thus, in the last example, I zeroed in on just the `value` property of the current object being defined — `this.value`. Although the formats of `this.form` and `this.value` appear the same, they encompass entirely different ends of the range of focus — simply by virtue of the meaning of the keywords to the right of the period. As long as you know that a form is an object of larger scope than the currently defined object, you will know that `this.form` includes an entire form object (and all its elements); conversely, if you know that a text object has a property named "value," you will know that a reference to `this.value` focuses only on the `value` property of the currently defined object. This is why it's so important for JavaScript authors to be completely familiar with both the object hierarchy and the range of property and method names for those objects.

Related Items: `form.defaultValue` property.

Methods

blur()

Returns: Nothing.

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

Just as a camera lens blurs when it goes out of focus, a text object blurs when it loses focus — when someone clicks or tabs out of the field. Under script control, `blur()` deselects whatever may be selected in the field, and the text insertion pointer leaves the field. The pointer does not proceed to the next field in tabbing order, as it does if you perform a blur by tabbing out of the field manually.

Example

```
document.forms[0].vanishText.blur()
```

Related Items: `focus()` method; `onBlur=` event handler.

focus()

Returns: Nothing.

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

For a text object, having focus means that the text insertion pointer is flashing in that text object's field (it means something different for buttons in a Windows environment). Giving a field focus is like opening it up for human editing.

Setting the focus of a field containing text does not let you place the cursor at any specified location in the field. The cursor usually appears at the beginning of the text. To prepare a field for entry to remove the existing text, use both the `focus()` and `select()` methods.

The `focus()` method does not work reliably in Navigator 3 for UNIX clients. While the `select()` method selects the text in the designated field, focus is not handed to the field.

Example

See Listing 22-4 for an example of an application of the `focus()` method in concert with the `select()` method.

Related Items: `select()` method; `onFocus=` event handler.

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.

select()

Returns: Nothing.

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

Selecting a field under script control means selecting all text within the text object. A typical application is one in which an entry validation script detects a mistake on the part of the user. After alerting the user to the mistake (via a `window.alert()` dialog box), the script finishes its task by selecting the text of the field in question. Not only does this action draw the user's eye to the field needing attention (especially important if the validation code is checking multiple fields),

but it also keeps the old text there for the user to examine for potential problems. With the text selected, the next key the user presses erases the former entry.

Trying to select a text object's contents with a click of a button is problematic. One problem is that a click of the button brings the document's focus to the button, which disrupts the selection process. For more ensured selection, the script should invoke both the `focus()` and the `select()` methods for the field, in that order. No penalty exists for issuing both methods, and the extra insurance of the second method provides a more consistent user experience with the page.

Selecting a text object via script does *not* trigger the same `onSelect=` event handler for that object as the one that triggers if a user manually selects text in the field. Therefore, no event handler script is executed when a user invokes the `select()` method.

Example

A click of the Verify button in Listing 22-4 sends the text object's contents for validation as all numbers. If the validation returns false, the script preselects the field entry for the user. To make sure the selection takes place, you first set the document's focus to the field and then select its contents. If the focus was on the field immediately before clicking the button, the selection may work without requiring you to set the focus. But you cannot be sure of what the user will do between entering text and clicking the button. Try commenting out (//) the `form.numeric.focus()` statement, and then reload the document and see how well the selection works by itself under a variety of circumstances. You will find that setting the focus is a surefire method.

Listing 22-4: **Selecting a Field**

```
<HTML>
<HEAD>
<TITLE>Text Object Select/Focus</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// general purpose function to see if a suspected numeric input is a
number
function isNumber(inputStr) {
        for (var i = 0; i < inputStr.length; i++) {
           var oneChar = inputStr.charAt(i)
           if (oneChar < "0" || oneChar > "9") {
              alert("Please make sure entries are numbers only.")
              return false
           }
        }
        return true
}
function checkIt(form) {
        inputStr = form.numeric.value
        if (isNumber(inputStr)) {
           // statements if true
        } else {
           form.numeric.focus()
           form.numeric.select()
```

```
        }
}

</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter any positive integer: <INPUT TYPE="text" NAME="numeric"><P>
<INPUT TYPE="button" VALUE="Verify" onClick="checkIt(this.form)">
</FORM>
</BODY>
</HTML>
```

Related Items: `focus()` method; `onSelect=` event handler.

Event handlers

onBlur=

onFocus=

onSelect=

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

All three of these event handlers should be used only after you have a firm understanding of the interrelationships of the events that reach text objects. You must use extreme care and conduct lots of user testing before including more than one of these three event handlers in a text object. Because some events cannot occur without triggering others either immediately before or after (for example, an `onFocus=` occurs immediately before an `onSelect=` if the field did not have focus before), whatever actions you script for these events should be as distinct as possible to avoid interference or overlap.

In particular, be careful about displaying modal dialog boxes (for example, `window.alert()` dialog boxes) in response to the `onFocus=` event handler. Because the text field loses focus when the alert displays and then regains focus when the alert is closed, you can get yourself into a loop that is difficult to break out of. If you should get trapped in this manner, try the keyboard shortcut for reloading the page (Ctrl+R or ⌘-R) repeatedly as you keep closing the dialog window.

A question arises about whether data-entry validation should be triggered by the `onBlur=` or `onChange=` event handler. An `onBlur=` validation cannot be fooled, whereas an `onChange=` one can be (the user simply doesn't change the bad entry

as he or she tabs out of the field). What I don't like about the `onBlur=` way is it can cause a frustrating experience for a user who wants to tab through a field now and come back to it later (assuming your validation requires data be entered into the field before submission). As in Chapter 37's discussion about form data validation, I recommend using `onChange=` event handlers to trigger immediate data checking and then using another last-minute check in a function called by the form's `onSubmit=` event handler.

Example

To demonstrate one of these event handlers, Listing 22-5 shows how you might use the window's status bar as a prompt message area when a user activates any field of a form. When the user tabs to or clicks on a field, the prompt message associated with that field appears in the status bar.

Listing 22-5: The onFocus= Event Handler

```
<HTML>
<HEAD>
<TITLE>Elements Array</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function prompt(msg) {
       window.status = "Please enter your " + msg + "."
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter your first name:<INPUT TYPE="text" NAME="firstName"
onFocus="prompt('first name')"><P>
Enter your last name:<INPUT TYPE="text" NAME="lastName"
onFocus="prompt('last name')"><P>
Enter your address:<INPUT TYPE="text" NAME="address"
onFocus="prompt('address')"><P>
Enter your city:<INPUT TYPE="text" NAME="city"
onFocus="prompt('city')"><P>
</FORM>
</BODY>
</HTML>
```

onChange=

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

Of all the event handlers for a text object, you will probably use the onChange= handler the most in your forms (see Listing 22-6). This event is the one I prefer for triggering the validation of whatever entry the user just typed in the field. The potential hazard of trying to do only a batch-mode data validation of all entries before submitting an entire form is that the user's mental focus is away from the entry of a given field as well. When you immediately validate an entry, the user is already thinking about the information category in question. See Chapter 37 for more about data-entry validation.

In Navigator 4, if you have both onChange= and any keyboard event handlers defined for the same text field tag, the onChange= event handlers are ignored. This is not true for Internet Explorer 4, where all events fire.

Example

Whenever a user makes a change to the text in a field in Listing 22-6 and then either tabs or clicks out of the field, the change event is sent to that field, triggering the onChange= event handler.

Because the form in Listing 22-6 has only one field, the same warning about the effect of pressing the Enter key applies here as to Listings 22-1 and 22-2 earlier in this chapter. Press Tab or click outside the field to trigger the onChange= event handler without submitting the form.

Listing 22-6: Data Validation via an onChange= Event Handler

```
<HTML>
<HEAD>
<TITLE>Text Object Select/Focus</TITLE>
<SCRIPT LANGUAGE="JavaScript">
// general purpose function to see if a suspected numeric input is a
number
function isNumber(inputStr) {
        for (var i = 0; i < inputStr.length; i++) {
           var oneChar = inputStr.substring(i, i + 1)
           if (oneChar < "0" || oneChar > "9") {
              alert("Please make sure entries are numbers only.")
              return false
           }
        }
        return true
}
function checkIt(form) {
        inputStr = form.numeric.value
```

(continued)

Listing 22-6 *(continued)*

```
        if (isNumber(inputStr)) {
           // statements if true
        } else {
           form.numeric.focus()
           form.numeric.select()
        }
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter any positive integer: <INPUT TYPE="text" NAME="numeric"
onChange="checkIt(this.form)"><P>
</FORM>
</BODY>
</HTML>
```

onKeyDown=

onKeyPress=

onKeyUp=

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

New text field events for Navigator 4 and Internet Explorer 4 let your scripts capture user activity from the keyboard while the field has focus. The keyDown event occurs the instant the user presses the key far enough to make contact; the keyUp event occurs when electrical contact with the key breaks; and a keyPress event occurs after the keyUp event, signaling the completion of a matched pair of keyDown and keyUp events. No event exists to let you know when a key starts repeating itself.

Whichever event(s) you trap for, you can also extract the ASCII (or UniCode in Internet Explorer 4) value of the key. All of this event activity occurs *before* the typed character ever appears in the field. This allows your scripts a chance to filter out unwanted characters or even convert characters to different ones (for example, turning all letters into their uppercase equivalents for storage in a database).

Navigator and Internet Explorer have different ways of revealing the typed character to a script. But, as you can see in Listing 22-7, they are close enough that you can accommodate both browsers in simple key capture event handlers.

For an event handler to inspect a key character, the event handler must pass the event object that automatically accompanies such an event. The way you do this is to pass the keyword `event` as one of the parameters to the function called by the event handler:

```
onKeyPress="filterKeys(event)"
```

Notice that the `event` keyword is not a string, but rather a reference to the event object generated by the press of the key. You can also assign this kind of event handler in a script statement in the following manner:

```
document.formName.fieldName.onkeypress = filterKeys
```

In this latter format, the event object is automatically passed to the function whose reference is assigned to the event for the field object. Be aware, however, that assignments like the previous one must be lower in the document than the object for which the event hander is being assigned. Otherwise, the field won't yet be defined in the browser's object model for the document, and the statement generates an error during page loading.

You can control whether the keyboard action ever reaches the display area of the text object. Like many of the objects in the document model, if an event handler evaluates to `return false`, the event goes no further than the function invoked by the event handler; but if the handler evaluates to `return true`, then the natural behavior of the event carries on.

In Navigator 4, if you have both `onChange=` and any keyboard event handlers defined for the same text field tag, the `onChange=` event handlers are ignored. This is not true for Internet Explorer 4, where all events fire.

Example

Listing 22-7 is a simple script that inspects the key presses in a field expecting only numbers. By checking the value of each key before it reaches the value of the field, the event handler function can alert the user about the correct kind of characters to enter. The `checkIt()` function receives the event object passed by the event handler. To work with both Navigator and Internet Explorer, the function does one differentiation about how to extract the value of the typed character, depending on the browser version. After that, the value is in the `charCode` variable, and the rest of the function works the same for both platforms.

Notice the construction of the event handler such that it evaluates to `return true` or `return false`, depending on the results of the function. When the handler evaluates to `return false`, the typed character does not appear in the field, since in this case, I don't want nonnumbers to show themselves in the field.

Listing 22-7: **Keyboard Filtering for Numbers Only**

```
<HTML>
<HEAD>
<TITLE>Keyboard Capture</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function checkIt(e) {
        var charCode = (navigator.appName == "Netscape") ? e.which :
e.keyCode
        status = charCode // see ASCII character value!
        if (charCode > 31 && (charCode < 48 || charCode > 57)) {
            alert("Please make sure entries are numbers only.")
            return false
        }
        return true
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter any positive integer: <INPUT TYPE="text" NAME="numeric"
onKeyPress="return checkIt(event)"><P>
</FORM>
</BODY>
</HTML>
```

Learn more about handling events for Navigator 4 and cross-compatibility in Chapter 39.

Related Items: event object.

Password Object

Properties	*Methods*	*Event Handlers*
defaultValue	blur()	onBlur=
form	focus()	onChange=
name	handleEvent()	onFocus=
type	select()	onKeyDown=
value		onKeyPress=
		onKeyUp=
		onSelect=

Syntax

Creating a password object:

```
<FORM>
<INPUT
        TYPE="password"
        NAME="fieldName"
        [VALUE="contents"]
        [SIZE="characterCount"]
        [MAXLENGTH="maxCharactersAllowed"]
        [onBlur="handlerTextOrFunction"]
        [onChange="handlerTextOrFunction"]
        [onFocus="handlerTextOrFunction"]
        [onKeyDown="handlerTextOrFunction"]
        [onKeyPress="handlerTextOrFunction"]
        [onKeyUp="handlerTextOrFunction"]
        [onSelect="handlerTextOrFunction"]>
</FORM>
```

About this object

A password-style field looks like a text object, but when the user types something into the field, only asterisks or bullets (depending on your operating system) appear in the field. For the sake of security, any password exchanges should be handled by a server-side CGI program.

Many properties of the password object were blocked from scripted access until Navigator 3. Scripts now can treat a password object exactly like a text object. This might lead a scripter to capture a user's Web site password for storage in the `document.cookie` of the client machine. A password object `value` property is returned in plain language, so such a captured password would be stored in the cookie file the same way. Because a client machine's cookie file can be examined on the local computer (perhaps by a snoop during lunch hour), plain-language storage of passwords is a potential security risk. Instead, develop a scripted encryption algorithm for your page for reading and writing the password in the cookie. Most password-protected sites, however, usually have a CGI program on the server encrypt the password prior to sending it back to the cookie.

See the text object discussion for the behavior of password object's properties, methods, and event handlers. The `type` property for this object returns `"password"`.

Textarea Object

Properties	*Methods*	*Event Handlers*
defaultValue	blur()	onBlur=
form	focus()	onChange=
name	handleEvent()	onFocus=

(continued)

Properties	Methods	Event Handlers
type	select()	onKeyDown=
value		onKeyPress=
		onKeyUp=
		onSelect=

Syntax

Creating a textarea object:

```
<TEXTAREA
        NAME="fieldName"
        [ROWS="rowCount"]
        [COLS="columnCount"]
        [WRAP="off" | "virtual" | "physical"]
        [onBlur="handlerTextOrFunction"]
        [onChange="handlerTextOrFunction"]
        [onFocus="handlerTextOrFunction"]
        [onKeyDown="handlerTextOrFunction"]
        [onKeyPress="handlerTextOrFunction"]
        [onKeyUp="handlerTextOrFunction"]
        [onSelect="handlerTextOrFunction"]>
        defaultText
</TEXTAREA>
</FORM>
```

Accessing textarea object properties or methods:

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

About this object

Although not in the same syntax family as other <INPUT> elements of a form, a textarea object is indeed a form input element. Any definition for a textarea object must be written within the confines of a <FORM>...</FORM> tag pair.

A textarea object closely resembles a text object, except for attributes that define its physical appearance on the page. Because the intended use of a textarea object is for multiple-line text input, the attributes include specifications for height

(number of rows) and width (number of columns in the monospaced font). No matter what size you specify, the browser displays a textarea with horizontal and vertical scrollbars. Neither the user nor a script can resize the field, nor does text wrap within the visible rectangle of the field unless you set the `WRAP` attribute to `"virtual"` or `"physical"`. Instead, the user is encouraged to press Enter to make manual carriage returns. If the user fails to do so, the text scrolls for a significant distance horizontally (the horizontal scrollbar appears when wrapping has the default off setting). This field is, indeed, a primitive text field by GUI computing standards.

All properties, methods, and event handlers of text objects apply to the textarea object. They all behave exactly the same way (except, of course, for the `type` property, which is `"textarea"`). Therefore, refer to the previous listings for the text object for scripting details.

Carriage returns inside textareas

The three classes of operating systems supported by Netscape Navigator — Windows, Macintosh, and UNIX — do not agree about what constitutes a carriage return character in a text string. This discrepancy carries over to the textarea object and its contents on these platforms.

When a user enters text and uses Enter/Return on the keyboard, one or more unseen characters are inserted into the string. In the parlance of JavaScript's literal string characters, the carriage return consists of some combination of the new line (`\n`) and return (`\r`) character. The following list shows the characters inserted into the string for each operating system category.

Operating System	*Character String*
Windows	\r\n
Macintosh	\r
Unix	\n

This tidbit is valuable if you need to remove carriage returns from a textarea for processing in a CGI or local script. The problem is that you obviously need to perform platform-specific operations on each. For the situation in which you must preserve the carriage return locations, but your server-side database cannot accept the carriage return values, I suggest you use the `string.escape()` method to URL-encode the string. The return character is converted to `%0D` and the newline character is converted to `%0A`. Of course these characters occupy extra character spaces in your database, so these additions must be accounted for in your database design.

As far as writing carriage returns into textareas, the situation is a bit easier. From Navigator 3 onward, if you specify any one of the combinations in the preceding list, all platforms know how to automatically convert the data to the form native to the operating system. Therefore, you can set the value of a textarea object to `"1\r\n2\r\n3"` in all platforms, and a columnar list of the numbers 1, 2, and 3 will appear in those fields. Or, if you URL-encoded the text for saving to a

database, you can unescape that character string before setting the textarea value, and no matter what platform the visitor has, the carriage returns are rendered correctly. Upon reading those values again by script, you can see that the carriage returns are in the form of the platform (as shown in the previous table).

Hidden Object

Properties	*Methods*	*Event Handlers*
defaultValue	(None)	(None)
form		
name		
type		
value		

Syntax

Creating a hidden object:

```
<FORM>
<INPUT
        TYPE="hidden"
        NAME="fieldName"
        [VALUE="contents"]>
</FORM>
```

Accessing hidden object properties:

```
[window.] document.formName.fieldName.property
[window.] document.formName.elements[index].property

[window.] document.forms[index].fieldName.property

[window.] document.forms[index].elements[index].property
```

About this object

A hidden object is a simple string holder within a form object whose contents are not visible to the user of your Web page. With no methods or event handlers, the hidden object's value to your scripting is as a delivery vehicle for strings that your scripts need for reference values or other hard-wired data.

The hidden object plays a vital role in applications that rely on CGI programs on the server. Very often, the server has data that it needs to convey to itself the next time the client makes a submission (for example, a user ID captured at the application's login page). A CGI program can generate an HTML page with the necessary data hidden from the user but located in a field transmitted to the server at submit time.

Along the same lines, a page for a server application may present a user-friendly interface that makes data entry easy for the user. But on the server end, the database or other application requires that the data be in a more esoteric format. A script located in the page generated for the user can perform the last minute assemblage of user-friendly data into database-friendly data in a hidden field. When the CGI program receives the request from the client, it passes along the hidden field value to the database.

I am not a fan of the hidden object for use on client-side-only JavaScript applications. If I want to deliver with my JavaScript-enabled pages some default data collections or values, I do so in JavaScript variables and arrays as part of the script.

Because scripted changes to the contents of a hidden field are fragile (for example, a soft reload erases the changes), the only place you should consider making such changes is in the same script that submits a form to a CGI program or in a function triggered by an `onSubmit=` event handler. In effect, you're just using the hidden fields as holding pens for the scripted data to be submitted. For more persistent storage, use the `document.cookie` property or genuine text fields in hidden frames, even if just for the duration of the visit to the page.

For information about the properties of the hidden object, consult the earlier listing for the text object.

✦ ✦ ✦