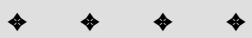
CHAPTER 19

Body Text Objects

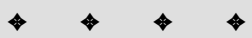


In This Chapter

Objects that display running body text in documents

Using the NN `Range` and IE `TextRange` objects

Scripting search and replace actions



A large number of HTML elements fall into a catchall category of elements whose purposes are slightly more targeted than contextual elements covered in Chapter 15. In this group are some very widely used elements, such as the H1 through H6 header elements, plus several elements that are not yet widely used because their full support may be lacking in even some of the most modern browsers. In this chapter, you find all sorts of text-related objects, excluding those objects that act as form controls (text boxes and such, which are covered in Chapter 25). For the most part, properties, methods, and event handlers of this chapter's objects are the generic ones covered in Chapter 15. Only those items that are unique to each object are covered in this chapter (as will be the case in all succeeding chapters).

Beyond the HTML element objects covered in this chapter, you also meet the `TextRange` object, first introduced in IE4, and the corresponding `Range` object from the W3C DOM implemented in NN6. This object is a very powerful one for scripters because it allows scripts to work very closely with body content — not in terms of, for example, the `innerText` or `nodeValue` properties of elements, but rather in terms of the text as it appears on the page in what users see as paragraphs, lists, and the like. The `TextRange` and `Range` objects essentially give your scripts cursor control over running body text for functions, such as cutting, copying, pasting, and applications that extend from those basic operations — search and replace, for instance. Bear in mind that everything you read in this chapter requires in the least the dynamic object models of IE4+ and NN6+; some items require IE5+. Unfortunately, the IE `TextRange` object is not implemented in IE5/Mac.

BLOCKQUOTE and Q Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`cite`		

Syntax

Accessing BLOCKQUOTE or Q element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/NN6) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About these objects

The BLOCKQUOTE element is a special-purpose text container. Browsers typically start the content on its own line in the body and indent on both the left and right margins approximately 40 pixels. An inline quotation can be encased inside a Q element, which does not force the quoted material to start on the next line.

From an object point of view, the only property that distinguishes these two objects from any other kind of contextual container is the `cite` property, which comes from the HTML 4.0 `CITE` attribute. This attribute simply provides a URL reference for the citation and does not act as an `SRC` or `HREF` attribute to load an external document.

Property

`cite`

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The cite property can contain a URL (as a string) that points to the source of the quotation in the BLOCKQUOTE or Q element. Future browsers may provide some automatic user interface link to the source document, but none of the browsers that support the cite property do anything special with this information.

Related Items: None.

BR Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
clear		

Syntax

Accessing BR element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/NN6) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About this object

The BR element forces a carriage return and line feed for rendered content on the page. This element does not provide the same kind of vertical spacing that goes between paragraphs in a series of P elements. Only one attribute (CLEAR) distinguishes this element from generic HTML elements and objects.

Property

clear

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The `clear` property defines how any text in an element following the BR element wraps around a floating element (for example, an image set to float along the right margin). While recent browsers expose this property, the attribute on which it is based is deprecated in the HTML 4.0 specification in an effort to encourage the use of the `clear` style sheet attribute for a BR element.

Values for the `clear` property can be one of the following strings: `all`, `left`, or `right`.

Related Items: `clear` stylesheet property.

FONT Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`color`		
`face`		
`size`		

Syntax

Accessing FONT element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/NN6) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About this object

In a juxtaposition of standards implementations, for the first time the FONT element is exposed as an object only in browsers that also support Cascading Style Sheets as the preferred way to control font faces, colors, and sizes. This change doesn't mean that you shouldn't use FONT elements in your page with the newer

browsers — using this element may be necessary for a single page that needs to be backward-compatible with older browsers. But it does present a quandary for scripters who want to use scripts to modify font characteristics of body text after the page has loaded. A good rule of thumb to follow is to use the FONT element (and script the FONT-HTML element object's properties) when the page must work in all browsers; use style sheets (and their scriptable properties) on pages that will be running exclusively in IE4+ and NN6+.

Properties

color

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

A FONT object's text color can be controlled via the `color` property. Values can be either hexadecimal triplets (for example, `#FFCCFF`) or the plain-language color names recognized by most browsers. In either case, the values are case-insensitive strings.

Example (with Listing 19-1) on the CD-ROM

Related Items: `color` stylesheet attribute.

face

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

A FONT object's font face is controllable via the `face` property. Just as the `FACE` attribute (and the corresponding `font-family` style sheet attribute), you can specify one or more font names in a comma-delimited string. Browsers start with the leftmost font face and look for a match in the client computer's system. The first

matching font face that is found in the client system is applied to the text surrounded by the FONT element. You should list the most specific fonts first, and generally allow the generic font faces (`sans-serif`, `serif`, and `monospace`) to come last; that way you exert at least some control over the look of the font on systems that don't have your pretty fonts. If you know that Windows displays a certain font you like and the Macintosh has something that corresponds to that font but with a different name, you can specify both names in the same property value. The browser skips over font face names not currently installed on the client.

Example on the CD-ROM

Related Items: `font-family` style sheet attribute.

size

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The size of text contained by a FONT element can be controlled via the `size` property. Unlike the more highly recommended `font-size` style sheet attribute, the `size` property of the FONT element object (and its corresponding `SIZE` attribute) are restricted to the relative font size scale imposed by early HTML implementations: a numbering scale from 1 to 7.

Values for the `size` property are strings, even though most of the time they are single numeral values. You can also specify a size relative to the default value by including a plus or minus sign before the number. For example, if the default font size (as set by the browser's user preferences) is 3, then you can bump up the size of a text segment by encasing it inside a FONT element and then setting its `size` property to `"+2"`.

For more accurate font sizing using units, such as pixels or points, use the `font-size` style sheet attribute.

Example on the CD-ROM

Related Items: `font-size` style sheet attribute.

H1 . . . H6 Element Objects

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	Methods	Event Handlers
align		

Syntax

Accessing H1 through H6 element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/NN6) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About these objects

The so-called "heading" elements (denoted by H1, H2, H3, H4, H5, and H6) provide shortcuts for formatting up to six different levels of headings and subheadings. While you can simulate the appearance of these headings with P elements and style sheets, the heading elements very often contain important contextual information about the structure of the document. With the IE5+ and NN6+ powers of inspecting the node hierarchy of a document, a script can generate its own table of contents or outline of a very long document by looking for elements whose `nodeName` properties are in the H*n* family. Therefore, it is a good idea to continue using these elements for contextual purposes, even if you intend to override the default appearance by way of style sheet templates.

As for the scriptable aspects of these six objects, they are essentially the same as the generic contextual objects with the addition of the `align` property. Because each H*n* element is a block-level element, you can use style sheets to set their alignment rather than the corresponding attribute or property. The choice is up to you.

Property

`align`

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

String values of the `align` property control whether the heading element is aligned with the left margin (`left`), center of the page (`center`), or right margin (`right`). The corresponding `ALIGN` attribute is deprecated in HTML 4.0 in favor of the `text-align` style sheet attribute.

Related Items: `text-align` style sheet attribute.

HR Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`align`		
`color`		
`noShade`		
`size`		
`width`		

Syntax

Accessing HR element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/NN6) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About this object

The HR element draws a horizontal rule according to size, dimension, and alignment characteristics normally set by the attributes of this element. Style sheets can also specify many of those settings, the latter route being recommended for pages that will be loaded exclusively in pages that support CSS. In IE4+ and NN6+, your scripts can modify the appearance of an HR element either directly through

element object properties or through style sheet properties. To reference a specific HR element by script, you must assign an `ID` attribute to the element — a practice that you are probably not accustomed to observing.

Properties

`align`

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

An HR object's horizontal alignment can be controlled via the `align` property. String values enable you to set it to align with the left margin (`left`), the center of the page (`center`), or right margin (`right`). By default, the element is centered.

Example (with Listing 19-2) on the CD-ROM

Related Items: `text-align` style sheet attribute.

`color`

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

An HR object's color can be controlled via the `color` property. Values can be either hexadecimal triplets (for example, #`FFCCFF`) or the plain-language color names recognized by most browsers. In either case, the values are case-insensitive strings. If you change the color from the default, the default shading (3-D effect) of the rule disappears. I have yet to find the magic value that lets you return the color to the browser default after it has been set to another color.

Example on the CD-ROM

Related Items: `color` style sheet attribute.

noShade

Value: Boolean Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

A default HR element is displayed with a kind of three-dimensional effect, called *shading*. You can turn shading off under script control by setting the `noShade` property to `true`. But be aware that in IE4+, the `noShade` property is a one-way journey: You cannot restore shading after it is removed. Moreover, default shading is lost if you assign a different color to the rule.

Example on the CD-ROM

Related Items: `color` style sheet attribute.

size

Value: Integer Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The size of an HR element is its vertical thickness, as controlled via the `size` property. Values are integers, representing the number of pixels occupied by the rule.

Example on the CD-ROM

Related Items: None.

width

Value: Integer or String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The width of an HR element is controlled via the `width` property. By default, the element occupies the entire width of its parent container (usually the BODY).

You can specify width as either an absolute number of pixels (as an integer) or as a percentage of the width of the parent container. Percentage values are strings that include a trailing percent character (%).

Example on the CD-ROM

Related Items: `width` style sheet attribute.

LABEL Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`accessKey`		
`form`		
`htmlFor`		

Syntax

Accessing LABEL element object properties or methods:

```
(IE4+)     [window.]document.all.elemID.property | method([parameters])
(IE5+/NN6) [window.]document.getElementById("elemID").property |
           method([parameters])
```

About this object

The LABEL element lets you assign a contextual relationship between a form control (text field, radio button, select list, and so on) and the otherwise freestanding text that is used to label the control on the page. This element does not control the rendering or physical relationship between the control and the label — the HTML source code order does that. Wrapping a form control label inside a LABEL element is important if scripts will be navigating the element hierarchy of a page's content and the relationship between a form control and its label is important to the results of the document parsing.

Properties

accessKey

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

For most other HTML element objects, the `accessKey` property description is covered in the generic element property descriptions of Chapter 15. The function of the property for the LABEL object is the same as the IE implementation for all other elements. The single-character string value is the character key to be used in concert with the OS- and browser-specific modifier key (for example, `Ctrl` in IE for Windows) to bring focus to the form control associated with the label. This value is best set initially via the `ACCESSKEY` attribute for the LABEL element.

Related Items: `accessKey` property of generic elements.

form

Value: Form object reference Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `form` property of a LABEL element object returns a reference to the form object that contains the form control with which the label is associated. This property can be useful in a node parsing script that wants to retrieve the form container from the perspective of the label rather than from the form control. The form object reference returned from the LABEL element object is the same form object reference returned by the `form` property of any form control object.

Related Items: `form` property of INPUT element objects.

htmlFor

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The `htmlFor` property is a string that contains the ID of the form control element with which the label is associated. This value is normally set via the `HTMLFOR` attribute in the LABEL element's tag. Modifying this property does not alter the position or rendering of the label, but it does change the relationships between label and control.

Related Items: None.

MARQUEE Element Object

For HTML element properties, methods, and event handlers, see Chapter 15.

Properties	*Methods*	*Event Handlers*
`behavior`	`start()`	`onBounce`
`bgColor`	`stop()`	`onFinish`
`direction`		`onStart`
`height`		
`hspace`		
`loop`		

Continued

Properties	*Methods*	*Event Handlers*
scrollAmount		
scrollDelay		
trueSpeed		
vspace		
width		

Syntax

Accessing MARQUEE element object properties or methods:

```
(IE4+)   [window.]document.all.elemID.property | method([parameters])
```

About this object

The MARQUEE element is a Microsoft proprietary element that displays scrolling text within a rectangle specified by the WIDTH and HEIGHT attributes of the element. Text that scrolls in the element goes between the element's start and end tags. The IE4+ object model exposes the element and many properties to the object model for control by script.

Properties

behavior

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The behavior property controls details in the way scrolled text moves within the scrolling space. Values for this property are one of the following three strings: alternate, scroll, and slide. When set to alternate, scrolling alternates between left and right (or up and down, depending on the direction property setting). A value of scroll means that the text marches completely to and through the space before appearing again. And a value of slide causes the text to march into

view until the last character is visible. When the `slide` value is applied as a property (instead of as an attribute value in the tag), the scrolling stops when the text reaches an edge of the rectangle. Default behavior for the MARQUEE element is the equivalent of `scroll`.

Example (with Listing 19-3) on the CD-ROM

Related Items: `direction` property of MARQUEE object.

bgColor

Value: Hexadecimal triplet or color name string Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `bgColor` property determines the color of the background of the MARQUEE element's rectangular space. To set the color of the text, either surround the MARQUEE element with a FONT element or apply the `color` style sheet attribute to the MARQUEE element. Values for all color properties can be either the common HTML hexadecimal triplet value (for example, `"#00FF00"`) or any of the Netscape color names (a list is available at `http://developer.netscape.com/docs/manuals/htmlguid/colortab.htm`).

Example on the CD-ROM

direction

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `direction` property lets you get or set the horizontal or vertical direction in which the scrolling text moves. Four possible string values are `left`, `right`, `down`, `up`. The default value is `left`.

Example on the CD-ROM

Related Items: `behavior` property of MARQUEE object.

height
width

Value: Integer Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `height` and `width` properties enable you to get or set the pixel size of the rectangle occupied by the element. You can adjust each property independently of the other, but like most attribute-inspired properties of IE objects, if no `HEIGHT` or `WIDTH` attributes are defined in the element's tag, you cannot use these properties to get the size of the element as rendered by default.

Related Items: None.

hspace
vspace

Value: Integer Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `hspace` and `vspace` properties let you get or set the amount of blank margin space surrounding the MARQUEE element. Adjustments to the `hspace` property affect both the left and right (horizontal) margins of the element; `vspace` governs both top and bottom (vertical) margins. Margin thicknesses are independent of the height and width of the element.

Related Items: None.

loop

Value: Integer Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `loop` property allows you to discover the number of times the MARQUEE element was set to repeat its scrolling according to the `LOOP` attribute. Although this property is read/write, modifying it by script does not cause the text to loop only that number of times more before stopping. Treat this property as read-only.

Related Items: None.

scrollAmount
scrollDelay

Value: Integers Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `scrollAmount` and `scrollDelay` properties control the perceived speed and scrolling smoothness of the MARQUEE element text. The number of pixels between redrawings of the scrolling text is controlled by the `scrollAmount` property. The smaller the number, the less jerky the scrolling is (the default value is `6`). At the same time, you can control the time in milliseconds between each redrawing of the text with the `scrollDelay` property. The smaller the number, the more frequently redrawing is performed (the default value is 85 or 90, depending on the operating system). Thus, a combination of low `scrollAmount` and `scrollDelay` property values presents the smoothest (albeit slow) perceived scrolling.

Example on the CD-ROM

Related Items: `trueSpeed` property of MARQUEE object.

trueSpeed

Value: Boolean Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

IE has a built-in regulator that prevents `SCROLLDELAY` attribute or `scrollDelay` property settings below 60 from causing the MARQUEE element text to scroll too quickly. But if you genuinely want to use a speed faster than 60 (meaning, a value lower than 60), then also set the `trueSpeed` property to `true`.

Related Items: `scrollDelay` property of MARQUEE object.

Methods

start()
stop()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Scripts can start or stop (pause) a MARQUEE element via the `start()` and `stop()` methods. Neither method takes parameters, and you are free to invoke them as often as you like after the page loads. Be aware that the `start()` method does not trigger the `onStart` event handler for the object.

Example on the CD-ROM

Event Handlers

onBounce

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `onBounce` event handler fires only when the MARQUEE element has its behavior set to `alternate`. In that back-and-forth mode, each time the text reaches a boundary and is about to start its return trip, the `onBounce` event fires. If you truly want to annoy your users, you could have the `onBounce` event handlers play a sound at each bounce (I'm kidding — please don't do this).

Related Items: `behavior` property of MARQUEE object.

onFinish

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `onFinish` event handler fires only when the MARQUEE element has its loop set to a specific value of `1` or greater. After the final text loop has completed, the `onFinish` event fires.

Related Items: `loop` property of MARQUEE object.

onStart

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `onStart` event handler fires as the MARQUEE element begins its scrolling, but only as a result of the page loading. The `start()` method does not trigger this event handler.

Related Items: `start()` method of MARQUEE object.

Range Object

Properties	*Methods*	*Event Handlers*
`collapsed`	`cloneContents()`	
`commonAncestorContainer`	`cloneRange()`	
`endContainer`	`collapse()`	
`endOffset`	`compareBoundaryPoints()`	
`startContainer`	`createContextualFragment()`	
`startOffset`	`deleteContents()`	
	`detach()`	
	`extractContents()`	
	`insertNode()`	
	`isValidFragment()`	
	`selectNode()`	
	`selectNodeContents()`	
	`setEnd()`	
	`setEndAfter()`	
	`setEndBefore()`	
	`setStart()`	
	`setStartAfter()`	
	`setStartBefore()`	
	`surroundContents()`	
	`toString()`	

Syntax

Creating a `Range` object:

```
var rangeRef = document.createRange()
```

Accessing `Range` object properties or methods:

```
(NN6+)    rangeRef.property | method([parameters])
```

About this object

The first release of NN6 suffers from several bugs and omissions with respect to the `Range` object. Discussions about the NN6 `Range` object throughout this chapter cover some features that may not be implemented or fixed until a later version of the NN6 browser. I mention specific bugs and omissions found in the early NN6 whenever the description here does not yet match the browser implementation. Even some of the example listings do not work correctly (or at all) with the first release of NN6. In time, however, everything described in this section will be a part of the Netscape browser.

The `Range` object is the W3C DOM (Level 2) version of what Microsoft had implemented earlier as its `TextRange` object. A number of important differences (not the least of which is an almost entirely different property and method vocabulary) distinguish the behaviors and capabilities of these two similar objects. Although Microsoft participated in the W3C DOM Level 2 working groups, no participant from the company is credited on the DOM specification chapter regarding the `Range` object. Because the W3C version has not been implemented as of IE5.5, it is unknown if IE will eventually implement the W3C version. In the meantime, see the IE/Windows `TextRange` object section later in this chapter for comparisons between the two objects. Neither the W3C DOM `Range` nor Microsoft `TextRange` objects are implemented in IE5/Mac.

The purpose of the W3C DOM `Range` object is to provide hooks to a different "slice" of content (most typically a portion of a document's content) that is not necessarily restricted to the node hierarchy (tree) of a document. While a `Range` object can be used to access and modify nodes and elements, it can also transcend node and element boundaries to encompass arbitrary segments of a document's content. The content contained by a range is sometimes referred to as a *selection,* but this does not mean that the text is highlighted on the page, such as a user selection. Instead, the term "selection" here means a segment of the document's content that can be addressed as a unit, separate from the node tree of the document. As soon as the range is created, a variety of methods let scripts examine, modify, remove, replace, and insert content on the page.

A range object (meaning, an instance of the static `Range` object) has a start point and an end point, which together define the boundaries of the range. The points are defined in terms of an offset count of positions within a container. These counts are usually character positions within text nodes (ignoring any HTML tag or attribute characters), but when both boundaries are at the edges of the same node, the

offsets may also be counts of nodes within a container that surrounds both the start and end points. An example helps clarify these concepts.

Consider the following simplified HTML document:

```
<HTML>
<BODY>
<P>This paragraph has an <EM>emphasized</EM> segment.</P>
</BODY>
</HTML>
```

You can create a range that encompasses the text inside the EM element from several points of view, each with its own offset counting context:

1. *From the EM element's only child node (a text node).* The offset of the start point is zero, which is the location of the insertion point in front of the first character (lowercase "e"); the end point offset is 10, which is the character position (zero-based) following the lowercase "d".
2. *From the EM element.* The point of view here is that of the child text node inside the EM element. Only one node exists here, and the offset for the start point is 0, while the offset for the end point is 1.
3. *From the P element's child nodes (two text nodes and an element node).* You can set the start point of a range to the very end (counting characters) of the first child text node of the P element; you can then set the end point to be in front of the first character of the last child text node of the P element. The resulting range encompasses the text within the EM element.
4. *From the P element.* From the point of view of the P element, the range can be set with an offset starting with 1 (the second node nested inside the P element) and ending with 2 (the start of the third node).

While these different points of view provide a great deal of flexibility, they also can make it more difficult to imagine how you can use this power. The W3C vocabulary for the `Range` methods, however, helps you figure out what kind of offset measure to use.

A range object's start point could be in one element, and its end point in another. For example, consider the following HTML:

```
<P>And now to introduce our <EM>very special</EM> guest:</P>
```

If the text shown in boldface indicates the content of a range object, you can see that the range crosses element boundaries in a way that would make HTML element or node object properties difficult to use for replacing that range with some other text. The W3C specification provides guidelines for browser makers on how to handle the results of removing or inserting HTML content that crosses node borders.

An important aspect of the Range object is that the size of a range can be zero or more characters. Start and end points always position themselves between characters. When the start point and end point of a range are at the same location, the range acts like a text insertion pointer.

Working with ranges

To create a range object, use the document.createRange() method and assign the range object returned by this method to a variable that you can use to control the range:

```
var rng = document.createRange()
```

The first release of NN6 requires that a newly created range be more explicitly defined (as described in a moment) before scripts can access the range's properties. The W3C DOM, however, suggests that a new range has as its containing node the document node (which encompasses all content of the page, including the <HTML> tag set). Moreover, the start and end points are set initially to zero, meaning that the initial range is collapsed at the very beginning of the document.

With an active range stored in a variable, you can use many of the object's methods to adjust the start and end points of the range. If the range is to consist of all of the contents of a node, you have two convenience methods that do so from different points of view: selectNode() and selectNodeContents(). The sole parameter passed with both methods is a reference to the node whose contents you want to turn into a range. The difference between the two methods is how the offset properties of the range are calculated as a result (see the discussion about these methods later in the chapter for details). Another series of methods (setStartBefore(), setStartAfter(), setEndBefore(), and setEndAfter()) let you adjust each end point individually to a position relative to a node boundary. For the most granular adjustment of boundaries, the setStart() and setEnd() methods let you specify a reference node (where to start counting the offset) and the offset integer value.

If you need to select an insertion point (for example, to insert some content into an existing node), you can position either end point where you want it, and then invoke the collapse() method. A parameter determines whether the collapse should occur at the range's start or end point.

A suite of other methods lets your scripts work with the contents of a range directly. You can copy (cloneContents()), delete (deleteContents(), extractContents()), insert a node (insertNode()), and even surround a range's contents with a new parent node (surroundContents()). Several properties let your scripts examine information about the range, such as the offset values, the containers that hold the offset locations, whether the range is collapsed, and a reference to the next outermost node that contains both the start and end points.

Netscape adds a proprietary method to the `Range` object (which is actually a method of an object that is built around the `Range` object) called `createContextualFragment()`. This method lets scripts create a valid node (of type `DocumentFragment`) from arbitrary strings of HTML content — a feature that the W3C DOM does not (yet) offer. This method was devised at first as a substitute for what eventually became the NN6 `innerHTML` property.

Using the `Range` object can be a bit tedious, because it often requires a number of script statements to execute an action. Three basic steps are generally required to work with a `Range` object:

1. Create the text range.
2. Set the start and end points.
3. Act on the range.

As soon as you are comfortable with this object, you will find it provides a lot of flexibility in scripting interaction with body content. For ideas about applying the `Range` object in your scripts, see the examples that accompany the descriptions of individual properties and methods in the following sections.

Properties

collapsed

Value: Boolean Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `collapsed` property reports whether a range has its start and end points set to the same position in a document. If the value is `true`, then the range's start and end containers are the same and the offsets are also the same. You can use this property to verify that a range is in the form of an insertion pointer just prior to inserting a new node:

```
if (rng.collapsed) {
    rng.insertNode(someNewNodeReference)
}
```

Example on the CD-ROM

Related Items: `endContainer`, `endOffset`, `startContainer`, `startOffset` properties; `Range.collapse()` method.

commonAncestorContainer

Value: Node object reference Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `commonAncestorContainer` property returns a reference to the document tree node that both the start and end points have in common. It is not uncommon for a range's start point to be in one node and the end point to be in another. Yet a more encompassing node most likely contains both of those nodes, perhaps even the `document.body` node. The W3C DOM specification also calls the shared ancestor node the *root node* for the range (a term that may make more sense to you).

Example on the CD-ROM

Related Items: `endContainer`, `endOffset`, `startContainer`, `startOffset` properties; all "set" and "select" methods of the Range object.

endContainer
startContainer

Value: Node object reference Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `endContainer` and `startContainer` properties return a reference to the document tree node that contains the range's end point and start point, respectively. Be aware that the object model calculates the container, and the container may not be the reference you used to set the start and end points of a range. For example, if you use the `selectNode()` method to set the start and end points of a range to encompass a particular node, the containers of the end points are most likely the

next outermost nodes. Thus, if you want to expand a range to the start of the node that contains the current range's start point, you can use the value returned by the `startContainer` property as a parameter to the `setStartBefore()` method:

```
rng.setStartBefore(rng.startContainer)
```

Example on the CD-ROM

Related Items: `commonAncestor`, `endOffset`, `startOffset` properties; all "set" and "select" methods of the Range object.

endOffset
startOffset

Value: Integer Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `endOffset` and `startOffset` properties return an integer count of the number of characters or nodes for the location of the range's end point and start point, respectively. These counts are relative to the node that acts as the container node for the position of the boundary (see `Range.endContainer` and `Range.startContainer` properties earlier in this chapter).

When a boundary is at the edge of a node (or perhaps "between" nodes is a better way to say it), the integer returned is the offset of nodes (zero-based) within the boundary's container. But when the boundary is in the middle of a text node, the integer returned is an index of the character position within the text node. The fact that each boundary has its own measuring system (nodes versus characters, relative to different containers) can get confusing if you're not careful, because conceivably the integer returned for an end point could be smaller than that for the start point. Consider the following nested elements:

```
<P>This paragraph has an <EM>emphasized</EM> segment.</P>
```

The next script statements set the start of the range to a character within the first text node and the end of the range to the end of the EM node:

```
var rng = document.createRange()
rng.setStart(document.getElementById("myP").firstChild, 19)
rng.setEndAfter(document.getElementById("myEM"))
```

Using bold face to illustrate the body text that is now part of the range and the pipe (|) character to designate the boundaries as far as the nodes are concerned, here is the result of the above script execution:

```
<P ID="myP">This paragraph has |an <EM ID="myEM">emphasized</EM>| segment.</P>
```

Because the start of the range is in a text node (the first child of the P element), the range's `startOffset` value is 19, which is the zero-based character position of the "a" of the word "an." The end point, however, is at the end of the EM element. The system recognizes this point as a node boundary, and thus counts the `endOffset` value within the context of the end container: the P element. The `endOffset` value is 2 (the P element's text node is node index 0; the EM element is node index 1; and the position of the end point is at the start of the P element's final text node, at index 2).

For the `endOffset` and `startOffset` values to be of any practical use to a script, you must also use the `endContainer` and `startContainer` properties to read the context for the offset integer values.

Example on the CD-ROM

Related Items: `endContainer`, `startContainer` properties; all "set" and "select" methods of the `Range` object.

Methods

cloneContents()
cloneRange()

Returns: `DocumentFragment` node reference; `Range` object reference.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `cloneContents()` method (not implemented in NN6.0, but expected in a future release) takes a snapshot copy of the contents of a `Range` object and returns a reference to that copy. The copy is stored in the browser's memory, but is not part of the document tree. The `cloneRange()` method (available in NN6.0) performs the same action on an entire range and stores the range copy in the browser's memory. A range's contents can consist of portions of multiple nodes and may not be surrounded by an element node; that's why its data is of the type `DocumentFragment`

(one of the W3C DOM's node types). Because a `DocumentFragment` node is a valid node, it can be used with other document tree methods where nodes are required as parameters. Therefore, you can clone a text range to insert a copy elsewhere in the document.

In contrast, the `cloneRange()` method deals with range objects. While you are always free to work with the contents of a range object, the `cloneRange()` method returns a reference to a range object, which acts as a kind of wrapper to the contents (just as it does when the range is holding content in the main document). You can use the `cloneRange()` method to obtain a copy of one range to compare the end points of another range (via the `Range.compareBoundaryPoints()` method).

Example on the CD-ROM

Related Items: `compareBoundaryPoints()`, `extractContents()` methods.

collapse([*startBoolean*])

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

Use the `collapse()` method to shrink a range from its current size down to a single insertion point between characters. Collapsing a range becomes more important than you may think at first, especially in a function that is traversing the body or large chunk of text. For example, in a typical looping word-counting script, you create a text range that encompasses the body fully. To begin counting words, you can first collapse the range to the insertion point at the very beginning of the range. Next, use the `expand()` method to set the range to the first word of text (and increment the counter if the `expand()` method returns `true`). At that point, the text range extends around the first word. You want the range to collapse at the end of the current range so that the search for the next word starts after the current one. Use `collapse()` once more, but this time with a twist of parameters.

The optional parameter of the `collapse()` method is a Boolean value that directs the range to collapse itself either at the start or end of the current range. The default behavior is the equivalent of a value of `true`, which means that unless otherwise directed, a `collapse()` method shifts the text range to the point in front of the current range. This method works great at the start of a word-counting script, because you want the text range to collapse to the start of the text. But for subsequent movements through the range, you want to collapse the range so that it is

after the current range. Thus, you include a `false` parameter to the `collapse()` method.

Example on the CD-ROM

Related Items: `Range.setEnd()`, `Range.setStart()` methods.

compareBoundaryPoints(*typeInteger*, *sourceRangeRef*)

Returns: Integer (`-1`, `0`, or `1`).

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

Generating multiple range objects and assigning them to different variables is not a problem. You can then use the `compareBoundaryPoints()` method to compare the relative positions of start and end points of both ranges. One range is the object you use to invoke the `compareBoundaryPoints()` method, and the other range is the second parameter of the method. The order in which you reference the two ranges influences the results, based on the value assigned to the first parameter.

Values for the first parameter can be one of four constant values that are properties of the static `Range` object: `Range.START_TO_START`, `Range.START_TO_END`, `Range.END_TO_START`, and `Range.END_TO_END`. What these values specify is which point of the current range is compared with which point of the range passed as the second parameter. For example, consider the following body text that has two text ranges defined within it:

```
It was the best of times.
```

The first text range (assigned in our discussion here to variable `rng1`) is shown in boldface, while the second text range (`rng2`) is shown in bold-italic. In other words, `rng2` is nested inside `rng1`. We can compare the position of the start of `rng1` against the position of the start of `rng2` by using the `Range.START_TO_START` value as the first parameter of the `compareBoundaryPoints()` method:

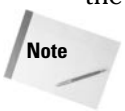
```
var result = rng1.compareBoundaryPoints(Range.START_TO_START, rng2)
```

The value returned from the `compareBoundaryPoints()` method is an integer of one of three values. If the positions of both points under test are the same, then the value returned is `0`. If the start point of the (so-called source) range is before the

range on which you invoke the method, the value returned is -1; in the opposite positions in the code, the return value is 1. Therefore, from the example above, because the start of rng1 is before the start of rng2, the method returns -1. If you change the statement to invoke the method on rng2, as in

```
var result = rng2.compareBoundaryPoints(Range.START_TO_START, rng1)
```

the result is 1.



In the first release of NN6, the returned values of 1 and -1 are the opposite of what they should be. This is to be corrected in a subsequent release.

In practice, this method is helpful in knowing if two ranges are the same, if one of the boundary points of both ranges is the same, or if one range starts where the other ends.

Example (with Listing 19-4) on the CD-ROM

Related Items: None.

createContextualFragment("*text*")

Returns: W3C DOM DocumentFragment Node.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The createContextualFragment() method is a method of the NN6 Range object (a proprietary extension of the W3C DOM Range object). This method provides a way, within the context of the W3C DOM Level 2 node hierarchy to create a string of HTML text (with or without HTML tags, as needed) for insertion or appendage to existing node trees. During the development of the NN6 browser, this method filled a gap that was eventually filled by Netscape's adoption of the Microsoft proprietary innerHTML property. The method obviates the need for tediously assembling a complex HTML element via a long series of document.createElement() and document.createTextNode() methods for each segment, plus the assembly of the node tree prior to inserting it into the actual visible document. The existence of the innerHTML property of all element objects, however, reduces the need for the createContextualFragment() method, while allowing more code to be shared across browser brands.

The parameter to the `createContextualFragment()` method is any text, including HTML tags. To invoke the method, however, you need to have an existing range object available. Therefore, the sequence used to generate a document fragment node is

```
var rng = document.createRange()
rng.selectNode(document.body) // any node will do
var fragment = rng.createContextualFragment("<H1>Howdy</H1>")
```

As a document fragment, the node is not part of the document node tree until you use the fragment as a parameter to one of the tree modification methods, such as `Node.insertBefore()` or `Node.appendChild()`.

Example on the CD-ROM

Related Items: Node object (Chapter 15).

deleteContents()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `deleteContents()` method removes all contents of the current range from the document tree. After deletion, the range collapses to an insertion point where any surrounding content (if any) cinches up to its neighbors.

Some alignment of a range's boundaries forces the browser to make decisions about how element boundaries inside the range are treated after the deletion. An easy deletion is one for which the range boundaries are symmetrical. For example, consider the following HTML with a range highlighted in bold:

```
<P>One paragraph with an <EM>emphasis</EM> inside.</P>
```

After you delete the contents of this range, the text node inside the EM element disappears, but the EM element remains in the document tree (with no child nodes). Similarly, if the range is defined as being the entire second child node of the P element, as follows

```
<P>One paragraph with an <EM>emphasis</EM> inside.</P>
```

then deleting the range contents removes both the text node and the EM element node, leaving the P element with a single, unbroken text node as a child (although

in the previous case, an extra space would be between the words "an" and "inside" because the EM element does not encompass a space on either side).

When range boundaries are not symmetrical, the browser does its best to maintain document tree integrity after the deletion. Consider the following HTML and range:

```
<P>One paragraph with an <EM>emphasis</EM> inside.</P>
```

After deleting this range's contents, the document tree for this segment looks like the following:

```
<P>One paragraph <EM>phasis</EM> inside.</P>
```

The range collapses to an insertion point just before the <EM> tag. But notice that the EM element persisted to take care of the text still under its control. Many other combinations of range boundaries and nodes are possible, so be sure that you check out the results of a contents deletion for asymmetrical boundaries before applying the deletion.

Example on the CD-ROM

Related Items: `Range.extractContents()` method.

detach()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `detach()` method instructs the browser to release the current range object from the object model. In the process, the range object is nulled out to the extent that an attempt to access the object results in a script error. You can still assign a new range to the same variable if you like. You are not required to detach a range when you're finished with it, and the browser resources employed by a range are not that large. But it is good practice to "clean up after yourself," especially when a script repetitively creates and manages a series of new ranges.

Related Items: `document.createRange()` method.

extractContents()

Returns: DocumentFragment node reference.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The extractContents() method (not implemented in the first release of NN6) deletes the contents of the range and returns a reference to the document fragment node that is held in the browser memory, but which is no longer part of the document tree. A range's contents can consist of portions of multiple nodes and may not be surrounded by an element node; that's why its data is of the type DocumentFragment (one of the W3C DOM's node types). Because a DocumentFragment node is a valid node, it can be used with other document tree methods where nodes are required as parameters. Therefore, you can extract a text range from one part of a document to insert elsewhere in the document.

Example on the CD-ROM

Related Items: cloneContents(), deleteContents() methods.

insertNode(*nodeReference*)

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The insertNode() method (not implemented in the first release of NN6) inserts a node at the start point of the current range. The node being inserted may be an element or text fragment node, and its source can be any valid node creation mechanism, such the document.createTextNode() method or any node extraction method.

Example (with Listing 19-5) on the CD-ROM

Related Items: None.

isValidFragment("*HTMLText*")

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `isValidFragment()` method belongs to the Netscape-specific version of the W3C DOM `Range` object. The method validates text as to whether it can be successfully converted to a document fragment node via Netscape's other proprietary `Range` method, `createContextualFragment()`. Knowing that this is *not* an HTML or XML validator is important. Ideally, you pass the text through the `isValidFragment()` method prior to creating the fragment, as in the following:

```
var rng = document.createRange()
rng.selectNode(document.body)
var newHTML = "<H1>Howdy</H1>"
if (rng.isValidFragment(newHTML)) {
    var newFragment = rng.createContextualFragment(newHTML)
}
```

See the description of the `Range.createContextualFragment()` method earlier in this chapter for the application of a document fragment node in NN6.

Example on the CD-ROM

Related Items: `Range.createContextualFragment()` method.

selectNode(*nodeReference*)
selectNodeContents(*nodeReference*)

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `selectNode()` and `selectNodeContents()` methods are convenience methods for setting both end points of a range to surround a node or a node's contents. The kind of node you supply as the parameter to either method (text node or element node) has a bearing on the range's container node types and units of

measure for each (see the container- and offset-related properties of the Range object earlier in this chapter).

Starting with the `selectNode()` method, if you specify an element node as the one to select, the start and end container node of the new range is the next outermost element node; offset values count nodes within that parent element. If you specify a text node to be selected, the container node for both ends is the parent element of that text node; offset values count the nodes within that parent.

With the `selectNodeContents()` method, the start and end container nodes are the very same element specified as the parameter; offset values count the nodes within that element. If you specify a text node's contents to be selected, the text node is the start and end parent, but the range is collapsed at the beginning of the text.

By and large, you specify element nodes as the parameter to either method, allowing you to set the range to either encompass the element (via `selectNode()`) or just the contents of the element (via `selectNodeContents()`).

Example on the CD-ROM

Related Items: `setEnd()`, `setEndAfter()`, `setEndBefore()`, `setStart()`, `setStartAfter()`, `setStartBefore()` methods.

setEnd(*nodeReference, offset*)
setStart(*nodeReference, offset*)

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

You can adjust the start and end points of a text range independently via the `setStart()` and `setEnd()` methods. While not as convenient as the `selectNode()` or `selectNodeContents()` methods, these two methods give you the ultimate in granularity over precise positioning of a range boundary.

The first parameter to both methods is a reference to a node. This reference can be an element or text node, but your choice here also influences the kind of measure applied to the integer offset value supplied as the second parameter. When the first parameter is an element node, the offset counts are in increments of child nodes inside the specified element node. But if the first parameter is a text node, the offset counts are in increments of characters within the text node.

When you adjust the start and end points of a range with these methods, you have no restrictions to the symmetry of your boundaries. One boundary can be defined relative to a text node, while the other relative to an element node — or vice versa.

To set the end point of a range to the last node or character within a text node (depending on the unit of measure for the *offset* parameter), you can use the `length` property of the units being measured. For example, to set the end point to the end of the last node within an element (perhaps there are multiple nested elements and text nodes within that outer element), you can use the first parameter reference to help you get there:

```
rng.setEnd(document.getElementById("myP"),
document.getElementById("myP").childNodes.length)
```

These kinds of expressions get lengthy, so you may want to make a shortcut to the reference to simplify the values of the parameters, as shown in this version that sets the end point to after the last character of the last text node of a P element:

```
var nodeRef = document.getElementById("myP").lastChild
rng.setEnd(nodeRef, nodeRef.nodeValue.length)
```

In both previous examples with the `length` properties, the values of those properties are always pointing to the node or character position after the final object because the index values for those objects' counts are zero-based. Also bear in mind that if you want to set a range end point at the edge of a node, you have four other methods to choose from (`setEndAfter()`, `setEndBefore()`, `setStartAfter()`, `setStartBefore()`). The `setEnd()` and `setStart()` methods are best used when an end point needs to be set at a location other than at a node boundary.

Example on the CD-ROM

Related Items: `selectNode()`, `selectNodeContents()`, `setEndAfter()`, `setEndBefore()`, `setStartAfter()`, `setStartBefore()` methods.

`setEndAfter(`*nodeReference*`)`
`setEndBefore(`*nodeReference*`)`
`setStartAfter(`*nodeReference*`)`
`setStartBefore(`*nodeReference*`)`

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

You can adjust the start and end points of a text range relative to existing node boundaries via your choice of these four methods. The "before" and "after" designations are used to specify which side of the existing node boundary the range should have for its boundary. For example, using `setStartBefore()` and `setEndAfter()` with the same element node as a parameter is the equivalent of the `selectNode()` method on that element. You may also specify a text node as the parameter to any of these methods. But because these methods work with node boundaries, the offset values are always defined in terms of node counts, rather than character counts. At the same time, however, the boundaries do not need to be symmetrical, so that one boundary can be inside one node and the other boundary inside another node.

Example on the CD-ROM

Related Items: `selectNode()`, `selectNodeContents()`, `setEnd()`, `setStart()` methods.

`surroundContents(`*nodeReference*`)`

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

The `surroundContents()` method (not implemented in the first release of NN6) surrounds the current range with a new parent element. Pass the new parent element as a parameter to the method. No document tree nodes or elements are removed or replaced in the process, but the current range becomes a child node of the new node; if the range coincides with an existing node, then the relationship between that node and its original parent becomes that of grandchild and grandparent. An application of this method may be to surround user-selected text with a SPAN element whose class renders the content with a special font style or other display characteristic based on a style sheet selector for that class name.

When the element node being applied as the new parent has child nodes itself, those nodes are discarded before the element is applied to its new location. Therefore, for the most predictable results, using content-free element nodes as the parameter to the `surroundContents()` method is best.

Example (with Listing 19-6) on the CD-ROM

Related Items: `Range.insertNode()` method.

`toString()`

Returns: String.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

Use the `toString()` method to retrieve a copy of the body text that is contained by the current text range. The text returned from this method is ignorant of any HTML tags or node boundaries that exist in the document tree. You also use this method (eventually) to get the text of a user selection, after it has been converted to a text range (as soon as NN6 implements the planned feature).

Example on the CD-ROM

Related Items: `selection.getRangeAt()`, `Range.extractContents()` methods.

selection Object

Properties	*Methods*	*Event Handlers*
`type`	`clear()`	
	`createRange()`	
	`empty()`	

Syntax

Accessing `selection` object properties or methods:

```
(IE4+)     [window.]document.selection.property | method()
```

About this object

In some ways, the short list of properties and methods for the selection object is misleading. The items shown in the list belong to the IE4+ `selection` object. NN6 implements a `selection` object (not a part of the W3C DOM), but the first release of the browser does not provide a way to create such an object. Opening remarks below provide a preview of how the NN6 `selection` object will work whenever it is implemented. Details about properties and methods are not provided at this time.

The IE version

The IE4+ `selection` object is a property of the `document` object, providing scripted access to any body text or text in a form text control that is selected either by the user or by script. A `selection` object of one character or more is always highlighted on the page, and only one `selection` object can be active at any given instant.

Take advantage of the `selection` object when your page invites a user to select text for some operation that utilizes the selected text. The best event to use for working with a selection is the `onMouseUp` event handler. This event fires on every release of the mouse, and your script can investigate the `document.selection` object to see if any text has been selected (using the selection's `type` property). Turn a selection into a `TextRange` object via the `createRange()` method. You can then use the `text` property of the text range to access the actual selected characters. This sequence may seem like a long way to go for the text, perhaps, but the IE `selection` object provides no direct property for reading or writing the selected text.

If you intend to perform some action on a selection, you may not be able to trigger that action by way of a button or link. In some browser versions and operating systems, clicking one of these elements automatically deselects the body selection.

The NN version

Navigator 4 provides the `document.getSelection()` method to let scripts look at the selected body text, but you have no `selection` object per se for that browser. The NN6 `selection` object intends to improve the situation.

The `document.getSelection()` is deprecated in NN6 in favor of the roundabout way of getting a copy of a selection similar to the IE route described previously: Make a range out of the selection and get the text of the range. To obtain the `selection` object representing the current selection, use the `window.getSelection()` method (as soon as the method is implemented in NN6). One important difference between the IE and NN selections is that the NN6 `selection` object works only on body text, and not on selections inside text-oriented form controls.

An NN6 `selection` object has relationships with the document's node tree in that the object defines itself by the nodes (and offsets within those nodes) that encase the start and end points of a selection. When a user drags a selection, the node in which the selection starts is called the *anchor* node; the node holding the text at the point of the selection release is called the *focus* node; for double- or triple-clicked selections, the direction between anchor and focus nodes is in the direction of the language script (for example, left-to-right in Latin-based script families). In many ways, an NN6+ `selection` object behaves just as the W3C DOM `Range` object, complete with methods to collapse and extend the selection. Unlike a range, however, the text encompassed by a `selection` object is highlighted on the page. If your scripts need to work with the nodes inside a selection, the `getRangeAt()` method of the `selection` object returns a range object whose boundary points coincide with the selection's boundary points.

Properties

type

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `type` property returns `Text` whenever a selection exists on the page. Otherwise the property returns `None`. A script can use this information to determine if a selection is made on the page:

```
if (document.selection.type == "Text") {
    // process selection
    ...
}
```

Microsoft indicates that this property can sometimes return `Control`, but that terminology is associated with an edit mode outside the scope of this book.

Example (with Listing 19-7) on the CD-ROM

Related Items: `TextRange.select()` method.

Methods

clear()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Use the `clear()` method to delete the current selection from the document. To the user, the `clear()` method has the same effect as setting the `TextRange.text` property to an empty string. The difference is that you can use the `clear()` method without having to generate a text range for the selection. After you delete a selection, the `selection.type` property returns `None`.

Example on the CD-ROM

Related Items: `selection.empty()` method.

createRange()

Returns: `TextRange` object.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

To generate a text range for a user selection in IE, invoke the `createRange()` method of the `selection` object. I'm not sure why the method for the `selection` object is called `createRange()` while text ranges for other valid objects are created with a `createTextRange()` method. The result of both methods is a full-fledged `TextRange` object.

Example on the CD-ROM

Related Items: `TextRange` object.

empty()

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `empty()` method deselects the current IE selection. After deselection, the `selection.type` property returns `None`. The action of the `empty()` method is the same as the `UnSelect` command invoked via the `execCommand()` method for a document. If the selection was made from a `TextRange` object (via the `TextRange.select()` method), the `empty()` method affects only the visible selection and not the text range.

Example on the CD-ROM

Related Items: `selection.clear()` method.

Text and TextNode Objects

Properties	*Methods*	*Event Handlers*
attributes†	appendChild()†	
childNodes†	appendData()	
data	cloneNode()†	
firstChild†	deleteData()	
lastChild†	hasChildNodes()†	
length†	insertBefore()†	
localName†	insertData()	
namespaceURI†	normalize()†	
nextSibling†	removeChild()†	
nodeName†	replaceChild()†	
nodeType†	replaceData()	
nodeValue†	splitText()	

Properties	Methods	Event Handlers
ownerDocument†	substringData()	
parentNode†		
prefix†		
previousSibling†		

†See Chapter 15

Syntax

Accessing `Text` and `TextNode` object properties or methods:

```
(IE5+/NN6+)  [window.]document.getElementById("id").textNodeRef.property |
             method()
```

About this object

Discussing both the `Text` object of the W3C DOM and NN6 in the same breath as the IE5+ `TextNode` object is a little tricky. Conceptually, they are the same kind of object in that they are the document tree objects — text nodes — that contain an HTML element's text (see Chapter 14 for details on the role of the text node in the `document` object hierarchy). Generating a new text node by script is achieved the same way in both object models: `document.createTextNode()`. What makes the discussion of the two objects tricky is that while the W3C DOM version comes from a strictly object-oriented specification (in which a text node is an instance of a `CharacterData` object, which, in turn is an instance of the generic `Node` object), the IE object model is not quite as complete. For example, while the W3C DOM `Text` object inherits all of the properties and methods of the `CharacterData` and `Node` definitions, the IE `TextNode` object exposes only those properties and method that Microsoft deems appropriate.

No discrepancy in terminology gets in the way as to what to call these objects because their object names never become part of the script. Instead script statements always refer to text nodes by other means, such as through a child node-related property of an element object or as a variable that receives the result of the `document.createTextNode()` method.

While both objects share a number of properties and one method, the W3C DOM `Text` object contains a few methods that have "data" in their names. These properties and methods are inherited from the `CharacterData` object in the DOM specification. They are discussed as a group in the section about object methods in this chapter. In all cases, check the browser version support for each property and method described here.

Properties

data

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓			✓	✓	✓

The `data` property contains the string comprising the text node. Its value is identical to the `nodeValue` property of a text node. See the description of the `nodeValue` property in Chapter 15.

Example on the CD-ROM

Related Items: `nodeValue` property of all element objects (Chapter 15).

Methods

```
appendData("text")
deleteData(offset, count)
insertData(offset, "text")
replaceData(offset, count, "text")
substringData(offset, count)
```

Returns: See text.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓					

These five methods of the W3C DOM `Text` object provide scripted manipulation of the text inside a text node. Methods that modify the node's data automatically change the values of both the `data` and `nodeValue` properties.

The purposes of these methods are obvious for the most part. Any method that requires an *`offset`* parameter uses this integer value to indicate where in the

existing text node the deletion, insertion, or replacement starts. Offsets are zero-based, meaning that to indicate the action should take place starting with the first character, specify a zero for the parameter. A *count* parameter is another integer, but one that indicates how many characters are to be included. For example, consider a text node that contains the following data:

```
abcdefgh
```

This node could be a node of an element on the page or a node that has been created and assigned to a variable but not yet inserted into the page. To delete the first three characters of that text node, the statement is

```
textNodeReference.deleteData(0,3)
```

This leaves the text node content as

```
defgh
```

As for the `replaceData()` method, the length of the text being put in place of the original chunk of text need not match the *count* parameter. The *count* parameter, in concert with the *offset* parameter, defines what text is to be removed and replaced by the new text.

The `substringData()` method is similar to the JavaScript core language `String.substr()` method in that both require parameters indicating the offset within the string to start reading and for how many characters. You get the same result with the `substringData()` method of a text node as you do from a `nodeValue.substr()` method when both are invoked from a valid text node object.

Of all five methods discussed here, only `substringData()` returns a value: a string.

Example (with Listing 19-8) on the CD-ROM

Related Items: `appendChild()`, `removeChild()`, `replaceChild()` methods of element objects (Chapter 15).

splitText(*offset*)

Returns: `Text` or `TextNode` object.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility				✓				✓	✓

The `splitText()` method performs multiple actions with one blow. The *`offset`* parameter is an integer indicating the zero-based index position within the text node at which the node is to divide into two nodes. After you invoke the method on the current text node, the current node consists of the text from the beginning of the node up to the offset position. The method returns a reference to the text node whose data starts with the character after the dividing point and extends to the end of the original node. Users won't notice any change in the rendered text: This method influences only the text node structure of the document. Using this method means, for example, that an HTML element that starts with only one text node will have two after the `splitText()` method is invoked. The opposite action (combining contiguous text node objects into a single node) is performed by the NN6 `normalize()` method (Chapter 15).

Example on the CD-ROM

Related Items: `normalize()` method (Chapter 15).

TextRange Object

Properties	*Methods*	*Event Handlers*
`boundingHeight`	`collapse()`	
`boundingLeft`	`compareEndPoints()`	
`boundingTop`	`duplicate()`	
`boundingWidth`	`execCommand()`	
`htmlText`	`expand()`	
`offsetLeft`†	`findText()`	
`offsetTop`†	`getBookmark()`	
`text`	`getBoundingClientRect()`†	
	`getClientRects()`†	
	`inRange()`	
	`isEqual()`	
	`move()`	
	`moveEnd()`	
	`moveStart()`	
	`moveToBookmark()`	
	`moveToElementText()`	

Properties	Methods	Event Handlers
	moveToPoint()	
	parentElement()	
	pasteHTML()	
	queryCommandEnabled()	
	queryCommandIndeterm()	
	queryCommandState()	
	queryCommandSupported()	
	queryCommandText()	
	queryCommandValue()	
	scrollIntoView()†	
	select()	
	setEndPoint()	

†See Chapter 15

Syntax

Creating a TextRange object:

```
var rangeRef = document.body.createTextRange()
var rangeRef = buttonControlRef.createTextRange()
var rangeRef = textControlRef.createTextRange()
var rangeRef = document.selection.createRange()
```

Accessing TextRange object properties or methods:

```
(IE4+)        rangeRef.property | method([parameters])
```

About this object

Unlike most of the objects covered in Part III of the book, the IE4+ TextRange object is not tied to a specific HTML element. The TextRange object is, instead, an abstract object that represents text content anywhere on the page (including text content of a text-oriented form control) between a start point and an end point (collectively, the *boundaries* of the range). The user may not necessarily know that a TextRange object exists, because no requirement exists to force a TextRange object to physically select text on the page (although the TextRange object can be used to assist scripts in automating the selection of text; or a script may turn a user selection into a TextRange object for further processing).

The purpose of the TextRange object is to give scripts the power to examine, modify, remove, replace, and insert content on the page. Start and end points of an IE TextRange object are defined exclusively in terms of character positions within the element that is used to create the range (usually the BODY element, but also button- and text-related form control elements). Character positions of body text do not take into account source code characters that may define HTML elements. This factor is what distinguishes a TextRange's behavior from, for instance, the various properties and methods of HTML elements that let you modify or copy elements and their text (for example, innerText and outerText properties). A TextRange object's start point can be in one element, and its end point in another. For example, consider the following HTML:

```
<P>And now to introduce our <EM>very special</EM> guest:</P>
```

If the text shown in boldface indicates the content of a TextRange object, you can see that the range crosses element boundaries in a way that makes HTML element object properties difficult to use for replacing that range with some other text. Challenges still remain in this example, however. Simply replacing the text of the range with some other text forces your script (or the browser) to reconcile the issue of what to do about the nested EM element, because the TextRange object handles only its text. (Your word processing program must address the same kind of issue when you select a phrase that starts in italic but ends in normal font, and then you paste text into that selection.)

An important aspect of the TextRange object is that the size of the range can be zero or more characters. Start and end points always position themselves between characters. When the start point and end point of a range are at the same location, the range acts as a text insertion pointer. In fact, when the TextRange object represents text inside a text-related form control, the select() method of the TextRange object can be used to display the text insertion pointer where your script desires. Therefore, through the TextRange object you can script your forms to always display the text insertion pointer at the end of existing text in a text box or textarea when the control receives focus.

Working with text ranges

To create a TextRange object, use the createTextRange() method with the document.body object or any button- or text-related form control object. If you want to convert a block of selected text to a text range, use the special createRange() method of the document.selection object. Regardless of how you create it, the range encompasses the entire text of the object used to generate the range. In other words, the start point is at the very beginning of the text and the end point is at the very end. Note that when you create a TextRange object from the BODY element, text that is inside text-related form controls is not part of the

text of the TextRange (just as text field content isn't selected if you select manually the entire text of the page).

After you create a TextRange object (assigned to a variable), the typical next steps involve some of the many methods associated with the object that help narrow the size of the range. Some methods (move(), moveEnd(), moveStart(), and sentEndPoint()) offer manual control over the intra-character position for the start and end points. Parameters of some of these methods understand concepts, such as words and sentences, so not every action entails tedious character counts. Another method, moveToElementText(), automatically adjusts the range to encompass a named element. The oft-used collapse() method brings the start and end points together at the beginning or end of the current range — helpful when a script must iterate through a range for tasks, such as word counting or search and replace. The expand() method can extend a collapsed range to encompass the whole word, whole sentence, or entire range surrounding the insertion point. Perhaps the most powerful method is findText(), which allows scripts to perform single or global search and replace operations on body text.

After the range encompasses the desired text, several other methods let scripts act on the selection. The types of operations include scrolling the page to make the text represented by the range visible to the user (scrollIntoView()) and selecting the text (select()) to provide visual feedback to the user that something is happening (or to set the insertion pointer at a location in a text form control). An entire library of additional commands are accessed through the execCommand() method for operations, such as copying text to the clipboard and a host of formatting commands that can be used in place of style sheet property changes. To swap text from the range with new text accumulated by your script, you can modify the text property of the range.

Using the TextRange object can be a bit tedious, because it often requires a number of script statements to execute an action. Three basic steps are generally required to work with a TextRange object:

1. Create the text range.
2. Set the start and end points.
3. Act on the range.

As soon as you are comfortable with this object, you will find it provides a lot of flexibility in scripting interaction with body content. For ideas about applying the TextRange object in your scripts, see the examples that accompany the following descriptions of individual properties and methods.

About browser compatibility

The TextRange object is available only for the Windows 9*x*/NT version of IE4 and IE5. MacOS versions through IE5 do not support the TextRange object.

The W3C DOM and NN6 implement a slightly different concept of text ranges in what they call the Range object. In many respects, the fundamental way of working with a Range object is the same as for a TextRange object: create, adjust start and end points, and act on the range. But the W3C version (like the W3C DOM itself) is more conscious of the node hierarchy of a document. Properties and methods of the W3C Range object reflect this node-centric point of view, so that most of the terminology for the Range object differs from that of the IE TextRange object. As of this writing, it is unknown if or when IE will implement the W3C Range object.

At the same time, the W3C Range object lacks a couple of methods that are quite useful with the IE TextRange object — notably findText() and select(). On the other hand, the Range object, as implemented in NN6, works on all OS platforms.

The bottom line question, then, is whether you can make range-related scripts work in both browsers. While the basic sequence of operations is the same for both objects, the scripting vocabulary is quite different. Table 19-1 presents a summary of the property and method behaviors that the two objects have in common and their respective vocabulary terms (sometimes the value of a property in one object is accessed via a method in the other object). Notice that the ways of moving individual end points are not listed in the table because the corresponding methods for each object (for example, moveStart() in TextRange versus setStart() in Range) use very different spatial paradigms.

Table 19-1
TextRange versus Range Common Denominators

TextRange Object	*Range Object*
text	toString()
collapse()	collapse()
compareEndPoints()	compareEndPoints()
duplicate()	clone()
moveToElementText()	selectContents()
parentElement()	commonParent

To blend text range actions for both object models into a single scripted page, you have to include script execution branches for each category of object model or create your own API to invoke library functions that perform the branching. On the IE side of things, too, you have to script around actions that can cause script errors when run on MacOS and other non-Windows versions of the browser.

Properties

boundingHeight
boundingLeft
boundingTop
boundingWidth

Value: Integer Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Every text range has physical dimension and location on the page, even if you cannot see the range reflected graphically with highlighting. Even a text insertion pointer (meaning a collapsed text range) has a rectangle whose height equals the line height of the body text in which the insertion point resides; its width, however, is zero.

The pixel dimensions of the rectangle of a text range can be retrieved via the `boundingHeight` and `boundingWidth` properties of the `TextRange` object. When a text range extends across multiple lines, the dimensions of the rectangle are equal to the smallest single rectangle that can contain the text (a concept identical to the bounding rectangle of inline body text, as described in the `TextRectangle` object later in this chapter). Therefore, even a range consisting of one character at the end of one line and another character at the beginning of the next, force the bounding rectangle to be as wide as the paragraph element.

A text range rectangle has a physical location on the page. The top-left position of the rectangle (with respect to the browser window edge) is reported by the `boundingTop` and `boundingLeft` properties. In practice, text ranges that are generated from selections can report very odd `boundingTop` values in IE4 when the page scrolls. Use the `offsetTop` and `offsetLeft` properties for more reliable results.

Example (with Listing 19-9) on the CD-ROM

Related Items: `offsetLeft`, `offsetTop` properties of element objects (Chapter 15).

htmlText

Value: String Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `htmlText` property returns the HTML of the text contained by a text range. If a range's start and end points are at the very edges of an element's text, then the HTML tag for that element becomes part of the `htmlText` property value. Also, if the range starts in one element and ends partway in another, the tags that influence the text of the end portion become part of the `htmlText`. This property is read-only, so you cannot use it to insert or replace HTML in the text range (see the `pasteHTML()` method and various insert commands associated with the `execCommand()` method in the following section).

Example on the CD-ROM

Related Items: `text` property.

text

Value: String Read/Write

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Use the `text` property to view or change the string of visible characters that comprise a text range. The browser makes some decisions for you if the range you are about to change has nested elements inside. By and large, the nested element (and any formatting that may be associated with it) is deleted, and the new text becomes

part of the text of the container that houses the start point of the text range. By the same token, if the range starts in the middle of one element and ends in the parent element's text, the tag that governs the start point now wraps all of the new text.

Example on the CD-ROM

Related Items: `htmlText` property.

Methods

`collapse([startBoolean])`

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Use the `collapse()` method to shrink a text range from its current size down to a single insertion point between characters. This method becomes more important than you may think at first, especially in a function that is traversing the body or large chunk of text. For example, in a typical looping word-counting script, you create a text range that encompasses the full body (or text in a TEXTAREA). When the range is created, its start point is at the very beginning of the text, and its end point is at the very end. To begin counting words, you can first collapse the range to the insertion point at the very beginning of the range. Next, use the `expand()` method to set the range to the first word of text (and increment the counter if the `expand()` method returns `true`). At that point, the text range extends around the first word. What you want is for the range to collapse at the end of the current range so that the search for the next word starts after the current one. Use `collapse()` once more, but this time with a twist of parameters.

The optional parameter of the `collapse()` method is a Boolean value that directs the range to collapse itself either at the start or end of the current range. The default behavior is the equivalent of a value of `true`, which means that unless otherwise directed, a `collapse()` method shifts the text range to the point in front of the current range. That works great as an early step in the word-counting example, because you want the text range to collapse to the start of the text before doing any counting. But for subsequent movements through the range, you want to collapse the range so that it is after the current range. Thus, you include a `false` parameter to the `collapse()` method.

Example on the CD-ROM

Related Items: `Range.collapse()`, `TextRange.expand()` methods.

`compareEndPoints("type", rangeRef)`

Returns: Integer (`-1`, `0`, or `1`).

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

Generating multiple `TextRange` objects and assigning them to different variables is no problem. You can then use the `compareEndPoints()` method to compare the relative positions of start and end points of two ranges. One range is the object that you use to invoke the `compareEndPoints()` method, and the other range is the second parameter of the method. The order doesn't matter, because the first parameter of the method determines which points in each range you will be comparing.

Values for the first parameter can be one of four explicit strings: `StartToEnd`, `StartToStart`, `EndToStart`, and `EndToEnd`. What these values specify is which point of the current range is compared with which point of the range passed as the second parameter. For example, consider the following body text that has two text ranges defined within it:

```
It was the best of times.
```

The first text range (assigned in our discussion here to variable `rng1`) is shown in boldface, while the second text range (`rng2`) is shown in bold-italic. In other words, `rng2` is nested inside `rng1`. We can compare the position of the start of `rng1` against the position of the start of `rng2` by using the `StartToStart` parameter of the `compareEndPoints()` method:

```
var result = rng1.compareEndPoints("StartToStart", rng2)
```

The value returned from the `compareEndPoints()` method is an integer of one of three values. If the positions of both points under test are the same, then the value returned is `0`. If the first point is before the second, the value returned is `-1`; if the first point is after the second, the value is `1`. Therefore, from the example above, because the start of `rng1` is before the start of `rng2`, the method returns `-1`. If you changed the statement to invoke the method on `rng2`, as in

```
var result = rng2.compareEndPoints("StartToStart", rng1)
```

the result would be 1.

In practice, this method is helpful in knowing if two ranges are the same, if one of the boundary points of both ranges is the same, or if one range starts where the other ends.

Example (with Listing 19-10) on the CD-ROM

Related Items: `Range.compareEndPoints()` method.

duplicate()

Returns: `TextRange` object.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `duplicate()` method returns a `TextRange` object that is a snapshot copy of the current `TextRange` object. In a way, a non-intuitive relationship exists between the two objects. If you alter the `text` property of the copy without moving the start or end points of the original, then the original takes on the new text. But if you move the start or end points of the original, the `text` and `htmlText` of the original obviously change, while the copy retains its properties from the time of the duplication. Therefore, this method can be used to clone text from one part of the document to other parts.

Example on the CD-ROM

Related Items: `Range.clone()`, `TextRange.isEqual()` methods.

execCommand("*commandName*"[, *UIFlag*[, *value*]])

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

IE4+ for Win32 operating systems lets scripts access a very large number of commands that act on insertion points, abstract text ranges, and selections that are made with the help of the `TextRange` object. Access to these commands is through the `execCommand()` method, which works with `TextRange` objects and the `document` object (see the `document.execCommand()` method discussion in Chapter 18 and list of document- and selection-related commands in Table 18-3).

The first, required parameter is the name of the command that you want to execute. Only a handful of these commands offer unique capabilities that aren't better accomplished within the IE object model and style sheet mechanism. Of particular importance is the command that lets you copy a text range into the Clipboard. Most of the rest of the commands modify styles or insert HTML tags at the position of a collapsed text range. These actions are better handled by other means, but they are included in Table 19-2 for the sake of completeness only (see Table 18-3 for additional commands).

Table 19-2
TextRange.execCommand() Commands

Command	*Parameter*	*Description*
`Bold`	None	Encloses the text range in a `<B>` tag pair
`Copy`	None	Copies the text range into the Clipboard
`Cut`	None	Copies the text range into the Clipboard and deletes it from the document or text control
`Delete`	None	Deletes the text range
`InsertButton`	ID String	Inserts a `<BUTTON>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertFieldset`	ID String	Inserts a `<FIELDSET>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertHoritontalRule`	ID String	Inserts an `<HR>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertIFrame`	ID String	Inserts an `<IFRAME>` tag at the insertion point, assigning the parameter value to the `ID` attribute

Command	*Parameter*	*Description*
`InsertInputButton`	ID String	Inserts an `<INPUT TYPE="button">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertIntpuCheckbox`	ID String	Inserts an `<INPUT TYPE="checkbox">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputFileUpload`	ID String	Inserts an `<INPUT TYPE="FileUpload">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputHidden`	ID String	Inserts an `<INPUT TYPE="hidden">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputImage`	ID String	Inserts an `<INPUT TYPE="image">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputPassword`	ID String	Inserts an `<INPUT TYPE="password">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputRadio`	ID String	Inserts an `<INPUT TYPE="radio">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputReset`	ID String	Inserts an `<INPUT TYPE="reset">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertInputSubmit`	ID String	Inserts an `<INPUT TYPE="submit">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertIntputText`	ID String	Inserts an `<INPUT TYPE="text">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertMarquee`	ID String	Inserts a `<MARQUEE>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertOrderedList`	ID String	Inserts an `<OL>` tag at the insertion point, assigning the parameter value to the `ID` attribute

Continued

Table 19-2 *(continued)*

Command	*Parameter*	*Description*
`InsertParagraph`	ID String	Inserts a `<P>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertSelectDropdown`	ID String	Inserts a `<SELECT TYPE="select-one">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertSelectListbox`	ID String	Inserts a `<SELECT TYPE="select-multiple">` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertTextArea`	ID String	Inserts an empty `<TEXTAREA>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`InsertUnroderedList`	ID String	Inserts a `<UL>` tag at the insertion point, assigning the parameter value to the `ID` attribute
`Italic`	None	Encloses the text range in an `<I>` tag pair
`OverWrite`	Boolean	Sets the text input control mode to overwrite (`true`) or insert (`false`)
`Paste`	None	Pastes the current Clipboard contents into the insertion point or selection
`PlayImage`	None	Begins playing dynamic images if they are assigned to the `DYNSRC` attribute of the IMG element
`Refresh`	None	Reloads the current page
`StopImage`	None	Stops playing dynamic images if they are assigned to the `DYNSRC` attribute of the IMG element
`Underline`	None	Encloses the text range in a `<U>` tag pair

An optional second parameter is a Boolean flag to instruct the command to display any user interface artifacts that may be associated with the command. The default is `false`. For the third parameter, some commands require an attribute value for the command to work. For example, insert a new paragraph at an insertion point, you pass the identifier to be assigned to the `ID` attribute of the P element. The syntax is

```
myRange.execCommand("InsertParagraph", true, "myNewP")
```

The `execCommand()` method returns Boolean `true` if the command is successful; `false` if not successful. Some commands can return values (for example, finding out the font name of a selection), but these values are accessed through the `queryCommandValue()` method.

While the commands in Table 19-2 work on text ranges, even the commands that work on selections (Table 18-3) can frequently benefit from some preprocessing with a text range. Consider, for example, a function whose job it is to find every instance of a particular word in a document and set its background color to a yellow highlight. Such a function utilizes the powers of the `findText()` method of a text range to locate each instance. Then the `select()` method selects the text in preparation for applying the `BackColor` command. Here is a sample:

```
function hiliteIt(txt) {
    var rng = document.body.createTextRange()
    for (var i = 0; rng.findText(txt); i++) {
        rng.select()
        rng.execCommand("BackColor", "false", "yellow")
        rng.execCommand("Unselect")
        // prepare for next search
        rng.collapse(false)
    }
}
```

This example is a rare case that makes the `execCommand()` method way of modifying HTML content more efficient than trying to wrap some existing text inside a new tag. The downside is that you don't have control over the methodology used to assign a background color to a span of text (in this case, IE wraps the text in a `<FONT>` tag with a `STYLE` attribute set to `BACKGROUND-COLOR:yellow` — probably not the way you'd do it on your own).

Example on the CD-ROM

Related Items: Several query command methods of the `TextRange` object.

expand("*unit*")

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The single `expand()` method can open any range — collapsed or not — to the next largest character, word, or sentence or to the entire original range (for example, encompassing the text of the BODY element if the range was generated by `document.body.createTextRange()`). The parameter is a string designating which unit to expand outward to: `character`, `word`, `sentence`, or `textedit`. If the operation is successful, the method returns `true`; otherwise it returns `false`.

When operating from an insertion point, the `expand()` method looks for the word or sentence that encloses the point. The routine is not very smart about sentences, however. If you have some text prior to a sentence that you want to expand to, but that text does not end in a period, the `expand()` routine expands outward until it can find either a period or the beginning of the range. Listing 15-14 demonstrates a workaround for this phenomenon. When expanding from an insertion point to a character, the method expands forward to the next character in language order. If the insertion point is at the end of the range, it cannot expand to the next characters, and the `expand()` method returns `false`.

It is not uncommon in an extensive script that needs to move the start and end points all over the initial range to perform several `collapse()` and `expand()` method operations from time to time. Expanding to the full range is a way to start some range manipulation with a clean slate, as if you just created the range.

Example on the CD-ROM

Related Items: `TextRange.collapse()` method.

findText("*searchString*"[, *searchScope*, *flags*])

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

One of the most useful methods of the `TextRange` object is `findText()`, whose default behavior is to look through a text range starting at the range's start point up to the end of the range in search of a case-insensitive match for a search string. If an instance is found in the range, the start and end points of the range are cinched up to the found text and the method returns `true`; otherwise it returns `false`, and the start and end points do not move. Only the rendered text is searched and not any of the tags or attributes.

Optional parameters let you exert some additional control over the search process. You can restrict the distance from a collapsed range to be used for searching. The *searchScope* parameter is an integer value indicating the number of characters from the start point. The larger the number, the more text of the range is included in the search. Negative values force the search to operate backward from the current start point. If you want to search backward to the beginning of the range, but you don't know how far away the start of the range is, you can specify an arbitrarily huge number that would encompass the text.

The optional *flags* parameter lets you set whether the search is to be case-sensitive and/or to match whole words only. The parameter is a single integer value that uses bit-wise math to calculate the single value that accommodates one or both settings. The value for matching whole words is `2`; the value for matching case is `4`. If you want only one of those behaviors, then supply just the desired number. But for both behaviors, use the bit-wise XOR operator (the `^` operator) on the values to reach a value of `6`.

The most common applications of the `findText()` method include a search-and-replace action and format changes to every instance of a string within the range. This iterative process requires some extra management of the process. Because searching always starts with the range's current start point, advancing the start point to the end of the text found in the range is necessary. This advancing allows a successive application of `findText()` to look through the rest of the range for another match. And because `findText()` ignores the arbitrary end point of the current range and continues to the end of the initial range, you can use the `collapse(false)` method to force the starting point to the end of the range that contains the first match.

A repetitive search can be accomplished by a `while` or `for` repeat loop. The Boolean returned value of the `findText()` method can act as the condition for continuing the loop. If the number of times the search succeeds is not essential to your script, a `while` loop works nicely:

```
while (rng.findText(searchString)) {
    ...
    rng.collapse(false)
}
```

Or you can use a `for` loop counter to maintain a count of successes, such as a counter of how many times a string appears in the body:

```
for (var i = 0; rng.findText(searchString); i++) {
    ...
    rng.collapse(false)
}
```

Some of the operations you want to perform on a range (such as many of the commands invoked by the `execCommand()` method) require that a selection exists for the command to work. Be prepared to use the `select()` method on the range after the `findText()` method locates a matching range of text.

Example (with Listing 19-11) on the CD-ROM

Related Items: `TextRange.select()` method.

getBookmark()

Returns: Bookmark String.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

In the context of a `TextRange` object, a bookmark is not to be confused with the kinds of bookmarks you add to a browser list of favorite Web sites. Instead, a bookmark is a string that represents a definition of a text range, including its location in a document, its text, and so on. Viewing the string is futile, because it contains string versions of binary data, so the string means nothing in plain language. But a bookmark allows your scripts to save the current state of a text range so that it may be restored at a later time. The `getBookmark()` method returns the string representation of a snapshot of the current text range. Some other script statement can adjust the `TextRange` object to the exact specifications of the snapshot with the `moveToBookmark()` method (described later in this chapter).

Example on the CD-ROM

Related Items: `TextRange.moveToBookmark()` method.

inRange(*otherRangeRef*)

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

You can compare the physical stretches of text contained by two different text ranges via the `inRange()` method. Typically, you invoke the method on the larger of two ranges and pass a reference to the smaller range as the sole parameter to the method. If the range passed as a parameter is either contained by or equal to the text range that invokes the method, then the method returns `true`; otherwise the method returns `false`.

Example on the CD-ROM

Related Items: `TextRange.isEqual()` method.

`isEqual(`*otherRangeRef*`)`

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

If your script has references to two independently adjusted `TextRange` objects, you can use the `isEqual()` method to test whether the two objects are identical. This method tests for a very literal equality, requiring that the text of the two ranges be character-for-character and position-for-position equal in the context of the original ranges (for example, body or text control content). To see if one range is contained by another, use the `inRange()` method instead.

Example on the CD-ROM

Related Items: `TextRange.inRange()` method.

`move("`*unit*`"[, `*count*`])`

Returns: Integer.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `move()` method performs two operations. First, the method collapses the current text range to become an insertion point at the location of the previous end

point. Next, it moves that insertion point to a position forward or backward any number of character, word, or sentence units. The first parameter is a string specifying the desired unit (`character`, `word`, `sentence`, or `textedit`). A value of `textedit` moves the pointer to the beginning or end of the entire initial text range. If you omit the second parameter, the default value is `1`. Otherwise you can specify an integer indicating the number of units the collapsed range should be moved ahead (positive integer) or backward (negative). The method returns an integer revealing the exact number of units the pointer is able to move — if you specify more units than are available, the returned value lets you know how far it can go.

Bear in mind that the range is still collapsed after the `move()` method executes. Expanding the range around desired text is the job of other methods.

You can also use the `move()` method in concert with the `select()` method to position the flashing text insertion pointer within a text box or textarea. Thus, you can script a text field, upon receiving focus or the page loading, to have the text pointer waiting for the user at the end of existing text. A generic function for such an action is shown in the following:

```
function setCursorToEnd(elem) {
    if (elem) {
        if (elem.type && (elem.type == "text" || elem.type == "textarea")) {
            var rng = elem.createTextRange()
            rng.move("textedit")
            rng.select()
        }
    }
}
```

You can then invoke this method from a text field's `onFocus` event handler:

```
<INPUT TYPE="text" ... onFocus="setCursorToEnd(this)">
```

The function previously shown includes a couple of layers of error checking, such as making sure that the function is invoked with a valid object as a parameter and that the object has a `type` property whose value is one capable of having a text range made for its content.

Example on the CD-ROM

Related Items: `TextRange.moveEnd()`, `TextRange.moveStart()` methods.

moveEnd("*unit*"[, *count*])
moveStart("*unit*"[, *count*])

Returns: Integer.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `moveEnd()` and `moveStart()` methods are similar to the `move()` method, but they each act only on the end and starting points of the current range, respectively. In other words, the range does not collapse before the point is moved. These methods allow you to expand or shrink a range by a specific number of units by moving only one of the range's boundaries.

The first parameter is a string specifying the desired unit (`character`, `word`, `sentence`, or `textedit`). A value of `textedit` moves the pointer to the beginning or end of the entire initial text range. Therefore, if you want the end point of the current range to zip to the end of the body (or text form control), use `moveEnd("textedit")`. If you omit the second parameter, the default value is `1`. Otherwise you can specify an integer indicating the number of units the collapsed range is to move ahead (positive integer) or backward (negative). Moving either point beyond the location of the other forces the range to collapse and move to the location specified by the method. The method returns an integer revealing the exact number of units the pointer is able to move — if you specify more units than are available, the returned value lets you know how far it can go.

Example on the CD-ROM

Related Items: `TextRange.move()` method.

moveToBookmark("*bookmarkString*")

Returns: Boolean.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

If a snapshot of a text range specification has been preserved in a variable (with the help of the `getBookmark()` method), the `moveToBookmark()` method uses that bookmark string as its parameter to set the text range to exactly the way it appeared when the bookmark was originally obtained. If the method is successful, it returns a value of `true`, and the text range is set to the same string of text as originally preserved via `getBookmark()`. It is possible that the state of the content of

the text range has been altered to such an extent that resurrecting the original text range is not feasible. In that case, the method returns `false`.

Example on the CD-ROM

Related Items: `TextRange.getBookmark()` method.

moveToElementText(*elemObjRef*)

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The fastest way to cinch up a text range to the boundaries of an HTML element on the page is to use the `moveToElementText()` method. Any valid reference to the HTML element object is accepted as the sole parameter — just don't try to use a string version of the object ID unless it is wrapped in the `document.getElementById()` method (IE5+). When the boundaries are moved to the element, the range's `htmlText` property contains the tags for the element.

Example on the CD-ROM

Related Items: `TextRange.parentElement()` method.

moveToPoint(*x*, *y*)

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `moveToPoint()` method shrinks the current text range object to an insertion point and then moves it to a position in the current browser window or frame. You control the precise position via the *x* (horizontal) and *y* (vertical) pixel coordinates specified as parameters. The position is relative to the visible window, and not the document, which may have been scrolled to a different position. Invoking

the `moveToPoint()` method is the scripted equivalent of the user clicking that spot in the window. Use the `expand()` method to flesh out the collapsed text range to encompass the surrounding character, word, or sentence.

Using the `moveToPoint()` method on a text range defined for a text form control may cause a browser crash. The method appears safe with the `document.body` text ranges, even if the x,y position falls within the rectangle of a text control. Such a position, however, does not drop the text range into the form control or its content.

Example on the CD-ROM

Related Items: `TextRange.move()`, `TextRange.moveStart()`, `TextRange.moveEnd()` methods.

parentElement()

Returns: Element object reference.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `parentElement()` method returns a reference to the next outermost HTML element container that holds the text range boundaries. If the text range boundaries are at the boundaries of a single element, the `parentElement()` method returns that element's reference. But if the boundaries straddle elements, then the object returned by the method is the element that contains the text of the least-nested text portion. In contrast to the `expand()` and various move-related methods, which understand text constructs, such as words and sentences, the `parentElement()` method is concerned solely with element objects. Therefore, if a text range is collapsed to an insertion point in body text, you can expand it to encompass the HTML element by using the `parentElement()` method as a parameter to `moveToElementText()`:

```
rng.moveToElementText(rng.parentElement())
```

Example on the CD-ROM

Related Items: `TextRange.expand()`, `TextRange.move()`, `TextRange.moveEnd()`, `TextRange.moveStart()` methods.

pasteHTML("*HTMLText*")

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

While the `execCommand()` method offers several commands that insert HTML elements into a text range, it is probably more convenient to simply paste fully formed HTML into the current text range (assuming you need to be working with a text range instead of even more simply setting new values to an element object's `outerHTML` property). Provide the HTML to be inserted as a string parameter to the `pasteHTML()` method.

Use the `pasteHTML()` method with some forethought. Some HTML that you may attempt to paste into a text range may force the method to wrap additional tags around the content you provide to ensure the validity of the resulting HTML. For example, if you were to replace a text range consisting of a portion of text of a P element with, for instance an LI element, the `pasteHTML()` method has no choice but to divide the P element into two pieces, because a P element is not a valid container for a solo LI element. This division can greatly disrupt your document object hierarchy, because the divided P element assumes the same ID for both pieces. Existing references to that P element will break, because the reference now returns an array of two like-named objects.

Example on the CD-ROM

Related Items: `outerHTML property`; `insertAdjacenHTML()` method.

queryCommandEnabled("*commandName*")
queryCommandIndeterm("*commandName*")
queryCommandState("*commandName*")
queryCommandSupported("*commandName*")
queryCommandText("*commandName*")
queryCommandValue("*commandName*")

Returns: See `document.queryCommandEnabled()` in Chapter 18.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

See descriptions under `document.queryCommandEnabled()` in Chapter 18.

`select()`

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

The `select()` method selects the text inside the boundaries of the current text range. For some operations, such as prompted search and replace, it is helpful to show the user the text of the current range to highlight what text is about to be replaced. In some other operations, especially several commands invoked by `execCommand()`, the operation works only on a text selection in the document. Thus, you can use the `TextRange` object facilities to set the boundaries, followed by the `select()` method to prepare the text for whatever command you like. Text selected by the `select()` method becomes a `selection` object (covered earlier in this chapter).

Example on the CD-ROM

Related Items: `selection` object.

`setEndPoint("type", otherRangeRef)`

Returns: Nothing.

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility							✓	✓	✓

In contrast to the `moveEnd()` method, which adjusts the end point of the current range with respect to characters, words, sentences, and the complete range, the

`setEndPoint()` method sets a boundary of the current range (not necessarily the ending boundary) relative to a boundary of another text range whose reference is passed as the second parameter. The first parameter is one of four types that control which boundary of the current range is to be adjusted and which boundary of the other range is the reference point. Table 19-3 shows the four possible values and their meanings.

Table 19-3
setEndPoint() Method Types

Type	*Description*
`StartToEnd`	Moves the start point of the current range to the end of the other range
`StartToStart`	Moves the start point of the current range to the start of the other range
`EndToStart`	Moves the end point of the current range to the start of the other range
`EndToEnd`	Moves the end point of the current range to the end of the other range

Note that the method moves only one boundary of the current range at a time. If you want to make two ranges equal to each other, you have to invoke the method twice, once with `StartToStart` and once with `EndToEnd`. At that instant, the `isEqual()` method applied to those two ranges returns `true`.

Setting a boundary point with the `setEndPoint()` method can have unexpected results when the revised text range straddles multiple elements. Don't be surprised to find that the new HTML text for the revised range does not include tags from the outer element container.

Example on the CD-ROM

Related Items: `TextRange.moveEnd()`, `TextRange.moveStart()`, `TextRange.moveToElementText()` methods.

TextRectangle Object

Properties	*Methods*	*Event Handlers*
`bottom`		
`left`		

Properties	Methods	Event Handlers
right		
top		

Syntax

Accessing `TextRectangle` object properties:

```
[window.]document.all.elemID.getBoundingClientRect().property
[window.]document.all.elemID.getClientRects()[i].property
```

About this object

The IE5+ `TextRectangle` object (not implemented in IE5/Mac) exposes to scripts a concept that is described in the HTML 4.0 specification, whereby an element's rendered text occupies a rectangular space on the page just large enough to contain the text. For a single word, the rectangle is as tall as the line height for the font used to render the word and no wider than the space occupied by the text. But for a sequence of words that wraps to multiple lines, the rectangle is as tall as the line height times the number of lines and as wide as the distance between the leftmost and rightmost character edges, even if it means that the rectangle encloses some other text that is not part of the element.

If you extract the `TextRectangle` object for an element by way of, for example, the `getBoundingClientRect()` method, be aware that the object is but a snapshot of the rectangle when the method was invoked. Resizing the page may very well alter dimensions of the actual rectangle enclosing the element's text, but the `TextRectangle` object copy that you made previously does not change its values to reflect the element's physical changes. After a window resize or modification of body text, any dependent `TextRectangle` objects should be recopied from the element.

Properties

bottom
left
right
top

Values: Integers Read-Only

	NN2	NN3	NN4	NN6	IE3/J1	IE3/J2	IE4	IE5	IE5.5
Compatibility								✓	✓

The screen pixel coordinates of its four edges define every `TextRectangle` object. These coordinates are relative to the window or frame displaying the page. Therefore, if you intend to align a positioned element with an inline element's `TextRectangle`, your position assignments must take into account the scrolling of the body.

To my eye, the left edge of a `TextRectangle` does not always fully encompass the left-most pixels of the rendered text. You may have to fudge a few pixels in the measure when trying to align a real element with the `TextRectangle` of another element.

Example (with Listing 19-12) on the CD-ROM

Related Items: `getBoundingClientRect()`, `getClientRects()` methods of element objects (Chapter 15).

✦ ✦ ✦