Dynamic HTML

Communicator 4.0, August 1997

.


Recycled and Recyclable Paper

Version 4.0

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Dynamic HTML in Netscape Communicator

Contents

This book describes how to use Dynamic HTML to incorporate style sheets, positioned content, and downloadable fonts in your web pages.

About This Guide

This guide discusses the concept and use of Dynamic HTML, which includes style sheets, content positioning, and downloadable fonts.

Purpose of This Document

This document is for content developers who wish to have more control over the layout and appearance of their web page, and who wish to incorporate animations using HTML and JavaScript.

This document discusses each of the three components of Dynamic HTML, describes how to use them, and gives examples of the use of each one.

Structure of This Document

This document is divided into three parts, one for each major component of Dynamic HTML.

Part 1. Style Sheets contains the following chapters:

Chapter 2, "Introduction To Style Sheets," introduces style sheets, discusses the two kinds of syntax you can use to define them, gives an introductory example, and discusses the concept of style inheritance.

Chapter 3, "Creating Style Sheets and Assigning Styles," discusses the different ways to define styles and apply them to content elements.

Chapter 4, "Format Properties for Block-Level Elements," discusses the border and format characteristics you can set for block-level elements.

Chapter 5, "Style Sheet Reference," lists the tags and attributes that pertain to style sheets, and lists all the properties you can define for styles.

Chapter 6, "Advanced Style Sheet Example," presents and discusses a web page that makes extensive use of style sheets.

Part 2. Positioning HTML Content contains the following chapters:

Chapter 7, "Introduction," introduces the concept of positioning HTML content and discusses the two kinds of syntax you can use to create positioned blocks of content.

Chapter 8, "Defining Positioned Blocks of HTML Content," discusses absolute versus relative positioning, lists the attributes and properties you can use for creating positioned blocks of content, discusses the `<NOLAYER>` tag, and summarizes the behavior of applets, plug-ins, and forms in positioned blocks of content.

Chapter 9, "Using JavaScript With Positioned Content," discusses how to use JavaScript to create and modify positioned blocks of content.

Chapter 10, "Fancy Flowers Farm Example," illustrates how to how to hide and show blocks of HTML content. It uses a pop-up menu to pick which block to display.

Chapter 11, "Swimming Fish Example," presents an example in two parts. The first part illustrates how to position and move blocks of content. The second part illustrates how to change the stacking order of the blocks.

Chapter 11, "Nikki's Diner Example," illustrates a simple use of using external files as the source for a positioned block of content.

Chapter 12, "Expanding Colored Squares Example," illustrates how to expand and contract the clipping region of a positioned block of content, without changing the wrapping width of the block.

Chapter 13, "Changing Wrapping Width Example," illustrates how to capture mouse events for a block of content and how to change the wrapping width of a block. It provides the basic groundwork for making "draggable" blocks of content.

Part 3. Downloadable Fonts contains the following single chapter:

Chapter 14, "Using Downloadable Fonts," discusses why you would use downloadable fonts and how to use them.

Typographic Conventions

The following conventions are used throughout this guide:

- Code identifiers that express literal JavaScript and HTML syntax appear in a monospaced font like this: `computer voice`.

- Italic font is used for emphasis and to indicate a special term like this: *special term*.
- Variable names are presented in italic like this: *variable*.

Chapter

1

Introducing Dynamic HTML

Navigator 4 from Netscape, which is part of the Communicator product suite, includes three new areas of functionality that taken together give you Dynamic HTML. The three components of Dynamic HTML are style sheets, content positioning, and downloadable fonts. Used together, these three components give you greater control over the appearance, layout, and behavior of your web pages.

This chapter contains the following sections:

Style sheets let you specify the stylistic attributes of the typographic elements of your web page. With content positioning, you can ensure that pieces of content are displayed on the page exactly where you want them to appear, and you can modify their appearance and location after the page has been displayed. With downloadable fonts, you can use the fonts of your choice to enhance the appearance of your text. Then you can package the fonts with the page so that the text is always displayed with your chosen fonts.

Introducing Style Sheets

Prior to the introduction of style sheets for HTML documents, web page authors had limited control over the presentation of their web pages. For example, you could specify that certain text should be displayed as headings, but you could not set margins for your pages or specify the line heights or border decoration for text.

Style sheets give you greater control over the presentation of your web documents. Using style sheets, you can specify many stylistic attributes of your web page, such as text color, margins, alignment of elements, font styles, font sizes, font weights and more. You can use borders to make certain elements stand out from the body of the content. You can specify different fonts to use for different elements, such as paragraphs, headings, and blockquotes. You can guarantee that your chosen fonts will be available on all systems by packaging them as downloadable fonts and attaching them to the web page.

In addition, you can use a style sheet as a template or "master page" so that multiple pages can use the same style sheet.

Part 1. Style Sheets, discusses the two kinds of syntax you can use for defining styles; describes how to define and use styles; discusses how to define border characteristic for block-level elements; gives the list of style properties; and presents an advanced example of the use of styles.

Introducing Content Positioning

No longer are you constrained to use sequential content laid out linearly in your web pages. By specifying positions for blocks of HTML content, you can decide what contents goes where on the page, instead of leaving it up to the browser to lay it out for you. You could, for example, place one block of content in the top-left corner of the page, and another block in the bottom-right corner. Blocks of content can share space too, so images and text can overlap. You decide precisely where each part of the content will appear, and Navigator 4 will lay your page out exactly as you want.

Using JavaScript, you can change the layout of your page dynamically, and you can modify the page in a variety of ways after the user has opened it. You can make content vanish or appear, and you can change the color of individual parts of your page. You can incorporate animation into your web pages by moving and modifying individual parts of your HTML page on the fly.

Used together, content positioning and style sheets allow you to create web pages that use different styles in different parts of the page.

Part 2. Positioning HTML Content, discusses the two kinds of syntax you can use for positioning HTML content; describes the attributes and properties you can specify for positioned content; discusses how to use JavaScript to create and modify positioned content; and gives five complete, working examples of the use of positioned content.

Introducing Downloadable Fonts

Using downloadable fonts, you can attach specific fonts to your web page. As a result, your page will always be displayed with the fonts you picked out for it. No longer need you use generic fonts to make your pages look approximately similar on each platform. No longer are you subject to the vagaries of platform-specific fonts, because a downloadable font can be displayed on any platform.

To protect the rights of the font designers, the downloadable fonts are locked so that users cannot copy them and use them again. You can include your own fonts in your web documents without worrying that your readers may copy them for their own purposes.

Whether you apply font attributes directly to a piece of text or use style sheets to define the font family for different kinds of elements, you can use downloadable fonts in your web page to guarantee that the user sees your page as you want it to be seen.

Part 3. Downloadable Fonts, discusses how to create and use downloadable fonts, and how to attach them to your web page.

Part 1.Style Sheets

Contents

Chapter

2

Introduction To Style Sheets

This chapter introduces the use of style sheets in Netscape Communicator. It gives an overview of the two different types of syntax you can use to define styles, presents an introductory example of the use of styles, and explains about style inheritance,

Style Sheets in Communicator

Prior to the introduction of style sheets for HTML documents, web page authors had limited control over the presentation of their web pages. For example, you could specify text to be displayed as headings, but you could not set margins for your pages or specify the line heights or margins for text.

Style sheets give you greater control over the presentation of your web documents. Using style sheets, you can specify many stylistic attributes of your web page, such as text color, margins, element alignments, font styles, font sizes, font weights and more.

Netscape Communicator supports two types of style sheet syntax. It supports style sheets written in cascading style sheet (CSS) syntax. It also supports style sheets written in JavaScript that use the document object model. In the document object model, a document is an object that has properties. Each property can in turn be an object that has further properties, and so on.

When you define a style sheet, you must declare its type as either `"text/CSS"` or `"text/JavaScript"`. To try to keep things straight, this manual uses the term *CSS syntax* to refer to the syntax for style sheets whose type is `"text/CSS".` It uses the term *JavaScript syntax* to refer to the syntax for style sheets whose type is `"text/JavaScript".`

Using Cascading Style Sheets to Define Styles

Netscape Communicator fully supports cascading style sheets. Web pages that use cascading style sheets will be displayed appropriately in Netscape Communicator with a few minor exceptions.

This document describes the style sheet functionality that is implemented in Netscape Navigator 4.0. However, if you'd like to see the original specification for style sheets as authored by the World Wide Web Consortium, you can go to:

http://www.w3.org/pub/WWW/TR/REC-CSS1

A style sheet consists of a one or more style definitions. In CSS syntax, the property names and values of a style are listed inside curly braces following the selection criteria for that style.

The selection criteria determines which elements the style is applied to, or which elements it can be applied to. If the selection criteria is an HTML element, the style is applied to all instances of that element. The selection criteria can also be a class, an ID, or it can be contextual. Each of these kinds of selection criteria are discussed in this document.

Each property in the style definition is followed by a colon then by the value for that property. Each property name/value pair must be separated from the next pair by a semicolon.

For example, the following cascading style sheet defines two styles definitions. One style definition specifies that the font size for all <P> elements is 18 and the left margin for all <P> elements is 20. The other style definition specifies that the color for all <H1> elements is blue.

```
<STYLE TYPE="text/css">
<!--
  P {font-size:18pt; margin-left:20pt;}
  H1 {color:blue;}
-->
</STYLE>
```

You can include the contents of the style sheet inside a comment (<!-- ... -->) so that browsers that do not recognize the <STYLE> element will ignore it.

Important: When specifying values for cascading style sheet properties, do not include double quotes.

Cascading style sheets require strict adherence to correct syntax. Be sure not to omit any semicolons between name/value pairs. If you miss a single semicolon, the style definition will be ignored. Similarly if you accidentally include a single extraneous character anywhere within a style definition, that definition will be ignored.

Using JavaScript and the Document Object Model to Define Styles

Using JavaScript, you can define style sheets that use the document object model. In this model, you can think of a document such as a web page as an object that has properties that can be set or accessed. Each property can in turn

be an object that has further properties. For example, the following code sets the `color` property of the object in the `H1` property of the object in the `tags` property of the `document`:

```
document.tags.H1.color = "red";
```

The `tags` property always applies to the `document` object for the current document, so you can omit `document` from the expression `document.tags`.

The following example uses JavaScript and the document object model to define a style sheet that has two style definitions. One style definition specifies that the font size for all `<P>` elements (tags) is 18 and the left margin for all `<P>` elements is 20. The other style definition specifies that the color for all `<H1>` elements is blue.

```
<STYLE TYPE = "text/javascript">
 tags.P.fontSize = "18pt";
 tags.P.marginLeft = "20pt";
 tags.H1.color = "blue";
</STYLE>
```

Do not wrap the contents of the style sheet in a comment (`!-- ... -->`) for style sheets that use JavaScript syntax.

You can also use the `with (tags.element)` syntax to shorten the style specification for elements that have several style settings. The following example specifies that all `<P>` elements are displayed in green, bold, italic, Helvetica font.

```
with (tags.P) {
 color="green";
 font-weight="bold";
 font-style="italic";
 font-family="helvetica";
}
```

Introductory Example

Using style sheets, you can specify many stylistic attributes of your web page. The stylistic characteristics you can set for font and text include text alignment, text color, font family (such as Garamond), font style (such as italic), font weight (such as bold), line height, text decoration (such as underlining), horizontal and vertical alignment of text, and text indentation (which allows indented and outdented paragraphs). You can specify background colors and images for elements. You can specify the color and style to use for the bullets and numbers in lists.

You can set margins and also specify borders for block-level elements. You can set the padding for elements that have borders, to indicate the distance between the element's content and its border.

The following code shows a simple style sheet in both CSS syntax and JavaScript syntax. This style sheet specifies that all <P> elements have left and right margins, and their text is centered between the margins. All <H4> elements are green and underlined. All <H5> elements are uppercase. They have a red border that is four points thick. The border is in outdented 3D style and the padding between the text and the border is four points. The text color is red and the background color is yellow. All <BLOCKQUOTE> elements are blue italic, with a line height that is 150% larger than the font size. The first line is indented by 10% of the width of the element.

CSS Syntax

```
<STYLE TYPE="text/css">
P {
  textAlign:center; margin-left:20%; margin-right:20%;}
H4 {
  text-decoration:underline; color: green;}
H5 {
  text-transform:uppercase; color: red;
  border-width:4pt; border-style:outset;
  background-color:yellow; padding: 4pt;
  border-color:red;}
```

```
BLOCKQUOTE {
  color:blue; font-style:italic;
  line-height:1.5; text-indent:10%;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
  with (tags.P) {
    textAlign = "center"; marginLeft="20%". margin-right="20%";}
  with (tags.H4) {
    textDecoration = "underline; color = "green";
    textTransform = "uppercase;}
  with (tags.H5) {
    color = "red";
    borderWidths="4pt"=; borderStyle="outset";
    backgroundColor="yellow"; paddings("4pt");
    borderColor="red";}
  with (tags.BLOCKQUOTE) {
    color="blue"; fontStyle="italic";
    lineHeight = 1.5; textIndent = "20pt";}

</STYLE>
```

Style Sheet Use

```
<H4>Underlined Heading 4</H4>
<BLOCKQUOTE>
This is a blockquote. It is usual for blockquotes to be indented, but
the first line of this blockquote has an extra indent. Also the line
height in this blockquote is bigger than you usually see in blockquotes.
<h5>uppercase heading 5 with a border</H5>
</BLOCKQUOTE>
<P>This paragraph has a text alignment value of center. It also has
large margins, so each line is not only centered but is also inset on
both sides from the element that contains it, which in this case is the
document.</P>
```

Inheritance of Styles

An HTML element that contains another element is considered to be the parent element of the element it contains, and the element it contains is considered to be its child element.

For example, in the following HTML text, the <BODY> element is the parent of the <H1> element which in turn is the parent of the <EM> element.

```
<BODY>
<H1>The headline <EM>is</EM> important!</H1>
</BODY>
```

In many cases, child elements acquire or inherit the styles of their parent elements. For example, suppose a style has been assigned to the <H1> element as follows:

```
<STYLE type="text/css">
H1 {color:blue;}
</STYLE>
<BODY>
<H1>The headline <EM>is</EM> important!</H1>
```

In this case, the child <EM> element takes on the style of its parent, which is the <H1> element, so the word *is* appears in blue. However, suppose you had previously set up a style specifying that <EM> elements should be displayed in red. In that case, the word *is* would be displayed in red, because properties set on the child override properties inherited from the parent.

Inheritance starts at the top-level element. In HTML, this is the <HTML> element, which is followed by the <BODY> element.

To set default style properties for all elements in a document, you can specify a style for the <BODY> element. For example, the following code sets the default text color to green.

CSS Syntax

```
<STYLE TYPE="text/css">
  BODY {color: green;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
  tags.BODY.color="green";
</STYLE>
```

A few style properties are not inherited by the child element from the parent element, but in most of these cases, the net result is the same as if the property was inherited. For example, consider the background color property, which is not inherited. If a child element does not specify its own background color, then the parent's background color is visible through the child element. It will look as if the child element has the same background color as its parent element.

Chapter

3

Creating Style Sheets and Assigning Styles

This chapter looks at each of the different ways you can defines styles, and shows how to apply styles to HTML elements.

- Defining Style Sheets with the <STYLE> Tag
- Defining Style Sheets in External Files
- Defining Classes of Styles
- Defining Named Individual Styles
- Using Contextual Selection Criteria
- Combining Style Sheets

A style sheet is a series of one or more style definitions.You can define a style sheet directly inside the document that uses it, or you can define a style sheet in an external document. If the style sheet is in an external document, then it can be used by other documents. For example, a series of pages for a particular site could all use a single externally defined style sheet that sets up the house style.

If the style sheet is unlikely to be applicable to other documents, it can be more convenient to define it directly in the document that uses it, since then you have the style sheet and the content in one place.

Defining Style Sheets with the <STYLE> Tag

To define a style sheet directly inside a document, use the <STYLE> tag in the header part of your document. The <STYLE> tag opens the style sheet, and the </STYLE> tag closes the style sheet. Be sure to use the <STYLE> tag before the <BODY> tag.

When you use the <STYLE> tag, you can specify the TYPE attribute to indicate if the type is "text/css" or "text/javascript". The default value for TYPE is "text/css".

The following example defines a style sheet that specifies that all level-one headings are uppercase blue, and all blockquotes are red italic.

CSS Syntax

```
<HEAD>
<STYLE TYPE="text/css">
  H1 {color: blue; text-transform: uppercase;}
  BLOCKQUOTE {color: red; font-style: italic;}
</STYLE>
</HEAD>
<BODY>
```

JavaScript Syntax

```
<HEAD>
<STYLE TYPE="text/javascript">
    tags.H1.textTransform = "uppercase";
    tags.H1.color = "blue";
    tags.BLOCKQUOTE.color = "red";
    tags.BLOCKQUOTE.font-style: italic;
</STYLE>
</HEAD>
<BODY>
```

Style Sheet Use

```
<H1>This Heading Is Blue</H1>

<B>BLOCKQUOTE>This blockquote is displayed in red.</B>
```

Defining Style Sheets in External Files

You can define style sheets in a file that is separate from the document and then link the style sheet to the document. The benefit of this approach is that the style sheet can be used by any HTML document. You could think of an externally defined style sheet as a style template that can be applied to any document. For example, you could apply a style template to all pages served from a particular web site by including a link to the style sheet file in each page.

The syntax for defining styles in external files is the same as for defining styles inside a document file, except that you do not need the opening and closing <STYLE> and </STYLE> tags. Here is an example:

CSS Syntax

```
/* external style sheet mystyles1.htm */

all.BOLDBLUE {color:blue; font-weight: bold;}

H1 {line-height: 18pt;}

P {color: yellow;}

/* end of file */
```

JavaScript Syntax

```
/* external style sheet mystyles1.htm */

classes.BOLDBLUE.all.color = "blue";

classes.BOLDBLUE.all.fontWeight = "bold";

tags.H1.lineHeight="18pt";

tags.P.color="yellow";

/* end of file */
```

To use an externally defined style sheet in a document, use the <LINK> tag to link to the style sheet, as in this example:

CSS Syntax

```
<HTML>
<HEAD>
  <TITLE>A Good Title</TITLE>
  <LINK REL=STYLESHEET TYPE="text/css"
   HREF="http://style.com/mystyles1.htm">
</HEAD>
```

JavaScript Syntax

```
<HTML>
<HEAD>
  <TITLE>A Good Title</TITLE>
  <LINK REL=STYLESHEET TYPE="text/javascript"
   HREF="http://style.com/mystyles1.htm">
</HEAD>
```

Defining Classes of Styles

If a document includes or links to a style sheet, all the styles defined in the style sheet can be used by the document. If the style sheet specifies the style of any HTML elements, then all the HTML elements of that kind in the document will use the specified style.

There may be times when you want to selectively apply a style to HTML elements. For example, you may want some of the paragraphs (`<P>` elements) in a document to be red, and others to be blue. In this situation, defining a style that applies to all `<P>` elements is not the right thing to do. Instead, you can define classes of style, and apply the appropriate class of style to each element that needs to use a style.

To apply a style class to an HTML element, define the style class in the style sheet, and then use the `CLASS` attribute in the HTML element.

The following examples show how to define a class called `GREENBOLD`, whose color is a medium shade of green and whose font weight is bold. The example then illustrates how to use the style in HTML text.

CSS Syntax

```
<STYLE TYPE="text/css">
 all.GREENBOLD {color:#44CC22; font-weight:bold;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
  classes.GREENBOLD.all.color = "#44CC22"
  classes.GREENBOLD.all.fontWeight = "bold"
</STYLE>
```

Style Sheet Use

```
<H1 CLASS=GREENBOLD>This Heading Is Very Green</H1>

<P CLASS = GREENBOLD>This paragraph uses the style class GREENBOLD. You
can use the CLASS attribute to specify the style class to be used by an
HTML element.</P>

<BLOCKQUOTE CLASS = GREENBOLD>

This blockquote uses the style class GREENBOLD. As a consequence, it is
both green and bold. It can be useful to use styles to make blockquotes
stand out from the rest of the page.

</BLOCKQUOTE>
```

In JavaScript syntax, you cannot use hyphens inside class names. A hyphen is actually a minus sign, which is a JavaScript operator. Class names In JavaScript syntax cannot include any JavaScript operators, including but not limited to -, +, *, /, and %.

When defining a style class, you can specify which HTML elements can use this style, or you can use the keyword `all` to let all elements use it.

For example, the following code creates a style class `DARKYELLOW`, which can be used by any `HTML` element. The code also creates a class called `RED1`, which can be used only by `<P>` and `<BLOCKQUOTE>` elements.

CSS Syntax

```
<STYLE type="text/css">
 all.DARKYELLOW {color:#EECC00;}
 P.RED1 {color: red; font-weight:bold;}
```

```
 BLOCKQUOTE.red1 {color:red; font-weight:bold;}
</STYLE>
```

JavaScript Syntax

```
<STYLE type="text/javascript">
 classes.DARKYELLOW.all.color="#EECC00";
 classes.RED1.P.color = "red";
 classes.RED1.P.fontWeight = "bold";
 classes.RED1.BLOCKQUOTE.color = "red";
 classes.RED1.BLOCKQUOTE.fontWeight = "bold";
</STYLE>
```

Style Sheet Use

```
<BODY>
<P CLASS=red1>This paragraph is red.</H1>
<P>This paragraph is in the default color, since it does not use the
class RED1.</P>
<BLOCKQUOTE CLASS="red1">This blockquote uses the class RED1.
</BLOCKQUOTE>
<H5 CLASS=red1>This H5 element tried to use the style RED1, but was not
allowed to use it.</H5>
<P CLASS=darkyellowclass>This paragraph is dark yellow.
<H5 CLASS=darkyellowclass>This H5 element tried to use the style
DARKYELLOW and was succesful.</H5>
```

An HTML element can use only one class of style. If you specify two or more classes of style for an HTML element, the first one specified is used. For example, in the following code the paragraph will use the `RED1` style and ignore the `DARKYELLOW` style:

```
<P CLASS="RED1" CLASS="DARKYELLOW">Another paragraph.</P>
```

Defining Named Individual Styles

You can create individual named styles. An HTML element can use both a class of style and a named individual style. Thus you can use individual named styles to express stylistic exceptions to a class of style. For example, if a paragraph uses the `MAIN` style class, it could also use the named style `BLUE1` which could express some differences to the `MAIN` style.

Individual names styles are also useful for defining layers of precisely positioned HTML content. For more details of layers, see the Part 2. Positioning HTML Content.

To define named styles in CSS syntax, precede the name of the style with the `#` sign. In JavaScript syntax, use the `ids` property.

To apply the style to an element, specify the style name as the value of the element's `ID` attribute.

The following codes defines a style class called `MAIN`. This style class specifies a a line height of 20 points, a font size of 18 points; a font weight of bold, and a color of red. The code also defines a named style BLUE1 whose color is blue.

CSS Syntax

```
<STYLE TYPE="text/css">
 all.STYLE1 {line-height: 20pt; font-size: 18pt;
   font-weight: bold; color: red;}
 #BLUE1 {color: blue;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
 with (classes.STYLE1.all) {
  lineHeight = "20pt";
  fontSize = "18pt";
  fontWeight = "bold";
  all.color = "red";
```

```
 }
 ids.BLUE1.color = "blue";
</STYLE>
```

Style Sheet Use

```
<P CLASS="STYLE1">Here you see some tall red text. The text in this
paragraph is much taller, bolder, and bigger than paragraph text
normally is.</P>

<P CLASS="STYLE1" ID="BLUE1">This paragraph has tall, bold, blue text.
Although this paragraph is in class STYLE1 1, whose members are normally
red, it also has a unique ID that allows it to be blue.</P>
```

Using Contextual Selection Criteria

You can define the style to be used by all HTML elements of a particular kind. If you need more control over when a style is used, you can define a style class that you can selectively apply to any element. Sometimes however, even that level of control is not enough. You might, for example, want to specify a green color for all `<EM>` elements inside level-one headings.

You can achieve this level of control over the application of styles by using contextual selection criteria in your style definition. Contextual selection criteria allow you to specify criteria such as "this style applies to this kind of element nested inside that kind of element nested inside the other kind of element."

To specify contextual selection criteria in CSS syntax, list the HTML elements in order before the opening curly brace of the properties list. In JavaScript syntax, use the predefined method `contextual()`.

The following example shows how to specify a green text color for all `<EM>` elements inside `<H1>` elements.

CSS Syntax

```
<STYLE TYPE="text/css">
  H1 EM {color:green;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
  contextual(tags.H1, tags.EM).color = "green";
</STYLE>
```

Style Sheet Use

```
<H1>This is<EM> green, emphasized text,</EM> but this is plain heading-
one text</H1>
```

Consider another example, given first in CSS syntax then in JavaScript syntax:.

```
UL UL LI {color:blue;}
```

```
contextual(tags.UL, tags.UL, tags.LI).color = "blue";
```

In this case, the selection criteria match <LI> elements with at least two <UL> parents. That is, only list items that are two levels deep in an unordered list will match this contextual selection and thus be displayed in blue.

You can use contextual selection criteria to look for tags, classes, IDs, or combinations of these. For example, the following example creates a class called MAGENTACLASS. Everything in this class is magenta colored. All paragraphs in MAGENTACLASS that are also inside <DIV> elements are italic. All text inside <B> tags nested inside paragraphs in MAGENTACLASS that are inside <DIV> elements is extra large.

CSS Syntax

```
<STYLE TYPE="text/css">
all.MAGENTACLASS {color: magenta;}
DIV P.MAGENTACLASS {font-style: italic;}
DIV P.MAGENTACLASS B {font-size:18pt;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
  classes.MAGENTACLASS.all.color = "magenta";
contextual(tags.DIV, classes.MAGENTACLASS.P).fontStyle = "italic";
contextual(tags.DIV, classes.MAGENTACLASS.P, tags.B).fontSize = "18pt";
</STYLE>
```

Style Sheet Use

```
<DIV CLASS=MAGENTACLASS>

<H3>Heading 3 in the MAGENTACLASS</H3>

<P>Is this paragraph magenta and italic? It should be. Here comes some
<B>big bold text.</B> We achieved this result with contextual
selection.</P>

<P>This paragraph should be magenta too.</P>

</DIV>

<P>This paragraph is still magenta colored, but since it is not inside a
DIV block, it should not be italic.</P>
```

Specifying Styles for Individual Elements

As well as defining styles in style sheets, you can also use the STYLE attribute of an HTML tag to define a style for use by that individual tag, and that tag only. This approach basically defines the style on the fly, and can be useful in situations where you want an element to use a style in a unique situation, where you do not need to reuse the style elsewhere in the document.

In general though, it is better to define all the style used by a document in a single place (be it at the top of the document or in a separate style sheet file) so that you know where to make changes to the style. If a style is defined in a style sheet, any element in the document can use that style. If you want to change the properties of the style, you need to make the change only once and it is automatically applied to all elements that use that style.

Sometimes, however, youmay want to specify the style of an individual element, and an easy way to do this is to use the STYLE attribute. The following example specifies the style of an individual <P> element. It also shows how to use the STYLE attribute with the <SPAN> tag to apply a style to a piece of arbitrary content.

CSS Syntax

```
<P STYLE="color:green; font-weight:bold;

margin-right:20%; margin-left:20%;

border-width:2pt; border-color:blue;">

This paragraph, and only this paragraph is bold green with big margins
```

```
and a blue border.</P>

<P>This paragraph is in the usual color, whatever that may be, but this
<SPAN STYLE="color:red; font-style:italic;">word </span> is different
from the other words in this paragraph.</P>
```

JavaScript Syntax

```
<P STYLE="color = 'green'; fontWeight='bold';

marginRight='20%' marginLeft='20%';

borderWidth='2pt'; borderColor='blue;">

This paragraph, and only this paragraph is bold green with big margins
and a blue border.</P>

<P>This paragraph is in the usual color, whatever that may be, but this
<SPAN STYLE="color='red'; fontStyle='italic'">word </span> is different
from the other words in this paragraph.</P>
```

Combining Style Sheets

You can use more than one style sheet to set the styles for a document. You might want to do this when you have several partial styles sheets that you wish to mix and match, or perhaps where your document falls into several different categories, each with its own style sheet.

For example, suppose you are are writing a white paper about the benefits of a network product from a company called Networks Unlimited. You might need to use three style sheet: one defining the company's usual style for white papers, another defining their usual style for documents about networking products, and yet another defining the corporate style for Networks Unlimited.

The following example illustrates the use of several style sheets in one document:

```
<STYLE TYPE="text/css"

SRC="http://www.networksunlimited.org/styles/corporate"></STYLE>

<STYLE TYPE="text/css"

SRC="styles/whitepaper"></STYLE>

<STYLE TYPE="text/javascript"

SRC="styles/networkthings"></STYLE>

<STYLE TYPE="text/css">
```

```
H1 {color: red;} /* override external sheets */
</STYLE>
```

For externally linked style sheets, the last one listed takes precedence over previously listed style sheets. So in this case, if `networkthings` and `white-paper` specify conflicting styles, then the styles defined in `networkthings` take precedence over styles defined in `whitepaper`.

Locally defined styles take precedence over styles defined in the `<STYLE>` element and over styles defined in external style sheets. In general, local style values override values inherited from parent elements, and more specific style values override more general values, as illustrated in the following example.

CSS Syntax

```
<STYLE TYPE="text/css">
  P {color:blue;}
  B {color:green;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
  tags.P.color="blue";
  tags.B.color="green";
</STYLE>
```

Style Sheet Use

```
<P>This is a blue paragraph, as determined by the style sheet. But these
<B>bold words are green,</B> as you see.</P>

<P STYLE="color:red">This is a red paragraph, as determined by the local
style. However, these <B>bold words are still green,</B> since the style
defined directly for bold elements overrides the style of the parent
element.</P>
```

Chapter

4

Format Properties for Block-Level Elements

This chapter discusses the formatting options for block-level elements. Block-level elements start on a new line, for example, <H1> and <P> are block-level elements, but <EM> is not.

This chapter starts off by presenting an example that illustrates the various ways of formatting block-level elements. After that comes a section discussing each kind of formatting option in detail. The chapter and ends with a brief overview of the inheritance behavior of properties that are used for formatting block-level elements.

Block-level Formatting Overview and Example

Style sheets treat each block-level element as if it were surrounded by a box. Each box can have style characteristics in the form of margins, borders, and padding. Each box can have a background image or color.

The margins indicate the inset of the edge of the box from the edges of the document (or parent element). Each box can have a border that has a flat or three dimensional appearance. The padding indicates the distance between the element's border and the element's content.

You can also set the width of a block-level element, either to a specific value or to a percentage of the width of the document (or parent element). As you can imagine, it is redundant to set the left and right margin and to also set the width.

If values are specified for the width and for both margins, the left margin setting has the highest precedence. In this case, the value for the right margin indicates the absolute maximum distance from the right edge of the containing element at which the content wraps. If the value given for the width would cause the element to run past the right margin, the width value is ignored. If the width value would cause the element to stop short of the right edge, the width value is used.

You can set the horizontal alignment of an element to left, right, or center. You do this by setting the `float` property in CSS syntax or setting the `align` property in JavaScript syntax. It is also redundant to set the left and/or right margin and then also set the element's alignment.

The following example illustrates the use of margins, paddings, border widths, background color, width, and alignment properties.

CSS Syntax

```
<STYLE TYPE="text/css">
P {
 background-color:#CCCCFF;
 /* margins */
 margin-left:20%; margin-right:20%;
 /* border widths
 border-top-width: 10pt; border-bottom-width:10pt;
 border-right-width:5pt; border-left-width:5pt;
 /* border style and color
 border-style:outset; border-color:blue;
 /* paddings */
 padding-top:10pt; padding-bottom:10pt;
```

```
 padding-right:20pt; padding-left:20pt;
}
H3 {
 /* font size and weight */
 font-size: 14pt;
 font-weight:bold;
 background-image:URL("images/grenlite.gif");
 /* center the heading with a 90% width */
 width:90%;
 float:center;
 border-color:green;
 border-style:solid;
 /* all sides of the border have the same thickness */
 border-width:10pt;
 /* all sides have the same padding */
 padding:20pt;
}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
with (tags.P) {
 backgroundColor="#CCCCFF";
 /* P border style and color */
 borderStyle="outset"; borderColor="blue";
 /* P border widths */
 borderTopWidth="10pt"; borderBottomWidth="10pt";
 borderLeftWidth="5pt"; borderRightWidth="5pt";
 /* P paddings */
 paddingTop="10pt"; paddingBottom="10pt";
 paddingLeft="20pt"; paddingRight="20pt";
 /* P margins */
```

```
 marginLeft= "20%"; marginRight="20%";
}
with (tags.H3) {
 backgroundImage ="images/grenlite.gif";
 /* font size and weight */
 fontSize="14pt"; fontWeight="bold";
 /* H3 border style and color */
 borderStyle="solid"; borderColor="green";
/* center the heading with a 90% width */
 width="90%"; align="center";
 /* all sides of the border have the same thickness */
 borderWidths("10pt");
 /* all sides have the same padding */
 paddings("20pt");
}
</STYLE>
```

Style Sheet Use

```
<H3>H3 with a Solid Border</H3>
<P>Borders have their uses in everyday life. For example, borders round
a paragraph make the paragraph stand out more than it otherwise would.
</P>
<P>This is another paragraph with a border. You have to be careful not
to make the borders too wide, or else they start to take over the page.
</P>
```

Setting Margins or Width

The margins indicate the inset of the element from the edges of the document (or parent element.) You can set right, left, top, and bottom margins. The "edge" of the parent is the theoretical rectangle that would be drawn round the inside of the padding, border, or margins of the parent element, if it has any of these properties.

You can set the values of the margins for a block-level element by specifying the following properties (shown as CSS syntax/JavaScript syntax property names):

- `margin-top/marginTop`
- `margin-bottom/marginBottom`
- `margin-left/marginLeft`
- `margin-right/marginRight`
- You can set all four properties at once to the same value, either by setting the `margin` property in CSS syntax or by using the `margins()` function in JavaScript syntax.

You can set the horizontal alignment of an element to left, right, or center. You do this by setting the `float` property in CSS syntax or setting the `align` property in JavaScript syntax. It is redundant to set the left and/or right margin and then also set the element's alignment.

Instead of setting specific margin values, you can also set the `width` property. You can set this to either a specific value (for example, `200pt`) or to a percentage of the width of the containing element or document (for example, `60%`). If desired, you can set the width and either the left or right margin, so long as the total does not exceed 100% of the width of the parent. It is not useful, however, to set the width and also to set both margins, since two of the values imply the third. (For example, if the left margin is 25% and the width is 60%, then the right margin must be 15%.)

Two or more adjoining margins (that is, with no border, padding or content between them) are collapsed to use the maximum of the margin values. In the case of negative margins, the absolute maximum of the negative adjoining margins is deducted from the maximum of the positive adjoining margins.

To set the default margins for everything in a document, you can specify the margin properties for the <BODY> tag. For example, the following code sets the left and right margins to 20 points.

CSS Syntax

```
<STYLE TYPE="text/css">
BODY {margin-left:20pt; margin-right:20pt;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
with (tags.BODY) {
 marginLeft="20pt"; marginRight="20pt";
}
</STYLE>
```

See Block-level Formatting Overview and Example for an example of setting margins and width.

Setting Border Widths, Color, and Style

You can set the width of the border surrounding a block-level element by specifying the following properties (shown as CSS syntax/JavaScript syntax values):

- `border-top-width/borderTopWidth`
- `border-bottom-width/borderBottomWidth`
- `border-left-width/borderLeftWidth`
- `border-right-width/borderRightWidth`
- You can set all four properties at once to the same value, either by setting the `border-width` property in CSS syntax or by using the `borderWidths()` function in JavaScript syntax.

You can set the style of the border by specifying the `border-style` (CSS syntax) or `borderStyle` (JavaScript syntax) property. You can give the border a flat appearance by setting the border-style to `solid` or `double`, or you can give it a 3D appearance, by setting the border-style to `groove`, `ridge`, `inset`, or `outset`.

You can set the color of the border by specifying the `border-color` (CSS syntax) or `borderColor` (JavaScript syntax) property.

For an example of each of the border styles, see:

borders.htm *StyleSheetExample*

For another example of setting border widths, border style, and border color, see Block-level Formatting Overview and Example.

Setting Paddings

The padding indicates the distance between the border of the element and its content. The padding is displayed even if the element's border is not displayed.

You can set the size of the padding surrounding a block-level element by specifying the following properties (shown as CSS syntax/JavaScript syntax values):

- `padding-top/paddingTop`
- `padding-bottom/paddingBottom`
- `padding-left/paddingLeft`
- `padding-right/paddingRight`
- You can set all four properties at once to the same value, either by setting the `padding` property in CSS syntax or by using the `paddings()` function in JavaScript syntax.

See Block-level Formatting Overview and Example for an example of setting paddings.

Inheritance of Block-Level Formatting Properties

The width, margins, border characteristics, and padding values of a parent element are not inherited by its child elements. However, at first glance it might seem that these values are inherited, since the values of the parent elements affect the child elements.

For example, suppose you set the left margin of a `DIV` element to 10 points. You can think of this `DIV` element as a big box that gets inset by 10 points on the left. Assume that the `DIV` element has no border width and no padding. If all the elements inside the `DIV` have a margin of 0, they are smack up against the edge of that box. Since the box is inset by 10 points, all the elements end up being inset by 10 points.

Now consider what would happen if the child elements did inherit the margin value from their parent element. If that were the case, then the DIV block would have a left margin of 10 points, and child elements would also each have a left margin of 10 points, so they would be indented on the left by 20 points.

Chapter

5

Style Sheet Reference

This section includes reference information for both CSS syntax and JavaScript syntax. It covers style sheet functionality that is implemented in Netscape Navigator 4.0.

This reference does not include style sheet properties that can be used to specify positions for blocks of HTML content. These properties are discussed in Part 2. Positioning HTML Content.

This chapter is organized in the following sections:

Comments in Style Sheets

New HTML Tags

- <STYLE>
- <LINK>
- <SPAN>

New Attributes for Existing HTML Tags

- STYLE
- CLASS

- ID

New JavaScript Object Properties

- tags
- classes
- ids

Style Sheet Properties

Font Properties

- Font Size
- Font Style
- Font Family
- Font Weight

Text Properties

- Line Height
- Text Decoration
- Text Transform
- Text Alignment
- Text Indent

Block-Level Formatting Properties

- Margins
- Padding
- Border Widths
- Border Style

Comments in Style Sheets

Comments in style sheets are similar to those in the C programming language. For example:

```
B {color:blue;} /* bold text will be blue */
tags.B.color = "blue"; /* bold text will be blue */
```

JavaScript style sheet syntax also supports comments in the C++ style, for example:

```
tags.B.color = "blue"; // bold text will be blue
```

Comments cannot be nested.

New HTML Tags

This section lists the HTML tags that are useful for working with styles.

<STYLE>

The <STYLE> and </STYLE> tags indicate a style sheet. Inside <STYLE> and </STYLE> you can specify styles for elements, define classes and IDs, and generally establish styles for use within the document.

To specify that the style sheet uses JavaScript syntax, set the TYPE attribute to "text/javascript". To specify that the style sheet uses CSS syntax, set the TYPE attribute to "text/css". The default value for TYPE is "text/CSS".

For example:

```
<STYLE TYPE="text/css">
 BODY {margin-right: 20%; margin-left:20%;}
 PRE {color:green;}
 all.CLASS1 {float:right; font-weight: bold;}
</STYLE>
```

<LINK>

Use the <LINK> element to link to an external style sheet for use in a document. For example:

CSS Syntax

```
<HTML>
<HEAD>
<TITLE>A Good Title</TITLE>
<LINK REL=STYLESHEET TYPE="text/css"
 HREF="http://style.com/mystyles1.htm">
</HEAD>
```

JavaScript Syntax

```
<HTML>

<HEAD>

<TITLE>A Good Title</TITLE>

<LINK REL=STYLESHEET TYPE="text/javascript"

 HREF="http://style.com/mystyles1.htm">

</HEAD>
```

<SPAN>

Use the inline <SPAN> and </SPAN> elements to indicate the beginning and end of a piece of text to which a style is to be applied.

The following example applies an individual style to a piece of text.

```
<P>Here is some normal paragraph text. It looks OK, but would be much
better if it was<SPAN style="color:blue; font-weight:bold; font-
style:italic"> in bright, bold, italic blue. </SPAN>The blue text stands
out much more.</P>
```

You can use the <SPAN> element to achieve effects such as a large initial letter, for example:

```
<STYLE TYPE="text/css">

 init-letter.all {font-size:400%; font-weight:bold;}

</STYLE>

<P><SPAN class="init-letter">T</SPAN>his is...</P>
```

New Attributes for Existing HTML Tags

This section lists the new attributes for existing HTML tags that are useful for working with styles. These attributes can be used with any HTML tag to specify the style for that tag.

STYLE

The STYLE attribute determines the style of a specific element. For example:

CSS Syntax

```
<H3 STYLE="line-height:24pt; font-weight:bold; color:cyan;">
Cyan Heading</H3>
```

JavaScript Syntax

```
<H3 STYLE="lineHeight='24pt'; fontWeight='bold'; color='cyan'">
Cyan Heading</H3>
```

CLASS

The CLASSES JavaScript property allows you to define classes of styles in a style sheet. The CLASS attribute specifies a style class to apply to an element.

Although CSS syntax and JavaScript syntax use slightly different syntax to define classes of styles, the use of the CLASS attribute is the same in both syntaxes. For example:

CSS SyntaxExample

```
<STYLE TYPE="text/css">
 H3.class1 {font-style:italic; color:red;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/javascript">
classes.class1.H3.fontStyle="italic";
classes.class1.H3.color="red";
</STYLE>
```

Style Sheet Use

```
<H3 CLASS="class1">This H3 is in red italic letters.</H3>
```

Class names are case-sensitive.

Each HTML element can use only one style class.

To specify that a class can apply to all elements, use the element selector `all` when you set the properties for the class. For example, the code sample below specifies that the class `LEMON` can be applied to any element, and all elements that use the style class `LEMON` are yellow.

CSS Syntax

```
<STYLE TYPE="text/css">
 all.LEMON {color:yellow;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
 classes.LEMON.all.color="yellow";
</STYLE>
```

Style Sheet Use

```
<H1 class="LEMON">A Nice Yellow Heading</P>
<P CLASS="LEMON">What a nice shade of yellow this paragraph is.</P>
```

For more information about creating classes of style and for more examples, see the section Defining Classes of Styles in Chapter 3, “Creating Style Sheets and Assigning Styles.”

ID

When defining style sheets, you can create individual named styles.

An element can use a style class and also use a named style. This allows you to use named styles to express individual stylistic exceptions to a style class.

To define an individual names style in CSS syntax, you use the # sign to indicate a name for an individual style, while In JavaScript syntax, you use the ID selector.

In both CSS syntax and JavaScript syntax, you use the ID attribute in an HTML element to specify the style for that element.

ID names are case-sensitive.

ID styles are particularly useful for working with layers of precisely positioned HTML content, as discussed in Part 2. Positioning HTML Content.

The following code shows an example of the use of individual named styles. In this example, the STYLE1 class defines a style with several characteristics. The named style A1 specifies that the color is blue. This style can be used to specify that a paragraph has all the style characteristics of STYLE1, except that its color is blue instead of red.

CSS Syntax

```
<STYLE TYPE="text/css">
 P.STYLE1 {
 color:red; font-size:24pt; line-height:26pt;
 font-style:italic; font-weight:bold;
 }
 #A1 {color: blue;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
 with (classes.STYLE1.P) {
  color="red";
  fontSize="24pt";
  lineHeight="26pt";
  fontStyle="italic";
  fontWeight="bold";
 }
 ids.A1.color= "blue";
</STYLE>
```

Style Sheet Use

```
<P CLASS="STYLE1">Big red text</P>
<P CLASS="STYLE1" ID="A1">Big blue text</P>
```

New JavaScript Object Properties

This section discusses the new JavaScript object properties that are useful for defining style sheets using JavaScript syntax.

tags

When using JavaScript syntax within the <STYLE> element, you can set styles by using the tags property of the JavaScript object document.

The following example uses JavaScript syntax to specify that all paragraphs appear in red:

```
<STYLE TYPE="text/javascript">
 tags.P.color = red;
</STYLE>
```

In CSS syntax, this would be:

```
<STYLE TYPE="text/css">
 P {color:red;}
</STYLE>
```

The tags property always applies to the document object for the current document, so you can omit document from the expression document.tags. For example, the following two statements both say the same thing:

```
document.tags.P.color = "red";
tags.P.color = "red";
```

To set default styles for all elements in a document, you can set the desired style on the <BODY> element, since all other elements inherit from <BODY>. For example, to set a universal right margin for the document:

```
tags.body.marginRight="20pt"; /*JavaScript syntax */
BODY {margin-right:20pt;} /* CSS syntax */
```

classes

See the CLASS section for a discussion of the classes JavaScript property.

ids

See the ID section for a discussion of the `ids` JavaScript property.

Style Sheet Properties

Font Properties

Using styles, you can specify font size, font family, font style, and font weight for any element.

Font Size

CSS syntax name: `font-size`

JavaScript syntax name: `fontSize`

Possible values:	*absolute-size*, *relative-size*, *length*, *percentage*
Initial value:	`medium`
Applies to:	all elements
Inherited:	yes
Percentage values:	relative to parent element's font size

absolute-size

An *absolute-size* is a keyword such as:

`xx-small`
`x-small`
`small`
`medium`
`large`
`x-large`

`xx-large`

relative-size

A *relative-size* keyword is interpreted relative to the font size of the parent element. Note that relative values only equate to actual values when the element whose font size is a relative value has a parent element that has a font size. (A relative size has to have something to be relative to.)

Possible values are:

`larger`

`smaller`

For example, if the parent element has a font size of `medium`, a value of `larger` will make the font size of the current element be `large`.

length

A length is a number followed by a unit of measurement, such as `24pt`.

percentage

A *percentage* keyword sets the font size to a percentage of the parent element's font size.

CSS Syntax

```
P {font-size:12pt;}
EM {font-size:120%};
BLOCKQUOTE {font-size:medium;}
B {font-size:larger;}
```

JavaScript Syntax

```
tags.P.fontSize = "12pt";
tags.EM.fontSize = 120%;
tags.BLOCKQUOTE.fontSize = "medium";
tags.B.fontSize="larger";
```

Font Family

CSS syntax name: `font-family`

JavaScript syntax name: `fontFamily`

Possible values:	*fontFamily*
Initial value:	the default font, which comes from user preferences.
Applies to:	all elements
Inherited:	yes
Percentage values:	NA

fontFamily

The *fontFamily* indicates the font family to use, such as Helvetica or Arial. If a list of font names is given, the browser tries each named font in turn until it finds one that exists on the user's system. If none of the specified font families are available on the user's system, the default font is used instead.

If you link a font definition file to your web page, the font definition file will be downloaded with the page, thus guaranteeing that all the fonts in the definition file are available on the user's system while the user is viewing that page. For more information about linking fonts to a document, see Part 3. Downloadable Fonts.

There is a set of generic family names that are guaranteed to indicate a font on every system, but that exact font is system-dependent. The five generic font families are:

- serif
- sans-serif
- cursive
- monospace
- fantasy

CSS Syntax Example

```
<STYLE TYPE="text/css">
 H1 {fontFamily:Helvetica, Arial, sans-serif;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/javascript">
 tags.H1.fontFamily="Helvetica, Arial, sans-serif";
</STYLE>
```

Font Weight

CSS syntax name: `font-weight`

JavaScript syntax name: `fontWeight`

Possible values:	`normal, bold, bolder, lighter, 100 - 900`
Initial value:	normal
Applies to:	all elements
Inherited:	yes
Percentage values:	N/A

The font weight indicates the weight of the font. For example:

CSS Syntax Example

```
<STYLE>
BLOCKQUOTE {font-weight: bold;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE>
tags.BLOCKQUOTE.fontWeight="bold";
</STYLE>
```

The possible values are `normal`, `bold`, `bolder`, and `lighter`. You can also specify weight as a numerical value from 100 to 900, where 100 is the lightest and 900 is the heaviest.

Font Style

CSS syntax name: `font-style`

JavaScript syntax name: `fontStyle`

Possible values:	`normal, italic`
Initial value:	normal
Applies to:	all elements
Inherited:	yes
Percentage values:	N/A

This property determines the style of the font.

The following example specifies that emphasized text within `<H1>` elements appears in italic.

CSS Syntax Example

```
<STYLE>
H1 EM {font-style: italic;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE>
contextual(tags.H1, tags.EM).fontStyle = "italic";
</STYLE>
```

Text Properties

The use of style sheets allows you to set text properties such as line height and text decoration.

Line Height

CSS syntax name: `line-height`

JavaScript syntax name: `lineHeight`

Possible values	*number, length, percentage,* `normal`
Initial value:	normal for the font
Applies to:	block-level elements
Inherited:	yes
Percentage values:	refers to the font size of the element itself

This property sets the distance between the baselines of two adjacent lines. It applies only to block-level elements.

number:

If you specify a numerical value without a unit of measurement, the line height is the font size of the current element multiplied by the numerical value. This differs from a percentage value in the way it inherits: when a numerical value is specified, child elements inherit the factor itself, not the resultant value (as is the case with percentage and other units).

For example:

```
fontSize:10pt;
line-height:1.2; /* line height is now 120%, ie 12pt */
font-size:20pt; /* line height is now 24 pt, */
```

length:

An expression of line height as a measurement, for example:

```
line-height:0.4in;
line-height:18pt;
```

percentage

Percentage of the element's font size, for example:

```
line-height:150%;
```

Negative values are not allowed.

Text Decoration

CSS syntax name: `text-decoration`

JavaScript syntax name: `textDecoration`

Possible values:	`none, underline, line-through, blink`
Initial value:	`none`
Applies to:	all elements
Inherited:	no, but see clarification below
Percentage values:	N/A

This property describes decorations that are added to the text of an element. If the element has no text (for example, the `<IMG>` element in HTML) or is an empty element (for example, "`<EM></EM>`"), this property has no effect.

This property is not inherited, but children elements will match their parent. For example, if an element is underlined, the line should span the child elements. The color of the underlining will remain the same even if child elements have different `color` values.

For example:

```
BLOCKQUOTE {text-decoration: underline;}
```

The text decoration options do not include color options, since the color of text is derived from the `color` property value.

.

Text Transform

CSS syntax name: `text-transform`

JavaScript syntax name: `textTransform`

Possible values:,	`capitalize, uppercase, lowercase, none`
Initial value:	`none`
Applies to:	all elements
Inherited:	yes
Percentage values:	N/A

This property indicates text case.

`capitalize`

Display the first character of each word in uppercase.

`uppercase`

Display all letters of the element in uppercase.

`lowercase`

Display all letters of the element in lowercase.

`none`

Neutralizes inherited value.

For example:

CSS Syntax Example

```
<STYLE TYPE="text/css">
H1 {text-transform:capitalize;}
H1.CAPH1 {text-transform: uppercase;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE>
tags.H1.textTransform = "capitalize";
classes.CAPH1.H1.textTransform = "uppercase";
</STYLE>
```

Style Sheet Use

```
<H1>This is a regular level-one heading</H1>
<H1 CLASS=CAPH1>important heading</H1>
```

Text Alignment

CSS syntax name: `text-align`

JavaScript syntax name: `textAlign`

Possible values:	`left, right, center, justify`
Initial value:	`left`
Applies to:	block-level elements
Inherited:	yes
Percentage values:	N/A

This property describes how text is aligned within the element.

Example:

```
tags.P.textAlign = "center"
```

CSS Syntax Example

```
<STYLE TYPE="text/css">
all.RIGHTHEAD {text-align:right; color:blue;}
P.LEFTP {text-align:left; color:red;}
</STYLE>
```

JavaScript Syntax

```
<STYLE TYPE="text/javascript">
classes.RIGHTHEAD.all.textAlign="right";
classes.LEFTP.P.textAlign="left";
classes.RIGHTHEAD.all.color="blue";
classes.JUSTP.P.color="red";
```

```
</STYLE>
```

Style Sheet Use

```
<H3>A Normal Heading</H3>

<H3 CLASS=RIGHTHEAD>A Right-Aligned Heading</H3>

<P>This is a normal paragraph. This is what paragraphs usually look
like, when they are left to their own devices, and you do not use style
sheets to control their text alignment.</P>

<P CLASS = LEFTP>This paragraph is left-justified, which means it has a
ragged right edge. Whenever paragraphs contain excessively, perhaps
unnecessarily, long words, the raggedness of the justification becomes
more manifestly apparent than in the case where all the words in the
sentence are short.</P>
```

Text Indent

CSS syntax name: `text-indent`

JavaScript syntax name: `textIndent`

Possible values:	*length, percentage*
Initial value:	0
Applies to:	block-level elements
Inherited:	yes
Percentage values:	refer to parent element's width

The property specifies indentation that appears before the first formatted line. The `text-indent` value may be negative. An indent is not inserted in the middle of an element that was broken by another element (such as `<BR>` in HTML).

length

Length of the indent as a numerical value with units, for example:

```
P {text-indent:3em;}
```

percentage

Length of the indent as a percentage of the parent element's width, for example:

```
P {text-indent:25%;}
```

CSS Syntax Example

```
<STYLE TYPE="text/css">

P.INDENTED {text-indent:25%;}

</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/css">

classes.INDENTED.P.textIndent="25%";

</STYLE>
```

Style Sheet Use

```
<P CLASS=INDENTED>

The first line is indented 25 percent of the width of the parent
element, which in this case happens to be the BODY tag, since this
element is not embedded in anything else.</P>

<BLOCKQUOTE>

<P CLASS=INDENTED>

This time the first line is indented 25 percent from the blockquote that
surrounds this element. A blockquote automatically indents its contents.

</P>

</BLOCKQUOTE>
```

Block-Level Formatting Properties

Style sheets treat each block-level element as if it is surrounded by a box. Block-level elements start on a new line, for example, <H1> and <P> are block-level elements, but <EM> is not.

Each box can have padding, border, and margins.You can set values for top, bottom, left and right paddings, border widths, and margins.

For a more detailed overview discussion of block-level formatting, see Chapter 4, "Format Properties for Block-Level Elements."

Margins

CSS syntax names: `margin-left`, `margin-right`, `margin-top`, `margin-bottom`, `margin`

JavaScript syntax names: `marginLeft`, `marginRight`, `marginTop`, `marginBottom` and `margins`()

Possible values	*length, percentage,* `auto`
Initial value:	0
Applies to:	all elements
Inherited:	no
Percentage values:	refer to parent element's width

These properties set the margin of an element. The margins express the minimal distance between the borders of two adjacent elements.

You can set each margin individually by specifying values for `margin-left`/`marginLeft`, `margin-right`/`marginRight`, `margin-top`/`marginTop` and `margin-bottom`/`marginBottom.`

In CSS syntax you can set all margins to the same value at one time by setting the `margin` property (note that the property name is singular). In JavaScript syntax you can use the `margins()` method sets the margins for all four sides at once. (Note that the function name is plural.)

The arguments to the `margin` property and `margins()` method are top, right, bottom and left margins respectively. For example:

CSS Syntax

```
/* top=10pt, right=20pt, bottom=30pt, left=40pt */

P {margin:10pt 20pt 30pt 40pt;}

/* set all P margins to 40 pt */
```

```
P {margin:40pt;}
```

JavaScript Syntax

```
/* top=10pt, right=20pt, bottom=30pt, left=40pt */
tags.BODY.margins("10pt", "20pt", "30pt", "40pt");
/* set all P margins to 40 pt */
tags.P.margins("40pt");
```

Adjoining margins of adjacent elements are added together, unless one of the elements has no content, in which case its margins are ignored. For example, if an `<H1>` element with a bottom margin of 40 points, is followed by a `<P>` element with a top margin of 30 points, then the separation between the two elements is 70 points. However, if the `<H1>` element has content, but the `<P>` element is empty, then the margin between them is 40 points.

When margin properties are applied to replaced elements (such as an `<IMG>` tag), they express the minimal distance from the replaced element to any of the content of the parent element.

The use of negative margins is not recommended because it may have unpredictable results.

For a working example of setting margins, see the section Block-level Formatting Overview and Example.

Padding

CSS syntax names: `padding-top`, `padding-right`, `padding-bottom`, `padding-left`, `paddings`

JavaScript syntax names: `paddingTop`, `paddingRight`, `paddingBottom`, `paddingLeft`, and `paddings`()

Possible values:	*length, percentage*
Initial value:	0
Applies to:	all elements
Inherited:	no
Percentage values:	refer to parent element's width

These properties describe how much space to insert between the border of an element and the content (such as text or image). You can set the padding on each side individually by specifying values for `padding-top`/`paddingTop`, `padding-right`/`paddingRight`, `padding-left`/`paddingLeft` and `padding-bottom`/`paddingBottom`.

In CSS syntax you can use the `padding` property (note that it is padding singular) to set the padding for all four sides at once. In JavaScript syntax you can use the `paddings()` method to set the margins for all four sides at once.

The arguments to the `padding` property (CSS syntax) and the `paddings()` method (JavaScript syntax) are the top, right, bottom and left padding values respectively.

CSS Syntax

```
/* top=10pt, right=20pt, bottom=30pt, left=40pt */
P {padding:10pt 20pt 30pt 40pt;}
/* set the padding on all sides of P to 40 pt */
P {padding:40pt;}
```

JavaScript Syntax

```
/* top=10pt, right=20pt, bottom=30pt, left=40pt */
tags.P.paddings("10pt", "20pt", "30pt", "40pt")
/* set the padding on all sides of P to 40 pt */
tags.P.paddings("40pt");
```

Padding values cannot be negative.

To specify the color or image that appears in the padding area, you can set the background color or background image of the element. For information about setting background color, see the section Background Color. For information about setting a background image, see the section Background Image.

For a working example of setting paddings, see the section Block-level Formatting Overview and Example.

Border Widths

CSS syntax names: `border-top-width`, `border-bottom-width`, `border-left-width`, `border-right-width`, `border-width`

JavaScript syntax names: `borderTopWidth`, `borderBottomWidth`, `borderLeftWidth`, `borderRightWidth`, and `borderWidths`()

Possible values:	*length*
Initial value:	none
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

These properties set the width of a border around an element.

You can set the width of the top border by specifying a value for `border-top-width`/`borderTopWidth.` You can set the width of the right border by specifying a value for `border-right-width`/`borderRightWidth.` You can set the width of the bottom border by specifying a value for `border-bottom-width`/`borderBottomWidth`. You can set the width of the bottom border by specifying a value for `border-left-width`/ `borderLeft-Width.`

In CSS syntax, you can set all four borders at once by setting the `border-width` property. In JavaScript syntax you can set all four borders at once by using the `borderWidths()` function.

The arguments to the `border-width` property (CSS syntax) and the `border-Widths()` function (JavaScript syntax) are the top, right, bottom and left border widths respectively.

```
/* top=1pt, right=2pt, bottom=3pt, left=4pt */

P {border-width:1pt 2pt 3pt 4pt;} /* CSS */

tags.P.borderWidths("1pt", "2pt", "3pt", "4pt"); /* JavaScript syntax */

/* set the border width to 2 pt on all sides */

P {border-width:40pt;} /* CSS */

tags.P.borderWidths("40pt"); /* JavaScript syntax */
```

For a working example of setting border widths, see the section Block-level Formatting Overview and Example.

Border Style

CSS syntax name: `border-style`

JavaScript syntax name: `borderStyle`

Possible values:,	`none, solid, double, inset, outset, groove, ridge`
Initial value:	none
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

This property sets the style of a border around a block-level element.

For the border to be visible however, you must also specify the border width. For details of setting the border width see the section Setting Border Widths, Color, and Style or the section Border Widths.

For an example of each of the border values, see:

borders.htm *StyleSheetExample*

Border Color

CSS name: `border-color`

JavaScript syntax name:`borderColor`

Possible values:	`none,` *colorvalue*
Initial value:	none
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

This property sets the color of the border. The color can either be a named color or a 6-digit hexadecimal value indicating a color or an rgb color value.

For a list of the named colors, see the section Color Units.

For example:

CSS Syntax

```
P {border-color:blue;}
BLOCKQUOTE {border-color:#0000FF;}
H1 {border-color:rgb(0%, 0%, 100%);}
```

JavaScript Syntax

```
tags.P.borderColor="blue";
tags.BLOCKQUOTE.borderColor="#0000FF";
tags.H1.borderColor="rgb(0%, 0%, 100%);
```

For a working example of setting border color, see the section Block-level Formatting Overview and Example.

Width

CSS syntax name: width

JavaScript syntax name: width

Possible values:	*length*, *percentage*, `auto`
Initial value:	`auto`
Applies to:	block-level and replaced elements
Inherited:	no
Percentage values:	refer to parent element's width

This property determines the width of an element.

Note that if you set the left and right margins, and also the width of a property, the margin settings take precedence over the width setting. For example, if the left margin setting is 25%, the right margin setting is 10%, and the width setting is 100%, the width setting is ignored. (The width will end up being 65% total.)

CSS Syntax Example

```
all.NARROW {width:50%;}

all.INDENTEDNARROW {margin-left:20%; width:60%;}
```

JavaScript Syntax Example

```
classes.NARROW.all.width = "50%";

classes.INDENTEDNARROW.all.width = "60%";

classes.INDENTEDNARROW.all.marginLeft = "20%";
```

For a working example of setting the width of an element, see the section Block-level Formatting Overview and Example.

Alignment

CSS syntax name: float

JavaScript syntax name: align

Possible values:	`left, right, none`
Initial values:	`none`
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

The `float` property (CSS syntax) and `align` property (JavaScript syntax) determine the alignment of an element within its parent. (Note that the `text-align`/`textAlign` property determines the alignment of the content of text elements.)

The term `float` is a reserved word in JavaScript, which is why the JavaScript syntax uses the name `align` instead of `float` for this property.

Using the `float`/`align` property, you can make an element float to the left or the right and indicate how other content wraps around it.

If no value is specified, the default value is `none`. If the value is `none`, the element is displayed where it appears in the text.

If the value is `left` or `right`, the element is displayed on the left or the right (after taking margin properties into account). Other content appears on the right or left side of the floating element. If the value is `left` or `right`, the element is treated as a block-level element.

Using the `float`/`align` property, you can declare elements to be outside the normal flow of elements. For example, if the `float`/`align` property of an element is `left`, the normal flow wraps around on the right side.

If you set an element's `float`/`align` property set, do not also specify margins for it. If you do, the wrapping effect will not work properly. However, if you want a floating element to have a left or right margin, you can put it inside another element, such as a `<DIV>` block, that has the desired margins.

CSS Syntax Example

```
<STYLE TYPE="text/css">
H4 {
 width:70%;
 border-style:outset;
 border-width:2pt;
 border-color:green;
 background-color:rgb(70%, 90%, 80%);
 padding:5%;
 font-weight:bold;
}
H4.TEXTRIGHT {text-align:right; margin-right:30%;}
H4.TEXTRIGHT_FLOATLEFT {text-align:right; float:left;}
H4.FLOATRIGHT {float:right;}
H4.FIXED_RIGHT_MARGIN {float:right; margin-right:30%;}
```

```
</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/javascript">
with (tags.H4) {
 width="70%";
 borderStyle="outset";
 borderWidth="2pt";
 borderColor="green";
 backgroundColor = "rgb(70%, 90%, 80%)";
 paddings("5%");
 fontWeight="bold";
}
classes.TEXTRIGHT.H4.textAlign="right";
classes.TEXTRIGHT.H4.marginRight="30%;"
classes.TEXTRIGHT_FLOATLEFT.H4.textAlign="right";
classes.TEXTRIGHT_FLOATLEFT.H4.align="left";}
classes.FLOATRIGHT.H4.align="right";
classes.FIXED_RIGHT_MARGIN.H4.align="right";
classes.FIXED_RIGHT_MARGIN.H4.marginRight="30%";
</STYLE>
```

Style Sheet Use

```
<BODY>
<H4>Level-Four Heading</H4>
<P>I am a plain paragraph, positioned below a non-floating level-four
heading.
</P>
<H4 CLASS=TEXTRIGHT>H4 - My Text On Right, No Float</H4>
<P>I am also a plain paragraph, positioned below a non-floating level-
four heading. It just happens that the heading above me has its text
alignment set to right.
</P>
```

```
<H4 CLASS = FLOATRIGHT>H4 - Float = Right</H4>

<P>I am a regular paragraph. There's not much more you can say about me.
I am positioned after a level-four heading that is floating to the
right, so I come out positioned to the left of it.</P>

<BR CLEAR>

<H4 CLASS=TEXTRIGHT_FLOATLEFT>H4 - My Text on Right, Float = Left </H4>

<P>I'm also just a plain old paragraph going wherever the flow takes me.

</P>

<BR CLEAR>

<H4 CLASS=FIXED_RIGHT_MARGIN>H4 - Float = Right, Fixed Right Margin</H4>
```

<P>Hello? Hello!! I am wrapping round an H4 that is floating to the right and has a fixed right margin. When I try to satisfy all these requirements, you see what happens! For best results, do not set the left and/or right margin when you set the float (CSS syntax) or align (JavaScript syntax) property. Use an enclosing element with margins instead.

```
</P>

<BR CLEAR>

<DIV STYLE="margin-left:30%;">

<H4 CLASS = FLOATRIGHT>H4 - Float = Right</H4>

<P>Notice how the heading next to me seems to have a right margin.
That's because we are both inside a DIV block that has a right margin.</
P>

<BR CLEAR>

</DIV>

</BODY>
```

Clear

CSS syntax name: `clear`

JavaScript syntax name: `clear`

Possible values:	`none, left, right, both`
Initial value:	`none`
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

This property specifies whether an element allows floating elements on its sides. More specifically, the value of this property lists the sides where floating elements are not accepted. With `clear` set to `left`, an element will be moved below any floating element on the left side. With `clear` set to `none`, floating elements are allowed on all sides.

Example:

```
P {clear:left;}
```

```
tags.H1.clear = "left";
```

Color and Background Properties

Just as you can set color and background properties for a document as a whole, you can set them for block-level elements too. These properties are applied to the "box" that contains the element.

Color

CSS syntax name: `color`

JavaScript syntax name: `color`

Possible values:	*color*
Initial value:	black
Applies to:	all elements
Inherited:	yes
Percentage values:	N/A

This property describes the text color of an element, that is, the "foreground" color.

See the section Color Units for information about how to specify colors.

The following examples illustrate the ways to set the color to red.

CSS Syntax Example

```
<STYLE TYPE="text/css">
EM {color:red;}
B {color:rgb(255, 0, 0);}
I {color:rgb(100%, 0%, 0%);}
CODE {color:#FF0000;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/javascript">
tags.EM.color="red";
tags.B.color="rgb(255, 0, 0)";
tags.I.color="rgb(100%, 0%, 0%)";
tags.CODE.color="#FF0000";
</STYLE>
```

Background Image

CSS syntax name: `background-image`

JavaScript syntax name: `backgroundImage`

Possible values:	*url*
Initial value:	empty
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

This property specifies the background image of an element.

Partial URLs are interpreted relative to the source of the style sheet, not relative to the document.

CSS Syntax Example

```
<STYLE TYPE="text/css">
 H1.SPECIAL {
 background-image:url(images/glass2.gif);
 padding:20pt;
 color:yellow;
}
H2.SPECIAL {
 padding:20pt;
 background-color:#FFFF33;
 border-style:solid;
 border-width:1pt;
 border-color:black;
}
P.SPECIAL B {background-image:url(images/tile1a.gif); }
P.SPECIAL I {background-color:cyan;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/javascript">
classes.SPECIAL.H1.backgroundImage = "images/glass2.gif";
classes.SPECIAL.H1.paddings("20pt");
classes.SPECIAL.H1.color="yellow";
classes.SPECIAL.H2.paddings("20pt");
classes.SPECIAL.H2.backgroundColor="FFFF33";
classes.SPECIAL.H2.borderStyle="solid";
classes.SPECIAL.H2.borderWidth="1pt";
classes.SPECIAL.H2.borderColor="black";
contextual(classes.SPECIAL.P, tags.B).backgroundImage=
  "images/tile1a.gif";
contextual(classes.SPECIAL.P, tags.I).backgroundColor="cyan";
</STYLE>
```

Style Sheet Use

```
<H1 CLASS=SPECIAL>Heading One with Image Background</H1>
<P CLASS=SPECIAL>
Hello. Notice how the portion of this paragraph that has an <B>image
background</B> is promoted to being a block-level element on its own
line.</P>
<H2 CLASS=SPECIAL>Heading Two with Solid Color Background</H2>
<P CLASS=SPECIAL>Hello, here is some <I>very interesting</I>
information. Notice that each <I>colored portion</I> of this paragraph
just continues right along in its normal place.
</P>
```

Background Color

CSS syntax name: `background-color`

JavaScript syntax name: `backgroundColor`

Possible Values:	*color*
Initial value:	empty
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

This property specifies a solid background color for an element.

See the previous section, Background Image, for a working example.

Classification Properties

These properties classify elements into categories more than they set specific visual parameters.

Display

CSS syntax name: `display`

JavaScript syntax name: `display`

Possible values:,	`block, inline, list-item none`
Initial value:	according to HTML
Applies to:	all elements
Inherited:	no
Percentage values:	N/A

This property indicates whether an element is inline (for example, `<EM>` in HTML), block-level element (for example. `<H1>` in HTML), or a block-level list item (for example, `<LI>` in HTML). For HTML documents, the initial value is taken from the HTML specification.

A value of `none` turns off the display of the element, including children elements and the surrounding box. (Thus if the value is set to `none`, the element is not be displayed.)

Note that block-level elements do not seem to respond to having their display property set to `inline`.

CSS Syntax Example

```
EM.LISTEM {display:list-item;}
```

JavaScript Syntax Example

```
classes.LISTEM.EM.display="list-item";
```

List Style Type

CSS syntax name: `list-style-type`

JavaScript syntax name: `listStyleType`

Possible values:	`disc, circle, square, decimal, lower-roman, upper-roman, lower-alpha, upper-alpha, none`
Initial value:	`disc`
Applies to:	elements with `display` property value of `list-item`
Inherited:	yes
Percentage values:	N/A

This property describes how list items (that is, elements with a `display` value of `list-item`) are formatted.

This property can be set on any element, and its children will inherit the value. However, the list style is only displayed on elements that have a `display` value of `list-item`. In HTML this is typically the case for the `<LI>` element.

CSS Syntax Example

```
<STYLE TYPE="text/css">
UL.BLUELIST {color:blue;}
UL.BLUELIST LI {color:aqua;list-style-type:square;}
OL.REDLIST {color:red;}
OL.REDLIST LI {color:magenta; list-style-type:upper-roman;}
</STYLE>
```

JavaScript Syntax Example

```
<STYLE TYPE="text/javascript">
classes.BLUELIST.UL.color="blue";
contextual(classes.BLUELIST.UL, tags.LI).color="aqua";
contextual(classes.BLUELIST.UL, tags.LI).listStyleType="square";
classes.REDLIST.OL.color="red";
contextual(classes.REDLIST.OL, tags.LI).color="magenta";
contextual(classes.REDLIST.OL, tags.LI).listStyleType="upper-roman";
</STYLE>
```

Style Sheet Use

```
<UL CLASS=BLUELIST> <!-- LI elements inherit from UL -->
<LI>Consulting
<LI>Development
<LI>Technology integration
</UL>
<OL CLASS=REDLIST> <!-- LI elements inherit from OL -->
<LI>Start the program.
<LI>Enter your user name and password.
<LI>From the File menu, choose the Magic command.
</OL>
```

White Space

CSS syntax name: `white-space`

JavaScript syntax name: `whiteSpace`

Possible values:	`normal, pre`
Initial value:	according to HTML
Applies to:	block-level elements
Inherited:	yes
Percentage values:	N/A

This property declares how white space inside the element should be handled. The choices are:

- `normal` (white space is collapsed),
- `pre` (behaves like the `<PRE>` element in HTML) .

For example:

```
P.KEEPSPACES {white-space:pre;} /* CSS syntax */
classes.KEEPSPACES.P.whiteSpace = "pre"; /* JavaScript syntax */
```

Units

This section discusses units of measurement.

Length Units

The format of a length value is an optional sign character ('+' or '-', with '+' being the default) immediately followed by a number followed by a unit of measurement. For example, `12pt`, `2em`, `3mm`.

There are three types of length units: relative, pixel and absolute. Relative units specify a length relative to another length property. Style sheets that use relative units will scale more easily from one medium to another (for example, from a computer display to a laser printer). Percentage units and keyword values (such as `x-large`) offer similar advantages.

Child elements inherit the computed value, not the relative value, for example:

```
BODY {font-size:12pt; text-indent:3em;}

H1 {font-size:15pt;}
```

In the example above, the text indent value of `H1` elements will be 36pt, not 45pt.

The following relative units are supported:

- `em` -- the height of the element's font, typically the width or height of the capital letter M
- `ex` -- half the height of the element's font, which is typically the height of the letter 'x'
- `px` -- pixels, relative to rendering surface

The following absolute units are supported:

- `pt` -- points
- `pc` -- picas
- `px` -- pixels
- `in` -- inches
- `mm` -- millimeters
- `cm` -- centimeters

Color Units

A color value is a either a color name or a numerical RGB specification.

The suggested list of color names is: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. These 16 colors are taken from the Windows VGA palette and will also be used in HTML 3.2.

```
tags.BODY.color = "black";

tags.backgroundColor = "white";

tags.H1.color = "maroon";

tags.H2.color = "olive";
```

You can specify an RGB color by a six digit hexadecimal number where the first two digits indicate the red value, the second two digits indicate the green value, and the last two digits indicate the blue value. For example:

```
BODY {color: #FF0000}; /* red */

BODY {background-color:#333333";} /* gray */
```

You can also specify an RGB color by using the `rgb()` function which takes three arguments, for the red, green, and blue values. Each color value can either be an integer from 0 to 255 inclusive, or a percentage, as in this example:

```
P {color: rgb(200, 20, 240);) /* bright purple */

BLOCKQUOTE {background-color: rgb(100%, 100%, 20%); /* bright yellow */
```

Chapter

6

Advanced Style Sheet Example

This chapter presents an advanced example that uses style sheets. The example web page discussed in this chapter is the home page for a fictional company called Style Sheets Ink.

You can view this page at:

styleink/index.htm *StyleSheetExample*

The page opens in a separate browser window. If you do not see the page after selecting the link, check your desktop in case the second browser window is hidden under this one.

This chapter discusses how the `index.htm` page uses style sheets.

However, Style Sheets Ink has also developed several alternative home pages, that each display the same content but use slightly different style sheets. To view the alternative home pages, select the following links:

styleink/version1.htm *StyleSheetExample*

styleink/version2.htm *StyleSheetExample*

styleink/version3.htm *StyleSheetExample*

Feel free to copy any of these examples and modify them to suit your needs.

Style Sheets Ink Home Page

To view the web page that is discussed in this chapter, select:

styleink/index.htm *StyleSheetExample*

The example page opens in a separate browser window, so if you do not see it immediately, check if it is hidden under another window on your desktop. Be sure to view the sample page in a web browser that supports style sheets, such as Navigator 4.0, so you can see the full effects of the styles.

The rest of this chapter discusses how style sheets are used in Style Sheets Ink's home page. The discussions include extracts of source code. However, to see the entire source code, view the page source in your web browser.

The style sheet for the page uses CSS syntax. The style sheet is included at the top of the page.

The Style Sheets Ink home page has several sections, including an introductory section, a training section, a web sites section, and a consultation section, which are all contained within a main block. There is also a background section which is in the back, outside the main block.

The introductory section is centered in the main block, but the sections after it alternate between being on the left and the right.

The example page makes extensive use of `<DIV>` tags to contain elements that share styles. It also illustrates how you can use a `<DIV>` block to draw a single border around multiple elements.

Overview of the Style Sheet

At the very top of the style sheet file, there's a link to a font definition file:

```
<LINK HXBURNED REL="fontdef" SRC="index.pfr">
```

This font definition file contains the definition for the Impact BT downloadable font, which is used in the page. (For more information about downloadable fonts, see Part 3. Downloadable Fonts.)

The style sheet defines several styles that are used in different parts of the page. For instance, the INTROBLOCK style is used for the introductory material, the TRAININGHEAD style is used for the heading in the training section, and the TRAINING style is used for the text in the training section.

However, the style sheet also defines a couple of styles that are used throughout the whole document. These include styles for the <BODY> element and for the <H1> element.

The body of the Style Sheets Ink home page has a medium blue background. This could be specified using the bgColor attribute in the <BODY> element, but Style Sheets Ink has instead specified a style for the <BODY> element:

```
<STYLE type="text/css">

BODY {background-color:#CCDDFF;}
```

Nearly all <H1> elements in the document use the same customized style, so the style sheet defines the style for first-level headings as follows:

```
H1 {
 font-size:18pt;
 font-weight:bold;
 font-style:italic;
 font-family:"Impress BT", "Helvetica", sans-serif;
}
```

The font-family property lists three fonts. The font Impress B" is defined in the font definition file index.pfr, which is automatically downloaded to the user's system when a user views the page. However, just in case the font definition file is not available for any reason, Helvetica is specified as a backup font. Many computers include Helvetica as a system font, so it is likely to be available for most users. But just in case the font definition file is not available and the user does not have Helvetica font on their system, the style specifies the generic sans-serif font family as a last resort.

Th style defines the default font size, the font weight, font style, and font family for all <H1> element in the page. It does not define the font color. Throughout the document, each <H1> element gets its color from other inherited styles. For example, the training heading is inside a <DIV> block that uses the TRAINING style. This style sets the color property to #111100 (a dark gold color). Thus the training heading gets some of its characteristics from the H1 style, and other characteristics from the TRAINING style.

Main Block

The very first thing in the body of the page is a <DIV> block that contains the main content for the page.

This DIV block has a gray border and a white background. It uses the MAIN style to define its border and background: The definition of the MAIN style is:

```
all.MAIN {
 background-color: white;
 margin-left:5%; margin-right:5%;
 border-color:gray; border-style:outset; border-width:6pt;
 padding:20 pt;
}
```

The Introductory Section

The MAIN <DIV> block contains another <DIV> block. This block is the intro block, which contains the introductory information for the page. The intro block uses the style INTROBLOCK.

This style defines a flat blue border and a blue background for the intro block. The color of the border is the same as the color of the background. The style also defines the text and font characteristics to be used by all elements inside the intro block.

Here's the definition of the style class INTROBLOCK:

```
all.INTROBLOCK {
 font-family: "new century schoolbook", serif;
 font-style:italic;
 font-size:12pt;
 color:#000055;
 background-color: #CCDDFF;
 margin-left:5%; margin-right:5%;
```

```
 border-color:#CCDDFF; border-style:solid;
 border-width:2pt;
 padding:10pt;
}
```

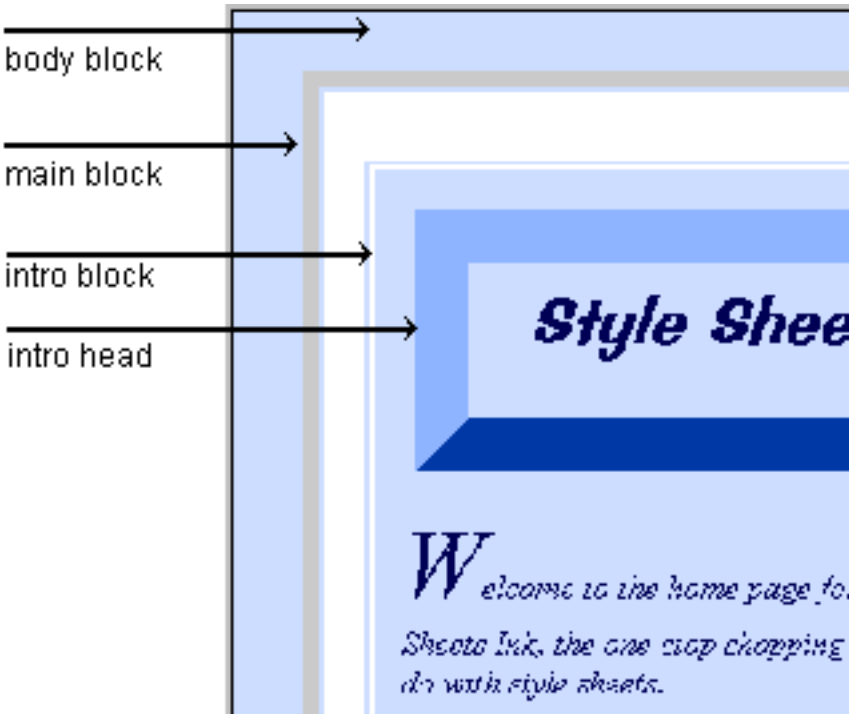

Figure 6.1 Blocks used in the Style Sheets Ink Home Page

Intro Head

The main heading for the page is inside the intro block. It has a wide outset 3D blue border. It uses the style `INTROHEAD`. Here's the definition of the style class `INTROHEAD`:

```
all.INTROHEAD{
```

```
 font-size:24pt;
 text-align:center;
 color:#000055;
 background-color:#CCDDFF;
 margin-left:2%; margin-right:2%;
 border-color:#0055FF; border-style:outset; border-width:20pt;
 padding:5pt;
}
```

Text in the Intro Block

The following code shows the first few lines in the body of the document:

```
<BODY >
<DIV CLASS=MAIN>
<DIV CLASS=INTROBLOCK>
<H1 CLASS=INTROHEAD>Style Sheets Ink.</H1>
```

The first letter of the first paragraph in the intro block needs to be extra large, so Style Sheets Ink uses a `<SPAN>` tag to apply the `INITCAP` style class to the first letter, as shown here:

```
<P STYLE="text-indent:0%;"><SPAN CLASS=INITCAP>W</
SPAN>elcome to the home page for our company, Style Sheets
Ink,...
```

The following code shows the definition of the style `INITCAP`:

```
all.INITCAP {font-size:36pt;}
```

All the paragraphs in the intro block inherit their styles (font styles and so on, not margins or paddings) from the enclosing element, which is the `DIV` block that uses the `INTROBLOCK` style.

The `text-indent` property is not inherited. The first line of each paragraph in the intro block (except for the first one) needs to be indented by ten percent. This could be achieved by specifying a local style for each paragraph as follows:

```
<P STYLE="text-indent:10%;>content...
```

However, several paragraphs need to be indented. Their best plan is to define a class of style, and use that style in each paragraph as appropriate. Although the amount of typing needed ends up being about the same, it is better to use a style class. That way, you can make changes to the style definition in one place, and those changes will be automatically reflected everywhere the style is used.

Thus you can define a simple style called INTROTEXT as follows:

```
all.INTROTEXT{text-indent:10%;}
```

Each paragraph that needs to be indented uses this style, for example:

```
<P CLASS=INTROTEXT>

At Style Sheets Ink we believe in the power of style sheets. We are
jazzed and excited at the myriad of ways that style sheets can liven up
a web site. We provide many services to help your company come up to
speed with using style sheets, including:

</P>
```

List of Services

The intro block includes a list of services offered by Style Sheets Ink. These services are presented in an unordered list.

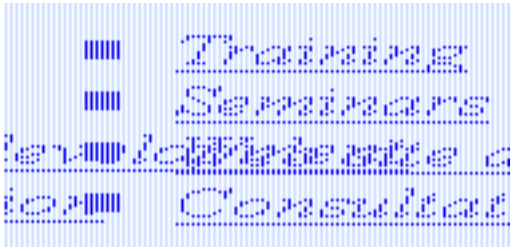

Figure 6.2 List of services

Style Sheets Ink specified the SQUAREDISCS class of style for the <UL> element so it is inherited by the <LI> element inside the <UL> element. (An alternative approach would be to specify the SQUAREDISCS style class for each <LI> element.)

The following code shows the definition of the SQUAREDISCS style:

```
all.SQUAREDISCS {list-style-type:square; color:green;}
```

The following code shows the body text that lists the services:

```
<UL CLASS=SQUAREDISCS>
<LI><A HREF="#TRAINING">Training</A>
<LI><A HREF="#SEMINAR">Seminars</A>
<LI><A HREF="#WEBDEV">Web site development</A>
<LI><A HREF="#CONSULTATION">Consultation</A>
</UL>
```

End of the Intro Block

At the end the intro block, there is a `</DIV>` tag that matches the `<DIV CLASS=INTROBLOCK>` tag. Notice that the border characteristics specified by the `INTROBLOCK` style apply to the `DIV` block as a whole, not to each individual element within the `DIV` block. Thus the entire `DIV` block is enclosed in a box with a blue background and a thin, flat, blue border.

The Training Section

Following the intro block is the training section, which displays the training heading on the left. The information about training wraps around the heading on the right.

Figure 6.3 The Training Section

The entire training section is contained within a `DIV` block that uses the `TRAINING` style. This style sets the text color, the left margin, and the right margin.

The definition of the `TRAINING` style is:

```
all.TRAINING{
 color:#111100;
 margin-right:30%;
 margin-left:5%;
```

```
}
```

The reason for setting the margins is to offset the contents of the training section from the edge of the surrounding block. The training section uses a floating element for the heading, and it's not wise to specify the `margin-left` property on an element if you also specify its `float` property. Therefore we put the floating heading inside a `DIV` block that has a left margin.

The heading for the training section floats to the left. It uses the `TRAININGHEAD` style, which specifies the color, the background image, the border and padding characteristics, and the float property. There's no need to specify the font size, font weight (bold) and font style (italic) since they are inherited from the style assigned to all `H1` tags. There's also no need to specify the color, because it is inherited from the `TRAINING` style. (However, if you wanted the heading to have a different color from the body text, you would need to specify the color here.)

The following code shows the definition of the `TRAININGHEAD` style:

```
H1.TRAININGHEAD {
 background-image:url(trainbg.gif);
 border-color:#666600;
 border-width:5pt;
 border-style:outset;
 padding:10pt;
 float:left;
}
```

The vertical effect in the heading is achieved simply by putting a `<BR>` tag after each letter, as shown here:

```
<DIV CLASS=TRAINING>
<H1 CLASS=TRAININGHEAD>
T<BR>
R<BR>
A<BR>
I<BR>
N<BR>
I<BR>
```

```
N<BR>

G

</H1>
```

All the paragraphs within the training section inherit their characteristics from the enclosing `DIV` block which uses the `TRAINING` style. So there's no need to specify which style these paragraphs need to use.

The training text wraps around the training heading. It doesn't reach all the way to the right since the `margin-right` property on the `TRAINING` style is set to 30%.

Just before the final `</P>` in this section, include a `<BR CLEAR>` tag, to ensure that the next element will not continue wrapping around the training heading.

The following code shows the paragraphs in the training section. Note the use of the `<SPAN>` tag to apply the `INITCAP` style to the first letter in the first paragraph.

```
<P ><SPAN CLASS=INITCAP>W</SPAN>e can build customized training courses
for you, to show you how useful style sheets can be.

</P>

<P >We also run regularly scheduled training courses at our offices that
are just jam-packed with information about style sheets. The training
course is very hands-on. Each participant has their own computer, and we
accept no more than ten students per class. The training courses usually
run for one full day, or two half days.

<BR CLEAR>

</P>

<!-- this ends the training section -->

</DIV>
```

The Seminars Section

Next comes the seminars section, which is very similar in style and structure to the training section. However, since the seminars section appears on the right, the `SEMINARHEAD` style sets the `float` property to `right`. Also, the

`SEMINAR` style sets the `margin-left` property to `30%` and the `margin-right` property to `10%`, so that the seminars section appears on the right of the main block.

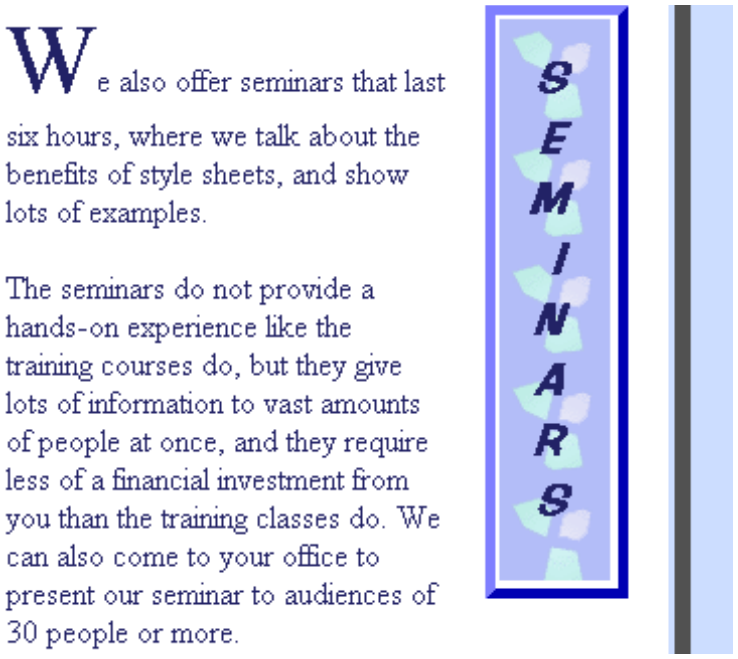

Figure 6.4 The Seminars Section

The seminar section includes a list of seminars:

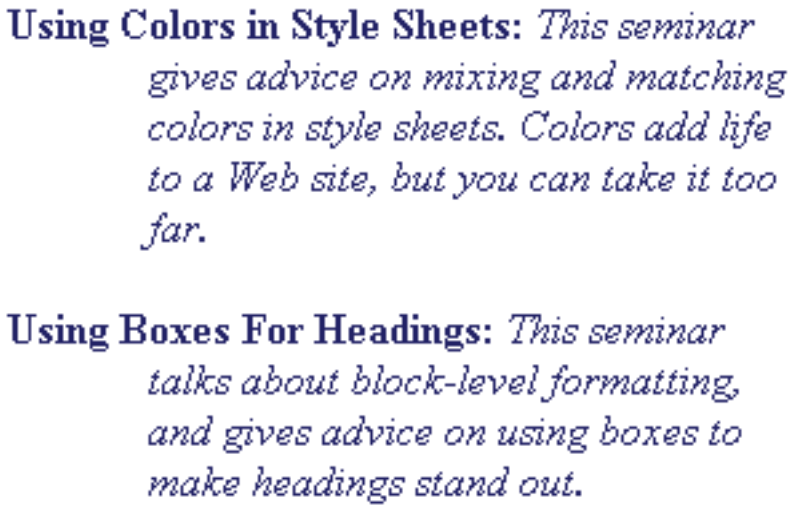

Figure 6.5 Outdented Items in the List of Seminars

The first line of each item in this list is outdented. This effect is achieved by using the `SEMINARLIST` style. This style sets the `margin-left` property and sets a `text-indent` value equal to minus the left margin, as shown here:

```
all.SEMINARLIST{margin-left:40pt; text-indent:-40pt;}
```

In the body text, each paragraph in the list of seminars uses the `SEMINARLIST` style, as shown below:

```
<P>Here is a list of available seminars:</P>
<P CLASS=SEMINARLIST> <B>Using Colors in Style Sheets: </B>
<I>discussion of this seminar...</I></P>
<P CLASS=SEMINARLIST> <B>Using Boxes For Headings:</B>
<I>discussion of this seminar...</I></P>
<P CLASS=SEMINARLIST> <B>Using Text Properties of Style Sheets: </B>
<I>discussion of this seminar...</I></P>
```

Note, however that you could achieve the same result by enclosing the paragraphs in a DIV block that uses the SEMINARLIST style, and then there would be no need to individually specify the SEMINARLIST class for each paragraph.

Web Sites and Consultation Sections

These two sections use the same layout and style structure as the training and seminars section.

The Background Block

At the bottom of the page, you see an explanatory paragraph that's in the main body of the page. This paragraph is at the top level (that is, it's directly in the BODY element.) It uses the INBACK style.

Although this paragraph is technically at the top level, it appears to live in the background, since it follows a big block with an outset 3D border.

Trouble-shooting Hints

In general, when you're working with style sheets, be sure to match opening and closing tags correctly. While web browsers are often fairly forgiving of HTML syntax mistakes, the browsers become very much stricter when style sheets are involved.

In particular, extraneous closing tags may end up closing other tags that you would not expect them to close. For example, in the code below, the extraneous </H3> tag may close the opening <DIV STYLE=INNERBLOCK> tag, and the second paragraph will thus be outside the inner block.

```
<DIV STYLE=INNERBLOCK>

<P>Here is some text. </P>

</H3>

<P>Here is some more text which is supposed to be in the innerblock.</P>
```

```
</DIV>
```

Be careful when using<A> and </A> tags in documents that use style sheets. For example, when you use DIV blocks with style sheets, don't start an <A HREF> tag before the start of the DIV block and then close it inside the DIV block, or you will get unpredictable results.

For example, the following code behaves as you would expect:

```
<DIV STYLE="margin-left:5%">
<A NAME="TRAINING">
<H1 CLASS=TRAININGHEAD>
content...</H1>
</A>
```

However, the code below has unpredictable results, because the <A HREF> and </A> tags are not in the correct places. (For example, the </A> tag may be used to close the <DIV> tag.)

```
<A NAME="TRAINING">
<DIV STYLE="margin-left:5%">
<H1 CLASS=TRAININGHEAD>
content...</H1>
</A>
```

Part 2. Positioning HTML Content

Contents

Chapter

7

Introduction

This chapter introduces the concept of using positioned blocks or layers of HTML content, and looks at the ways to define positioned blocks of HTML content.

Throughout this document, the terms *layer* and *positioned block of HTML content* are used interchangeably.

Overview

Netscape Navigator 4.0 introduces functionality that allows you to define precisely positioned, overlapping blocks of transparent or opaque HTML content in a web page.

You can write JavaScript code to modify these blocks of HTML content, or *layers*. Using JavaScript, you can move them, hide them, expand them, contract them, change the order in which they overlap, and modify many other characteristics such as background color and background image. Not only that, you

can change their content, and you can create new layers on the fly. Basically, you can use HTML and JavaScript to create dynamic animations on a web page and to create self-modifying web pages.

Using JavaScript and positioned blocks of HTML content, you can achieve dynamic animations directly in HTML. For example, layers can move, expand, and contract. You could also have many overlapping layers that can be dynamically peeled away to reveal the layer underneath.

Layers can be stacked on top of each other, and they can be transparent or opaque. If a layer is transparent, the content of underlying layers shows through it. You can specify background images and background colors for layers just as you can for the body of an HTML document.

Layers can be nested inside layers, so you can have a layer containing a layer containing a layer and so on.

Netscape Navigator 4.0 offers two ways to dynamically position HTML layers:

- Defining a style that has a `position` property
- Using the `<LAYER>` tag

A document can contain both layers that are defined as styles and layers that are defined with the `<LAYER>` tag. Also, if a layer is defined with the `<LAYER>` tag, it can use make use of styles.

The rest of this chapter discusses how to position a block of HTML content using styles, and then discusses how to do it using the `<LAYER>` tag.

Positioning HTML Content Using Styles

You can use styles to position blocks of HTML content. Part 1. Style Sheets talks about style sheets in general.

This section talks about using cascading style sheet (CSS) syntax to define styles for positioned blocks of HTML content. To see the original W3C Specification on using cascading style sheets for positioning blocks of HTML content, select:

http://www.w3.org/pub/WWW/TR/WD-positioning

Cascading style sheets are implemented in browsers from multiple vendors, while the <LAYER> tag may not be supported in non-Netscape browsers.

The style for a positioned block of HTML content always includes the property position. The value can be either absolute, which indicates a layer with an absolute position in its containing layer, or relative, which indicates a layer with a position relative to the current position in the document.

You can also specify the top and left properties to indicate the horizontal indent from the containing layer (for an absolutely positioned layer), or the current position in the document (for a relatively positioned layer).

A style that indicates a positioned block of HTML content *must* specify a value for the position property. Other than that, you can define the style however you like within the rules of defining style sheets. (See Part 1. Style Sheets for a full discussion of defining style sheets.)

If your document contains one or more layers with absolute positions, these layers are unlikely to share styles, since each one will need its own specific value for top and left to indicate its position. The use of individual named styles can be very useful for defining layers, since you can define a named style for each layer. (A named style is the same as a style with a unique ID.)

For example, the following <STYLE> tag defines styles for two layers. The layer named layer1 is positioned 20 pixels from the top of the page and 5 pixels in from the left. The layer named layer2 is positioned 60 pixels down from the top, and 250 pixels in from the left.

```
<STYLE TYPE="text/css">
<!--
#layer1 {position:absolute;
 top:20px; left:5px;
 background-color:#CC00EE;
 border-width:1; border-color:#CC00EE;
 width:200px;
 }
#layer2 {position:absolute;
 top:60px; left:250px;
 background-color:teal;
```

```
 width:200px;
 border-width:2px; border-color:white; }
}
-->
</STYLE>
```

Any style that specifies a value of absolute or relative for its `position` property defines a positioned layer. You use a layer style as you would use any other style in the body of your document. However, bear in mind that the idea of a layer is to act as a single entity of content. If you want your layer to contain more than one element, you can apply the layer style to a containing element, such as `DIV` or `SPAN`, that contains all the content.

For example:

```
<BODY BGCOLOR=white>
<DIV ID=layer1>
 <H1>Layer 1</H1>
 <P>Lots of content for this layer.</P>
 <IMG SRC="images/violets.jpg" align=right>
 <P>Content for layer 1.</P>
<P>More Content for layer 1.</P>
</DIV>
<P ID=layer2>Layer 2</P>
```

The following example uses the `STYLE` attribute directly in an element to specify that the element is a positioned layer:

```
<DIV STYLE="position:absolute; top:170px; left:250px;
 border-width:1px; border-color:white;
 background-color:#6666FF">
<H1>Layer 3 </H1>
<P>This is a blue block of HTML content.</P>
</DIV>
```

If you understand how to use style sheets to define styles, you can use the power of style sheets to define your layers. For example, you could create a colorful layer with a ridge-style 3D border as follows:

```
#layer4 {position:absolute;
 top:300px; left:100px;
 color:magenta;
 background-color:yellow;
 border-width:20px; border-color:cyan;
 border-style:ridge;
 padding:5%;
}
<BODY>
<DIV ID=layer4>
 <H1>Layer 4 </H1>
 <P>I am a very colorful layer.</P>
</DIV>
</BODY>
```

If you define a style with an absolute position, don't set margins for it, since it will get its position from the `top` and `left` properties.

For a full discussion of style sheets, see Part 1. Style Sheets.

To see the results of using the styles discussed so far in this section, select:

layercs1.htm lewin

The example opens a new Web browser window, so if you press the link and nothing seems to happen, have a hunt about on your desktop for the second Web browser window.

You can view the source code for `layerscs1.htm` to see the entire code for the examples.

Positioning HTML Content Using the <LAYER> Tag

Navigator 4.0 supports an alternative syntax for positioning blocks of HTML content. This syntax extends HTML to include the <LAYER> tag.

You can specify the position and content of a layer of HTML inside a <LAYER> tag in the body of the page -- there is no need to pre-define the layer before you specify the content for it. You can specify attributes for the layer such as ID, TOP, LEFT, BGCOLOR, WIDTH, and HEIGHT. (This is not a complete list of attributes -- all the attributes are discussed in Chapter 8, "Defining Positioned Blocks of HTML Content.")

At the time of writing, the <LAYER> tag is specific to the Netscape Navigator 4.0+ web browser. Other browser may not handle layers defined with the <LAYER> tag property.

When using the <LAYER> tag, you can use inline JavaScript in the layer definition, so for example, you can position layers relative to each other, such as having the top of one layer start just below the bottom of another.

The following code gives an example of the use of the <LAYER> tag.

```
<!-- default units for TOP, LEFT, and WIDTH is pixels -->
<LAYER ID=layer1 TOP=20pt LEFT=5pt
  BGCOLOR="#CC00EE" WIDTH=200>
 <H1>Layer 1</H1>
 <P>Lots of content for this layer.</P>
 <IMG SRC=violets.jpg align=right>
 <P>Content for layer 1.</P>
 <P>More Content for layer 1.</P>
</LAYER>
<LAYER ID=layer2 TOP=60 LEFT=250 BGCOLOR=teal WIDTH=200>
 <P>Layer 2</P>
</LAYER>
<LAYER ID=layer3 TOP=170 LEFT=250 BGCOLOR="#6666FF">
 <H1>Layer 3</H1>
```

```
 <P>This is a blue block of HTML content.</P>
</LAYER>
```

You can use the <LAYER> tag in conjunction with styles to create stylized layers. For example, the following code creates a colorful style class and applies it to a layer created with the <LAYER> tag:

```
<STYLE TYPE="text/css">
<!--
 all.style4 {
 color:magenta;
 border-width:20px; border-color:cyan;
 border-style:ridge;
 padding:5%;
}
-->
</STYLE>
<BODY BGCOLOR=white>
<LAYER ID=layer4 TOP=300 LEFT=100 BGCOLOR=yellow
    CLASS=style4>
 <H1>Layer 4 </H1>
 <P>I am a very colorful layer.</P>
</LAYER>
</BODY>
```

To see the results of using the styles discussed so far in this section, select:

layertg1.htm lewin

You can view the source code for `layerstg1.htm` to see the entire code for the examples.

Chapter

8

Defining Positioned Blocks of HTML Content

This chapter discusses how to specify either absolute or relative positions for blocks of HTML content. It lists all the characteristics you can specify for a positioned block of HTML content, describes the <NOLAYER> tag, and discusses the behavior of applets, plug-ins, and forms in positioned blocks of HTML content.

Absolute versus Relative Positioning

A layer can have an absolute position or a relative position.

Absolute Positioning

If a layer has an absolute position, you can specify its position within its containing layer, or within the document if it is not inside another layer. You define the exact position of the top, left corner of the layer by setting the `left` and `top` attributes or properties.

For a layer with absolute position, if you do not provide values for the `left` and `top` attributes or properties, they default to the value of the current position in the containing layer. For a layer at the top level, you can think of the document as the containing layer.

A layer with an absolute position is considered out-of-line in that it can appear anywhere in an HTML document, and does not take up space in the document flow.

To create a layer with an absolute position, use the `<LAYER>` tag with a matching `</LAYER>` tag to identify the end of the layer. For layers defined as styles, create a layer with an absolute position simply by specifying the `position` property as `absolute`. For example:

```
<LAYER ID=layer1 TOP=200 LEFT=260>
 <P>Layer 1 content goes here</P>
</LAYER>


<STYLE type="text/css">
<!--
#layer1 {position:absolute; top:200px; left:260px;}
-->
</STYLE>
```

Relative Positioning

A layer with a relative position is known as an inflow layer, and it appears wherever it naturally falls in the flow of the document. Inflow layers are considered to be both inflow, because they occupy space in the document flow, and inline, because they share line space with other HTML elements. If you want an inflow layer to appear on a separate line, you can insert a break before the layer, or wrap the layer in the `<DIV>` tag.

For layers with relative positions, you can use the `left` and `top` attributes or properties to specify the offset of the layer's top-left corner from the current position in the document.

To create an inflow layer, you can use the `<ILAYER>` tag with a closing `</ILAYER>` tag. For layers defined as styles, create an inflow layer by specifying the `position` property as `relative`.

For example:

```
<ILAYER ID=layer2>
 <P>Layer 2 content goes here</P>
</ILAYER>


<STYLE type="text/css">
<!--
#layer2 {position:relative; }
-->
</STYLE>
```

Attributes and Properties

This section lists all the attributes or properties that you can specify when defining layers, whether you use the `<LAYER>` and `<ILAYER>` tags to create layers, or you define layers as styles. (This list only includes only those properties that are relevant to layers. A style definition for a layer can include any style property. See Chapter 5, "Style Sheet Reference," for a list of all the other style sheet properties.)

For the sake of simplicity, in this section the term *parameter* means either an HTML attribute or a style property. For example, the `ID` parameter means either the `ID` attribute that can be used with the `<LAYER>` tag or the `ID` style property. Whenever the term *attribute* is used, it means an attribute for an HTML tag. Whenever the term *property* is used, it means a style property.

The `<LAYER>` tag always uses pixels as the unit of measurement for attributes that specify a distance. You do not need to specify the measurement units. For style properties however, you should always specified measurement units for properties that have numerical values.

- POSITION
- ID
- LEFT and TOP
- PAGEX and PAGEY
- SRC and source-include
- Z-INDEX, ABOVE and BELOW
- WIDTH
- HEIGHT
- CLIP
- VISIBILITY
- BGCOLOR and BACKGROUND-COLOR
- BACKGROUND and BACKGROUND-IMAGE
- OnMouseOver, OnMouseOut
- OnFocus, OnBlur
- OnLoad

POSITION

```
#block1 {position:absolute;}
#block2 {position:relative;}
```

The `position` property applies only to layers defined as styles. It indicates that the style represents a positioned block of HTML. Its value can be either `absolute` or `relative`.

A style whose `position` property is `absolute` creates a layer similar to one created by the `<LAYER>` tag. A style whose `position` property is `relative` creates a layer similar to one created by using the `<ILAYER>` tag.

ID

```
<LAYER ID=block1>
#block1 {position:absolute;} /* CSS */
```

The `ID` parameter is an identification handle, or name, for the layer. The `ID` must begin with an alphabetic character. (The `ID` attribute was previously called `NAME`. The `NAME` attribute still works, but its use is discouraged, since it is only applicable to the `<LAYER>` tag).

You can use the layer's id as a name to refer to the layer from within HTML and from external scripting languages such as JavaScript.

This attribute is optional; by default, layers are unnamed, that is, they have no id.

LEFT and TOP

The `LEFT` and `TOP` parameters specify the horizontal and vertical positions of the top-left corner of the layer within its containing layer, or within the document if it is at the top level. Both parameters are optional. The default values are the horizontal and vertical position of the layer's contents as if it was not enclosed in a layer. The value must be an integer.

For layers with absolute positions, the origin is the upper-left corner of the document or containing layer, with coordinates increasing downward and to the right.

The default units for LEFT and TOP when used in the <LAYER> tag is pixels. When defining a layer as a style, however, you need to specify the units. For example:

<LAYER> Tag Syntax

```
<LAYER BGCOLOR="yellow" TOP=300 LEFT =70
  WIDTH=400 HEIGHT=200>
 <P>Paragraph in layer with absolute position.</P>
 <LAYER BGCOLOR=teal TOP=50 LEFT=20
   WIDTH=200 HEIGHT=100>
  <P>Paragraph in embedded layer with absolute position</P>
 </LAYER>
</LAYER>
```

CSS Syntax

```
<DIV STYLE="position:absolute; background-color:yellow;
 top:300px; left:70px; width:200px; height:200px;
 border-width:1;">
 <P>Paragraph in layer with absolute position.</P>
 <DIV STYLE="position:absolute; background-color:teal;
  top:30px; left:20px; width:150px; height:120px;
  border-width:1px;">
  <P>Paragraph in embedded layer with absolute position.</P>
 </DIV>
</DIV>
```

For layers with relative positions, the origin is the layer's "natural" position in the flow, rather than the upper-left corner of the containing layer or page. You can also use the LEFT and TOP parameters to offset a relatively positioned layer from its natural position in the flow, as shown in the following example.

<LAYER> Tag Syntax

```
<P>Paragraph above relatively positioned layer.</P>
<P><ILAYER LEFT=2>
This relatively positioned layer is displaced 2 pixels to the right of
```

```
its normal position.
</ILAYER></P>
<P>Paragraph below relatively positioned layer</P>
<P>This <ILAYER TOP=3>word</ILAYER> is nudged down 3 pixels.</P>
```

CSS Syntax

```
<P>Paragraph above relatively positioned layer.</P>
<P STYLE="position:relative; left:2px;">
This relatively positioned layer is displaced 2 pixels to the right of
its normal position.</P>
<P>Paragraph below relatively positioned layer.</P>
<P>This <SPAN STYLE="position:relative; top:3px;>word </SPAN> is nudged
down 3 pixels.</P>
```

The following code illustrates another example of relatively positioned layers defined as styles.

```
STYLE TYPE="text/css">
<!--
all.UP {position:relative; top:-10pt;}
all.DOWN {position:relative; top:10pt;}
-->
</STYLE>
<BODY>
<P>This <SPAN CLASS=DOWN>text </SPAN>goes <SPAN CLASS=UP>up</SPAN>
and <SPAN CLASS=DOWN>down, </SPAN>up
<SPAN CLASS=DOWN>and <SPAN CLASS=DOWN>down.</SPAN></SPAN>
</P>
</BODY>
```

To see the results of some of the examples given in this section, see:

updown.htm lewin

Using Inline JavaScript to Position Layers

When using the <LAYER> tag, you can use also inline JavaScript scripted expressions to position the layer. For example, you can position one layer relative to another.

The following example uses inline JavaScript code to define a layer whose ID is `suspect1`, and then defines another layer whose ID is `suspect2` that is positioned 10 pixels below the bottom of the first suspect.

```
<LAYER ID="suspect1">
    <IMG WIDTH=100 HEIGHT=100 SRC="suspect1.jpg">
    <P>Name: Al Capone
    <P>Residence: Chicago
</LAYER>
<LAYER ID="suspect2"
    LEFT=&{"&"};{window.document.suspect1.left};
    TOP=&{"&"};{window.document.suspect1.top +
        document.suspect1.document.height + 10};>
    <IMG WIDTH=100 HEIGHT=100 SRC="suspect2.jpg">
    <P>Name: Lucky Luciano
    <P>Residence: New York
</LAYER>
```

Notice these two points in the previous example:

- You need to use a semicolon outside the closing curly brace.
- You get the value of `top` from the layer, but you get the value of `height` from the layer's document.

Although you cannot use inline JavaScript within a style definition for a layer, you CAN use JavaScript to reposition such a layer after it has been defined.

PAGEX and PAGEY

```
<LAYER PAGEX=100 PAGEY=100>
```

These attributes are used only with the <LAYER> tag; there is no equivalent style property.

The PAGEX and PAGEY attributes specify the horizontal and vertical positions in pixels of the top-left corner of the layer relative to the enclosing document (rather than the enclosing layer.)

SRC and source-include

```
<LAYER SRC="htmlsource/meals/special.htm>
source-include:url("htmlsource/meals/special.htm"); /* CSS */
```

The SRC attribute for the <LAYER> tag and the source-include style property specify an external file that contains HTML-formatted text to be displayed in this layer. (Note that the source-include style property is not approved by W3C.)

The file specified can contain an arbitrary HTML document.

The following code shows an example of the use of the SRC attribute and include-source property.

CSS Syntax

```
<STYLE TYPE="text/css">
<!--
#layer1 {
 position:absolute;
 top:50pt; left:25pt; width:175pt;
 include-source:url("content1.htm");
 background-color:purple;
color:yellow; border-width:1; }
-->
</STYLE>
<BODY BGCOLOR=white>
<DIV ID=layer1>
</DIV>
```

<LAYER> Tag Syntax

```
<LAYER top=50 left=250 width=175
 src="content1.htm"
```

```
 BGCOLOR="#8888FF">
</LAYER>
</BODY>
```

To see the results of this example, select:

source1.htm lewin

The source file can include JavaScript code. Any layers in the source file are treated as child layers of the layer for which the source file is providing content.

Using an external source as the content of your layer is particularly useful if you want to dynamically change the content of the layer. For example, a restaurant might have a web page that uses a layer to describe the special meal of the day. Each morning, after the chef has decided what the special is going to be for the day, he or she quickly edits the file "special.htm" to describe the meal.

The chef doesn't have to rewrite the entire page just to update the information about the special of the day.

It can also be a very good idea to use external source as the content of a layer when you wish to provide alternative content for browsers that do not support layers. In that case, you can use the <NOLAYER> tag to enclose the content to be displayed on browsers that do not support layers, as illustrated in the section "The <NOLAYER> Tag."

WIDTH

```
<LAYER WIDTH=200>
<LAYER WIDTH="80%">
width:200px; /* CSS */
width:80%; /* CSS */
```

The WIDTH parameter determines the width of the layer at which the layer's contents wrap. The width can be expressed as an integer value, or as a percentage of the width of the containing layer.

Note, however, that if the layer contains elements that cannot be wrapped, such as images, that extend beyond the specified width, the actual width of the layer expands accordingly.

If this parameter is not specified, the layer contents wrap at the right boundary of the enclosing layer.

See Chapter 13, "Changing Wrapping Width Example," for an example of dynamically changing the wrapping width of a layer.

HEIGHT

```
<LAYER HEIGHT=200>>

<LAYER HEIGHT = "50%">

height:200px; /* CSS */

height:50%; /* CSS
```

The HEIGHT parameter determines the initial height of the clipping region of the layer. The height can be expressed as an integer value, or as a percentage of the height of the containing layer (or the window for a top-level layer.)

Note, however, that if the contents of the layer do not fit inside the specified height, the layer increases its height to include all its contents.

The main purpose of the HEIGHT parameter is to act as the reference height for children layers that specify their heights as percentages.

By default, the height is the minimum height that contains all the layer contents.

CLIP

```
<LAYER CLIP="20,20,50,100">

clip:rect(0,100,100,0); /* CSS */
```

The CLIP parameter determines the clipping rectangle of the layer, that is, it defines the boundaries of the visible area of the layer.

The value is a set of four numbers, each separated by a comma, and optionally enclosed in a string. If you omit the quotes, be sure not to have any white space between the four numbers. The numbers indicate the left value, the top value, the right value, and the bottom value in order. The left and right values are specified as pixels in from the left edge of the layer itself, while the top and bottom values are specified as pixels down from the top edge of the layer itself.

Each of the four values are numbers of pixels. You can also specify the value as a set of two numbers, in which case the left and top values default to 0. For example:

```
CLIP="10,20"
```

is equivalent to

```
CLIP="0,0,10,20"
```

If the CLIP attribute is omitted, the clipping rectangle of a layer is determined by the values of WIDTH, HEIGHT, and the content of the layer. If neither of these values are given, by default, the clip left value of a layer is 0; clip top is 0; clip right is the wrapping width, and clip height is the height required to display all the contents.

For an example of changing the clipping region of a layer, see Chapter 12, "Expanding Colored Squares Example.".

Z-INDEX, ABOVE and BELOW

```
<LAYER Z-INDEX=3>
```

```
<LAYER ABOVE=layer1>
```

```
<LAYER BELOW=greenlayer>
```

```
z-index:3; /* css */
```

The ABOVE and BELOW attributes are used with the <LAYER> tag. There are no corresponding style properties.

These parameters specify the z-order (stacking order) of layers. If you set one of these parameters, it overrides the default stacking order which is determined by placing new layers on top of all existing layers. Only one of the Z-INDEX, ABOVE, or BELOW parameters can be used for a given layer.

The Z-INDEX parameter allows a layer's z-order to be specified in terms of an integer. Layers with higher-numbered Z-INDEX values are stacked above those with lower ones. Only positive Z-INDEX values are allowed.

The ABOVE attribute specifies the layer immediately on top of a newly created layer; that is, the new layer is created just below the layer specified by the ABOVE attribute. (The ABOVE and BELOW attributes are not available in as style properties.)

Similarly, the BELOW attribute identifies the layer immediately beneath the newly created layer. For either attribute, the named layer must already exist. Forward references to other layers result in default layer creation behavior (as if the ABOVE or BELOW attribute had not appeared).

Currently all nested layers exist above their parent layer in the stacking order. The Z-INDEX, ABOVE and BELOW values are relative to sibling layers, that is, other layers that have the same parent layer.

For an example of changing the stacking order or z order of layers, see Chapter 11, "Swimming Fish Example."

VISIBILITY

```
<LAYER VISIBILITY=SHOW>

<LAYER VISIBILITY=HIDE>

<LAYER VISIBILITY=INHERIT>

visibility:show; /* css */

visibility:hide; /* css */

visibility:inherit; /* css */
```

The VISIBILITY parameter determines whether the layer is visible or not. A value of HIDE hides the layer; SHOW shows the layer; INHERIT causes the layer to have the same visibility as its parent layer. For top level layers (that is, layers that are not nested inside other layers), a value of INHERIT has the same effect as SHOW since the body document is always visible.

By default, a layer has the same visibility as its parent layer, that is, the value of the VISIBILITY attribute is INHERIT.

Remember that even if the visibility of a layer is set to SHOW, you will only be able to see the layer if there are no other visible, opaque layers stacked on top of it.

If the visibility of a relatively positioned layer is HIDE, the layer contents are not shown, but the layer still takes up space in the document flow.

For an example of making layers visible and invisible, see Chapter 10, "Fancy Flowers Farm Example."

BGCOLOR and BACKGROUND-COLOR

```
<LAYER BGCOLOR="#00FF00">

<LAYER BGCOLOR="green">

background-color:green;

background-color:00FF00;
```

The BGCOLOR attribute and background-color style property determine the solid background color of a block of HTML content, similar to the BGCOLOR attribute of the <BODY> tag. The background color is either the name of a standard color such as red or an RGB value, such as #334455 (which has a red hexadecimal value of 33, a green hexadecimal value of 44 and a blue hexadecimal value of 55.)

By default, a layer is transparent -- layers below it show through the transparent areas of the layer's text and other HTML elements.

If a layer is defined with the <LAYER> tag, its background color is applied to the rectangular region occupied by the layer. If a layer is defined as a style, the background color is applied only to the actual content of the layer, not to the entire region of the layer. If the style has a border, the region enclosed by the border uses the background color, but this region is still limited to the region that contains content. If the style specifies width and height values that define a region larger than is needed to display the content, the background color will only be applied to the area inside the border, which will be drawn around the actual content.

Netscape Navigator 4.0 also supports a layer-background-color CSS style property, which sets the background color of the entire layer, but this property is not approved by the W3C.

This is really hard to explain in words, but is immediately obvious when you see the results. To see an illustration of this point, click on:

bgtest.htm lewin

BACKGROUND and BACKGROUND-IMAGE

```
<LAYER BACKGROUND="images/dogbg.gif">
```

```
background-image:url("images/dogbg.gif"); /* CSS */
```

The BACKGROUND attribute and background-image style property indicate a tiled image to draw across the background of a block of HTML content. The value is the URL of an image.

By default, a layer is transparent -- layers below it show through the transparent areas of layer's text and other HTML elements.

Note that Netscape Navigator 4.0 also supports a `layer-background-image` CSS style property, which sets the background color of the entire block that uses the style, but this property is not approved by the W3C.

If a layer is defined with the `<LAYER>` tag, the background image is applied to the rectangular region occupied by the layer. If a layer is defined as a style, the background image is applied to the region that contains the actual content of the layer. If the style specifies width and height values that define a region larger than is needed to display the content, the background image will only be applied to the area that encloses the actual content.

Netscape Navigator 4.0 also supports a `layer-background-image` CSS style property, which draws the image across the entire layer, but this property is not approved by the W3C.

To see an illustration of this point, click on:

bgimage.htm lewin

OnMouseOver, OnMouseOut

These attributes only apply to the `<LAYER>` tag.

```
<LAYER OnMouseOver="highlight(); return false;">
```

```
<LAYER OnMouseOut="dehighlight(); return false;">
```

These are event handlers. Their values must be functions or inline JavaScript code. The `onMouseOver` handler is invoked when the mouse enters the layer, and the `onMouseOut` handler is invoked when the mouse leaves the layer.

For on example of using an `onMouseOver` handler, see Chapter 12, “Expanding Colored Squares Example.”

OnFocus, OnBlur

These attributes only apply to the `<LAYER>` tag.

```
<LAYER OnFocus="function1(); return false;">
```

```
<LAYER OnBlur="function2(); return false;">
```

These are event handlers. Their values must be functions or inline JavaScript code. The `onFocus` handler is invoked the layer gets keyboard focus, and the `onBlur` handler is invoked when the layer loses keyboard focus.

OnLoad

This attribute only applies to the <LAYER> tag.

```
OnLoad="dosomething(); return false;"
```

This is an event handler. Its value must be a function or inline JavaScript code. The onLoad handler is invoked when the layer is loaded, regardless of whether the layer is visible or not.

For an example of setting the onLoad handler for a layer, see Chapter 11, "Swimming Fish Example" and Chapter 12, "Expanding Colored Squares Example.".

The <NOLAYER> Tag

If an HTML file that contains positioned blocks of HTML content is displayed in a browser that does not know how to position content, the content is displayed as if it was not positioned. If the file contains any scripts that require layers functionality, they will generate JavaScript errors if loaded into a browser that does not support positioning.

You can use the <NOLAYER> and </NOLAYER> tags to surround content that is ignored by Netscape Navigator 4. This enables you to provide alternative content that will be displayed by browsers that cannot position content. For example:

```
<LAYER SRC=layerContent.html></LAYER>
<NOLAYER>
This page would show some really cool things if you had
a browser that can position content.
</NOLAYER>
```

Applets, Plug-ins, and Forms

Layers can contain form elements, applets, and plug-ins, which are known as windowed elements. These elements are special in that they float to the top of all other layers, even if their containing layer is obscured.

When a windowed element is moved to the edge of its containing layer, it disappears as soon as one of its borders hits a border of the layer, instead of seeming to glide out of view as non-windowed elements would do. For form elements, it is the individual element that disappears on contact with the border of the layer, not the entire form.

Note however, that windowed elements do move and change visibility in accordance with their containing layer.

Forms cannot span layers. That is, you cannot have part of the form in one layer and another part in another layer.

Communicator introduces windowless plug-ins, which are plug-ins that do not pop to the top of the window and can be drawn below other items in the window. Windowless plug-ins are discussed in the Plug-in guide.

Here's the URL for the Plug-in Guide:

/library/documentation/communicator/plugin/index.htm lewin

To link to Chapter 1, "Plug-in Basics," which contains a section called "Windowed and Windowless Plug-ins" see:

/library/documentation/communicator/plugin/pg1bas.htm lewin

To link to Chapter 4, "Drawing and Event Handling," which contains a section on general issues in writing windowless plug-ins, see:

/library/documentation/communicator/plugin/pg4dr.htm lewin

Chapter

9

Using JavaScript With Positioned Content

This chapter discusses how to use JavaScript to modify and animate positioned blocks of HTML content. First the chapter gives an overview of why you might want to use JavaScript to modify blocks of content, then it discusses the Layer object, which represents a block of content. It shows how to use JavaScript to create new blocks of content, and how to write content dynamically. It discusses how you can make distinct blocks of HTML respond to events. It discusses how each block of content can contain its own localized script, and finishes up by addressing some of the issues involved in animating HTML content.

This chapter does not teach the basics of using the JavaScript language, although it does provide several examples that should help you get started. For more information about JavaScript see:

- JavaScript 3.0 Guide:

http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.htmllewin

- What's New in JavaScript for Navigator 4.0:

/library/documentation/communicator/jsguide/js1_2.htm lewin

The remaining chapters in this part of the book each present a separate complete example of using JavaScript to work with positioned content.

Using JavaScript to Bring Your Web Pages to Life

Regardless of how you define your positioned blocks of HTML content, you can write scripts in JavaScript that move them, change their color and size, change their content, make them visible or invisible, and generally modify them in a variety of ways. Furthermore, you can use JavaScript to change the contents of a positioned block or create new ones on-the-fly.

Using JavaScript to work with positioned blocks of HTML content allows you to define animations directly in a web page. For example, you could create an animation that dynamically peels away a series of layers of content to reveal the one underneath. You can make blocks of content move across, over, and under other blocks of content. You can make them appear and disappear. You can make them dynamically expand and contract in response to mouse events. You can generally bring your web page alive with animated content.

You can use JavaScript to modify positioned blocks of HTML content regarless of how the blocks are defined. You can manipulate positioned blocks of HTML content with JavaScript, even if they are defined as styles.

The Layer Object

Regardless of how you define a positioned block of HTML content, it can be treated as a modifiable object in JavaScript.

For each layer in an HTML page (whether it is defined with the `<LAYER>` tag or as a style whose `position` property is either `absolute` or `relative`) there is a corresponding JavaScript `layer` object. You can write JavaScript scripts that modify layers either by directly accessing and modifying property values on the *layer* objects, or by calling methods on the `layer` objects.

The Document Property of Layers and the Layers Property of Documents

Each document object has a `layers` property that contains an array of all the top-level layers in the document. Each layer in turn has a `document` property.

This `document` property has a `layers` array that contains all the top-level layers inside this layer. The document of a layer also has all the usual properties of a document object, such as the `images` property, which is an array of all the images in the layer, as well as properties that are arrays for all the applets, embeds, links, and named anchors in that layer.

How Do You Refer to a Positioned Block of Content from JavaScript?

There are several ways you can access a layer from JavaScript. If you know the layer's id (or name) you can access it in the following ways:

- `document.`*layername*

  For example, the following expression returns the layer named `"flower-layer"`.

  ```
  document.flowerlayer
  ```

- `document.layers[`*layername*`]`

For example, the following expression returns the layer named `"flower-layer"`.

```
document.layers["flowerlayer"]
```

If you know the index for the layer you can access it as follows:

- `document.layers`[*index*]

  Note that the first layer has an index of 0, the second layer has an index of 1, and so on. The following expression returns the fourth layer in the document.

  ```
  document.layers[3]
  ```

When accessed by integer index, array elements appear in z-order from back to front, where zero is the bottom-most layer and higher layers are indexed by consecutive integers. The index of a layer is not the same as its `zIndex` property, as the latter does not necessarily enumerate layers with consecutive integers. Also, adjacent layers can have the same `zIndex` property values, but two layers can never occupy the same index in the array.

You can find the number of layers in a document or another layer array by obtaining its `length` property. For example, the following expression returns the number of top level layers in the document:

```
document.layers.length
```

The following expression returns the number of layers nested at the top level inside the layer named `"houses"`.

```
document.layers["houses"].document.layers.length
```

The Layer Object Properties

As with any JavaScript object, you can access the properties of a layer object using the following syntax:

layerObject.propertyName

where *layerObject* is an expression that evaluates to a layer object, and *propertyName* is the name of the property to be accessed. For example, the following expression returns the value of the `visibility` property of the layer named `"flowerlayer"`:

```
document.flowerlayer.visibility;
```

The following expression sets the `left` property of the layer named `"flowerlayer"` to 300 pixels.

```
document.flowerlayer.left=300;
```

The following table lists all the properties that you can use to access or modify a layer in JavaScript. Notice that there is only one set of property names. No matter whether a layer was created with the `<LAYER>` tag or was defined as a style, you can use the property names listed in the following table to access it or modify it after it has been created.

These property names are case-sensitive.

Table 9.1 Layer Object Properties

Property Name	Modifiable by user?	Description
`document`	No	Each layer object contains its own document object. This object can be used to access the images, applets, embeds, links, anchors and layers that are contained within the layer. Methods of the document object can only also be invoked to change the contents of the layer.
`name`	No	The name assigned to the layer through the `NAME` or `ID` attribute.
`left`	Yes	The horizontal position of the layer's left edge, relative to the origin of its parent layer (for layers with absolute positions) or relative to the natural flow position (for layers with relative positions). The value can be an integer such as `12`, or a percentage, such as `" 25%"`. The default unit of measurement is pixels.
`top`	Yes	The vertical position of the layer's top edge relative to the origin of its parent layer. The value can be an integer, an integer such as `12`, or a percentage, such as `" 25%"`. The default unit of measurement is pixels.
`pageX`	Yes	The horizontal position of the layer relative to the page. The default unit of measurement is pixels.

Table 9.1 Layer Object Properties

Property Name	Modifiable by user?	Description
`pageY`	Yes	The vertical position of the layer relative to the page. The default unit of measurement is pixels.
`zIndex`	Yes	The relative z-order of this layer with respect to siblings. Sibling layers with lower numbered z-index's are stacked underneath this layer. The value must be `0` or a positive integer.
`visibility`	Yes	Determines whether or not the layer is visible. A value of `"show"` means show the layer; `"hide"` means hide the layer; `"inherit"` means inherit the visibility of the parent layer.

`clip.top`
`clip.left`
`clip.right`
`clip.bottom`
 	
`clip.width`
`clip.height`
	Yes	These properties define the clipping rectangle, which specifies the part of the layer that is visible. Any part of a layer that is outside the clipping rectangle is not displayed.
		The clipping region can extend beyond the area of the layer that contains content.
		Clipping values can be negative, 0, or positive integers.
		For example, to clip 10 pixels from the left edge, you would increase clip.left by 10. To reduce the clipping region by 20 pixels at the right edge, you would reduce clip.right by 20.
		The values for clip.top, clip.left, clip.bottom, and clip.right, are in the layer's coordinate system.
		Setting the clip.width value to w is the same as:
		`clip.right = clip.left + w;`
		Setting the clip.height to h is the same as:
		`clip.height = clip.top + h;`

Table 9.1 Layer Object Properties

Property Name	Modifiable by user?	Description
`background`	Yes	The image to use as the background for the layer. The image is tiled across the background of the layer. For example: `layer.background.src = "fishbg.gif";` The value is null if the layer has no backdrop.

`bgColor`	Yes	The color to use as a solid background color for the layer. The value can be an encoded RGB value, a string that indicates a pre-defined color, or `null` for a transparent layer
		For example:
		`//blue background`
		`layer.bgColor = "#0000FF";`
		`// red background`
		`layer.bgColor = "red";`
		`// transparent layer`
		`layer.bgColor = null;`

`siblingAbove`	No	The `layer` object above this one in the stacking order, among all layers that share the same parent layer or null if the layer has no sibling above.
`siblingBelow`	No	The `layer` object below this one in z-order, among all layers that share the same parent layer or null if layer is bottommost.

Table 9.1 Layer Object Properties

Property Name	Modifiable by user?	Description
`above`	No	The `layer` object above this one in z-order, among all layers in the document or the enclosing window object if this layer is topmost.
`below`	No	The `layer` object below this one in z-order, among all layers in the document or null if this layer is bottommost.
`parentLayer`	No	The `layer` object that contains this layer, or the enclosing window object if this layer is not nested in another layer.
`src`	Yes	Source of the content for the layer, specified as a URL.

The Layer Object Methods

There are several methods that you can use on a `layer` object to modify a layer. As with any JavaScript object, you can invoke a method on a `layer` object using the following syntax:

layerObject.methodName(args)

where *layerObject* is an expression that evaluates to a layer object, *methodName* is the method to be invoked, and *args* are the arguments to the method.

For example, the following expression invokes the method `moveBy()` on the layer named `flowerlayer`, to move the layer 10 pixels to the right and 10 pixels down from its current position.

```
document.flowerlayer.moveBy(10, 10);
```

The following table lists all the methods that you can use to access or modify a layer in JavaScript. You will notice that there is only one set of method names. It does not matter whether a layer was created with the `<LAYER>` tag or was defined as a style, you can use the methods listed in the following table to access it or modify it after it has been created.

These method names are **case-sensitive**

Table 9.2 Layer Object Methods

Method Name	Description
`moveBy(dx, dy)`	Moves this layer by `dx` pixels to the left, and `dy` pixels down, from its current position.
`moveTo(x, y)`	For layers with absolute positions, this method changes the layer's position to the specified pixel coordinates within the containing layer or document. For layers with relative positions, this method moves the layer relative to the natural position in the containing layer or document. This method is equivalent to setting both the `top` and `left` properties of the layer object.
`moveToAbsolute(x, y)`	Changes the layer position to the specified pixel coordinates within the page (instead of the containing layer.) This method is equivalent to setting both the `pageX` and `pageY` properties of the layer object.
`resizeBy(dwidth, dheight)`	Resizes the layer by the specified height and width values (in pixels). Note that this does not relayout any HTML contained in the layer. Instead, the layer contents may be clipped by the new boundaries of the layer. This method has the same effect as adding `dwidth` and `dheight` to the `clip.width` and `clip.height`.
`resizeTo(width, height)`	Resizes the layer to have the specified height and width values (in pixels). Note that this does not relayout any HTML contained in the layer. Instead, the layer contents may be clipped by the new boundaries of the layer. This method has the same effect as setting the `clip.width` and `clip.height`.
`moveAbove(layer)`	Stacks this layer (in z-order) above the layer specified in the argument, without changing either layer's horizontal or vertical position. After re-stacking, both layers will share the same parent layer. The value must be a valid layer object.
`moveBelow(layer)`	Stacks this layer (in z-order) below the specified layer, without changing the layer's horizontal or vertical position. After re-stacking, both layers will share the same parent layer. The value must be a valid layer object.

Table 9.2 Layer Object Methods

Method Name	Description
`load(sourcestring, width)`	Changes the source of a layer to the contents of the file indicated by sourcestring, and simultaneously changes the width at which the layer's HTML contents will be wrapped. This method takes two arguments. The first argument is a string indicating the external file name, and the second is the width of the layer in pixels.

Creating Positioned Blocks of Content Dynamically

You can use JavaScript to create new `layer` objects by calling the `new` operator on a `Layer` object, for example:

```
bluelayer = document.bluelayer;

newbluelayer = new Layer(300, bluelayer);
```

The first argument is the width of the new layer, and the second argument, which is optional, is its parent layer. The parent can also be a window, in which case the new layer is created as a top-level layer within the corresponding window. If you do not supply a parent layer, the new layer will be a top-level layer in the current document.

After creating a new layer, you can set its source either by setting a value for its `src` property, or by calling the `load` method. Alternatively, you can open the layer's document and write to it (as discussed in the next section.)

There are a few important things to know about creating layers and modifying their contents dynamically. You can create a new `layer` object by using the `new` operator *only after the page has completely finished loading*. You cannot open a layer's document and write to it until the page has finished loading. You can have only one layer open for writing at a time.

Writing Content in Positioned Blocks

While initially defining a layer, you can write to the layer's document using the document's `write` method.

```
<LAYER ID="layer1" BGcolor="green">
  <HR>
  <H1>First Heading</H1>
  <SCRIPT>
     document.write("<P>Here is some content<P>")
  </SCRIPT>
  <HR>
</LAYER>
```

After a layer has been initially created and the page has fully finished loading, you can modify the contents of the layer by using the `write`() method of the layer's document. If you use the `write()` method to write content to a layer after the layer has been created, the original content of the layer is wiped out, and replaced by the new content.

After writing to a layer's document, you need to close the document.

For example:

```
<LAYER ID="layer1" BGCOLOR="blue">
 <HR>
 <H1>First Heading</H1>
 <P>Here is the original content<P>
 <HR>
</LAYER>
</BODY>
</HTML>
<SCRIPT>
function changeLayerContent() {
 document.layer1.document.write("<HR><P>New content.</P><HR>");
```

```
 document.layer1.document.close();
}
</SCRIPT>

<FORM NAME="form">
<INPUT TYPE=button VALUE="CHANGE CONTENT"
ONCLICK='changeLayerContent();return false;'>
</FORM>
```

For a further example of writing to a layer, see Chapter 12, "Expanding Colored Squares Example."

Handling Events

Each layer can be thought of as a separate document. It has the same event-handling capabilities as a top-level window. You can capture events for a layer.

For an overview of event handling, see the section "Scripting Event Handlers" in the JavaScript guide for in JavaScript. The following link takes you to the JavaScript guide:

http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html lewin

When defining a layer with the `<LAYER>` tag, you can also supply the following attributes that specify event handlers:

`onMouseOver`

`onMouseOut`

`onLoad`

`onFocus`

`onBlur`

The `onMouseOver` event handler is invoked when the mouse cursor moves into a layer.

The `onMouseOut` event handler is invoked when the mouse cursor moves out of the area of a layer.

The `onLoad` event handler gets invoked when a layer is loaded, that is, the document that ultimately contains the layer is displayed. This is true regardless of whether a layer is visible or not.

The `onFocus` handler is invoked when the layer gets keyboard focus, and the `onBlur` handler is invoked when the layer loses keyboard focus.

Just as in the case of a document, if you want to define the mouse click response for a layer, you must capture `onMouseDown` and `onMouseUp` events at the level of the layer and process them as you want.

If an event occurs in a place where multiple layers overlap, the top-most layer gets the event, even if it is transparent. However, if a layer is hidden, it does not get events.

For an example of capturing events for a layer, see Chapter 13, "Changing Wrapping Width Example."

Using Localized Scripts

You can use the `<SCRIPT>` and `</SCRIPT>` tags within blocks of positioned content. The functions defined in the script will be scoped to the block that contains them, and they cannot be used outside that block.

This functionality is handy, for example, for defining event handlers for a layer.

<LAYER> Tag Syntax

```
<LAYER ID="layer1" BGCOLOR="red"
  onMouseOver='changeColor("blue");'
  onMouseOut='changeColor("red");'>
 <P>Layer content...</P>
 <SCRIPT>
  function changeColor(newcol) {
   bgColor=newcol; // Modifies the layer object's bgColor property
   return false;
   }
  </SCRIPT>
```

```
</LAYER>
```

CSS Syntax

```
<DIV STYLE="position:absolute; layer-background-color:red;
  width:200px; height:100px">
 <P>Layer content...</P>
 <SCRIPT>
 function onMouseOver() {changeColor("blue");}
 function onMouseOut() {changeColor("red");}
 function changeColor(newcol) {
    bgColor=newcol;
    return false;
 }
 </SCRIPT>
</DIV>
```

When the mouse moves into the layer, the layer turns blue. When the mouse moves out of the layer, it turns red. To see the example in action, select:

chgcolor.htm lewin

Animating Positioned Content

You can use JavaScript to modify layers to produce the effects of animation. Frequently, animation revolves around repeating actions over and over again, particularly for looping animations. You can use the JavaScript function `setInterval()` function to repeatedly call a function at a given interval.

For example, the following statement calls the `keepExpanding()` function every 25 milliseconds, with arguments of 20, 30, 40 and 50.

```
setInterval(keepExpanding, 25, 20, 30, 40, 50);
```

JavaScript also provides the `setTimeout()` function, which calls another function after a given amount of time.

The `setTimeOut()` function has two different forms:

```
setTimeout("code to be executed", delay)
setTimeout(function, delay, args...)
```

For example, to invoke `doItAgain("Sam", "piano")` after 3 milliseconds, you can use either of the following statements:

```
setTimeout("doItAgain('Sam', 'piano')", 3)
setTimeout(doItAgain, 3, "Sam", "piano");
```

The `setTimeout()` function is useful for conditionally re-invoking a function, whereas the `setInterval()` function is useful for kicking off the repeated, unconditional invocation of a function.

The following function uses `setTimeout()` to keep making the clipping area of a layer 5 pixels wider and 5 pixels higher until the layer is 450 pixels wide.

```
function expand(layer)
{
    if (layer.clip.right < 450) {
    layer.resizeBy(5, 5);
    setTimeout(expand, 5, layer);
  }
 return false;
}
```

Animating Images

You can achieve many interesting animations by changing the source of an image in conjunction with moving the image. To move an image, you can change the position of the layer that contains the image. To change the source of the image, you can assign a new value to the `src` property of the `image` object.

If the source of the image is changed too quickly or too often, the actual image may not download across the net quickly enough to keep up with the animation. Therefore if you have a script that changes the source of an image in a moving layer, it is best to make sure that the image has fully loaded before you try to do anything with it.

Using OnLoad Handlers

When a document has completely finished loading, it invokes its `onLoad` handler if it has one. You could define an `onLoad` handler for the `BODY` element of a document that initiates any animations in the document. The `onLoad` handler for a `BODY` element may be invoked before all frames in all animated GIF images have finished loading, but it will not be invoked until at least one frame of every animated GIF image has finished loading.

Layers can also have `onLoad` handlers. However, if a layer contains images, the images may load asynchronously from the rest of the layer's content, and the layer may think it has finished loading and thus fire its `onLoad` handler (if it has one) before all its images have finished loading.

Images can have `onLoad` handlers also. However, if the image is an animated GIF, its `onLoad` handler is invoked every time a frame in the image finishes loading. Therefore if your image is an animated GIF, it is better to define an `onLoad` handler that initiates any animations that use that image in the `BODY` element rather than directly on the image.However, it the image is a static GIF or JPEG, by all means define the `onLoad` handler directly on the image.

Chapter 11, "Swimming Fish Example," discusses an example, Positioning and Moving the Fish and Poles, that has a layer containing a fish that swims back and forth. The fish starts swimming when someone clicks on a button. To ensure that nobody can click the button before the fish image has finished loading, the layer containing the button is initially hidden. When the document has finished loading, its `onLoad` handler makes the form layer visible.

Pre-fetching Images

One way to reduce the time required to start an animation is to ensure that the images used in the animation are downloaded to the browser's cache before the animation starts. This approach is known as prefetching the images.

You can prefetch an image by embedding it in a layer. When a layer loads, it loads all its content, including all images, regardless of whether the layer is visible or not. If a page has a hidden layer that contains all the images needed in the animation then when the page opens, the source for the images is downloaded into the browser's cache, even though they are not visible.

Chapter 11, "Swimming Fish Example," discusses an example, Changing the Stacking Order of Fish and Poles, that illustrates the use of a hidden layer to contain images that are not needed when the page opens but are used in the course of animating the contents of the page.

Suppressing the Icon for Images that Have Not Yet Loaded

By default, when a page opens, it shows a placeholder icon for every image in the page that has not finished loading. Animation sequences may sometimes require multiple images. While the images are loading, the user could see lots of placeholder icons that you would prefer they did not see.

A new attribute has been introduced for the `IMG` tag to allow you to suppress the display of placeholder icons.

The `SUPPRESS` attribute for the `IMG` tag can be set to either true or false. The default value is `false`. If `SUPPRESS` is set to `true`, neither the place-holder icon or frame that appear during image loading will be displayed and tool-tips will be disabled for that image.

If `SUPPRESS` is set to `false`, the place-holder icon and frame will always be displayed during loading even if the images are transparent images that would not otherwise be displayed. Tool tips will be active.

Chapter

10

Fancy Flowers Farm Example

This example illustrates how to how to hide and show positioned blocks of content. It uses a pop-up menu to pick which block to display.

This example creates a web page that has five positioned blocks of content. Four of the blocks each contain information about a specific flower, and the fifth block contains a form with a pop-up menu.

The user can choose which flower block to display by using the pop-up menu.

To run the `<LAYER>` version of the example, select:

flower.htm

To run the style sheet version of the example, select:

flowercs.htm lewin

To view the complete code for either version of the example, use the `Page Source` command of the `View` menu in the Navigator browser that is displaying the example.

Introducing the Flower Farm

To start with, the page introduces the flower farm:

```
<HR>
<H1>Welcome to Fancy Flowers Farm </H1>
<HR>
<P>We sell bulbs, seeds, seedlings, and potted plants,
in all shapes, sizes, colors, and varieties.
This page presents information about our most popular varieties.
</P>
```

Creating the Form for Flower Selection

The form is placed in an inflow layer. The form contains a popup menu (a select menu) listing four kinds of flowers. The menu uses an `onClick` event handler, so that when it is clicked, the `changeFlower()` function is invoked to display the selected flower.

The only reason the form needs to be in a layer is so that you can specify the `LEFT` value for it, since it is to be indented from the left edge. Because this is an inflow layer, the natural cursor position in the page will be at the end of the layer when the layer has finished being drawn.

```
<ILAYER ID="formlayer" LEFT=50>
<P>Please select a flower:</P>
<FORM NAME=form1>
 <SELECT name=menu1
 onChange="changeFlower(this.selectedIndex);
 return false;">
 <OPTION >Mona Lisa Tulip
 <OPTION >Mixed Dutch Tulips
 <OPTION >Bijou Violet
```

```
 <OPTION >Pink Chrysanthemum
 </SELECT>
</FORM>
</ILAYER>
```

When the user selects an option in the menu, the `changeFlower()` function is invoked. This function calls the `hideAllFlowers()` function to hide all the flower layers, then shows the flower layer corresponding to the selected option. The flower layers are named `flower0`, `flower1`, `flower2`, and `flower3`. Thus, the name of the selected flower layer is simply the concatenation of `"flower"` and the index of the selected option.

```
<SCRIPT>
// this function hides all the flower layers
function hideAllflowerLayers() {
 document.flower0.visibility="hide";
 document.flower1.visibility="hide";
 document.flower2.visibility="hide";
 document.flower3.visibility="hide";
}
// this function makes a single flower layer visible
function changeFlower(n) {
 hideAllflowerLayers();
 document.layers["flower" + n].visibility="show";
}
</SCRIPT>
```

Positioning the Flower Layers

The page has four layers that contain information about a flower. Each flower layer contains a left-aligned image, a level 3 heading, and some number of paragraphs. The first layer is initially visible, and the remaining flower layers are initially hidden.

All the flower layers are positioned in exactly the same place, and they have the same width and height. The idea is that only one flower layer is visible at a time.

So far, the page does not contain any layers with absolute positions. So you can let the first flower layer fall at the natural cursor position in the page, which is at the end of the inflow layer that contains the form.

If the first flower layer has an absolute position, the natural cursor position in the page will still be at the end of the form layer. Thus you can let each flower layer fall at the natural position in the page, so long as each one has an absolute position.

The following code shows the code for the first flower layer:

```
<LAYER ID="flower0" LEFT=50 width=400
  BGCOLOR="#FFFFDD">
 <HR>
 <H3>Mona Lisa Tulip</H3>
 <HR>
 <IMG src="images/redtul.jpg" align="LEFT" hspace=5>
 <P>These tulips have been specially...</P>
 <BR CLEAR="ALL">
 <P>Priced at only $1 a bulb. . .</P>
</LAYER>
```

The code for the second and third flower layers is very similar. They all use the default value for TOP.

So far, each flower layer has used the default value for TOP. However, if the page had several layers with absolute positions, and you wanted to place another layer in a relative position to one of the existing layers, you could use inline JavaScript to calculate the value for LEFT or TOP. Or if you wanted to make the background of one layer be slightly darker than the background of another, you could use inline JavaScript to calculate the value of the BGCOLOR attribute. (Note however that you can use inline JavaScript only in layer definitions that use the <LAYER> tag. You cannot use inline JavaScript inside layer definitions that use cascading style sheet syntax, although you can use JavaScript to modify such layers after they have been defined and created.)

In this example, there is really no need to use inline JavaScript to position the last flower layer, since you could just let the TOP value default to its natural value, as in the other flower layers.

However, just to provide an illustration of using inline JavaScript, the TOP attribute is given the same value as the TOP attribute for the layer named flower0, as follows: (note that the TOP attribute in the <LAYER> tag can be any case, but the top property in JavaScript must be all lowercase)

```
<LAYER ID="flower3" LEFT=50
 TOP=&{"&"};{document.flower0.top;};
  width=400 VISIBILITY="HIDE"
  BGCOLOR="#DDFFDD">
 <HR>
 <H3>Pink Chrysanthemum</H3>
 <HR>
 <IMG src="images/spikey.jpg" align="LEFT" hspace=5>
 <P>These modern chrysanthemums...</P>
 </LAYER>
```

Chapter

11

Swimming Fish Example

This example is presented in two parts. The second part is an advanced version of the first part.

- Positioning and Moving the Fish and Poles

  This example illustrates how to position and move layers.

  In this example, a fish (an animated GIF) and three poles appear in the window (as shown in Figure 10.1) along with a button saying "Move the Fish." When the user clicks the button, the fish moves repeatedly from the left side of the window to the right, swimming in front of the two outer poles and swimming behind the middle one.

- Changing the Stacking Order of Fish and Poles

  This example illustrates how to change the stacking order of the layers.

  This example extends the previous one, so that when the user clicks the "Move the Fish" button, the fish swims to the right, then changes direction, and swims back to the left, this time swimming behind the outer poles and in front of the middle pole.

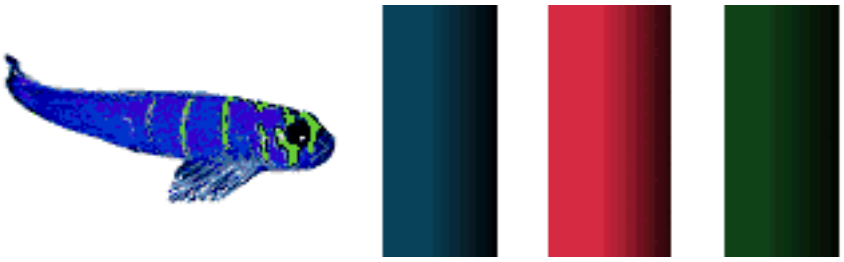

Figure 10.1 The fish and three poles in their initial positions

Positioning and Moving the Fish and Poles

In this example, a fish and 3 poles appear in the window along with a button saying "Move the Fish." When you click the button, the fish moves from the left side of the window to the right, swimming in front of the two outer poles and swimming behind the middle one.When it reaches the far right, it jumps back to the far left and starts swimming across the screen again.

The fish is an animated GIF, and the three poles are static GIFS.

To run the example that uses the `<LAYER>` tag, select:

fish1.htm lewin

To run the style sheet version of the example, select:

fish1css.htm lewin

To view the complete code for either version of the example, use the `Page Source` command of the `View` menu in the Navigator browser that is displaying the example.

In the `<LAYER>` version, the layer containing the form is initially hidden, and a `waiting` layer is temporarily displayed while the fish images are downloading. This version uses a `showForm()` function to hide the waiting layer and show the form layer.

In the style sheet version, the form layer is visible immediately. This version does not have a `waiting` layer or `showForm()` function.

The sections in the first part of this example are:

Defining the onLoad Handler for the BODY Element

This page has a form containing a button whose action is to start the fishing swimming. The form is contained in a layer that is initially hidden. The `BODY` element has an `onLoad` handler that makes the form layer visible. This approach ensures that the user cannot start the fish swimming until the form is visible which will not happen until all the contents in the document, including all the frames in the animated image of the fish, have finished loading.

The following statement defines the `BODY` element:

```
<BODY BGCOLOR="#FFFFFF" ONLOAD="showForm();">
```

Positioning the Fish and Poles

Here's the code that creates the three pole layers:

```
<HTML>
<HEAD>
<TITLE>Swimming Fish</TITLE>
</HEAD>
<BODY>


<LAYER ID="bluepole"LEFT=160 TOP=150>
```

```
<IMG SRC=images/bluepole.gif>
</LAYER>

<LAYER ID="greenpole" LEFT=360 TOP=150>
<IMG SRC=images/greenpol.gif>
</LAYER>
```

Here's the code that creates the fish layer.

```
<LAYER ID="fish" LEFT=40 TOP=170 above="redpole"
 ONLOAD="showForm();">
<IMG ALIGN=RIGHT SRC=images1.gif >
</LAYER>
```

After the definition of the fish layer comes the definition for the red pole layer.

```
<LAYER ID="redpole" LEFT=260 TOP=150>
<IMG SRC=images/redpole.gif>
</LAYER>
```

By default, each subsequent layer is placed on top of the one before it in the stacking order. So to start with, the blue pole is on the "bottom," the green pole is above the blue pole, and the fish is directly below the red pole (that is between the green pole and the red pole.) The red pole is on top of everything, as far as the stacking order goes. (It might help to imagine that all the images are slid into the center of the page so that they all overlap each other. This scenario might help you visualize that the blue pole is on the bottom, and the red pole is on the top.)

Defining the Form

The layer containing the form is initially this layer is hidden. The form has a button that the user clicks to start the fish swimming. The only reason for putting the form in a layer is to hide it initially. Since you don't need to set TOP or LEFT attributes, you can let this be an inflow layer so that it falls at the natural place in the page.

Here's the definition of the form layer:

```
<ILAYER ID=formlayer VISIBILITY=HIDE>
 <H1>Fish Example 1</H1>
 <FORM>
 <INPUT type=button value="Move the fish"
  OnClick="movefish(); return false;">
 </FORM>
</ILAYER>
```

There's also another "temporary" layer that displays a message while the fish is loading. The definition for this layer is:

```
<LAYER ID=waiting TOP=100 LEFT=50>
 <H3>Please wait while the fish loads...</H3>
</LAYER>
```

Moving the Fish

The file contains a script that has the definitions for the `moveFish()` and `showForm()` functions.

The following code defines the function `showForm()`, which makes the waiting layer become invisible and makes the form layer become visible.

```
<SCRIPT>
function showForm() {
 document.waiting.visibility="hide";
 document.formlayer.visibility="show";
 return false;
}
```

The following code defines the function `moveFish()`, which causes the fish to move repeatedly across the window.

```
<!-- Simple move function -->
function movefish() {
 var fish = document.fish;
```

```
 if (fish.left < 400) {
  fish.moveBy(5, 0);}
 else {
  fish.left = 10;}
 // use the windows method setTimeOut
 setTimeout(movefish, 10);
}
</SCRIPT>
```

This function binds the variable `fish` to the layer named `"fish."` The function checks if the horizontal location of the fish layer is less than 400, in which case it uses the `moveBy()` method to move the layer 10 pixels to the right. If the horizontal location is greater than 400, the function sets the horizontal location back to 10.

Then the function waits 10 milliseconds and calls `movefish()` (that is, itself) again.

The net result is that when this function is invoked, the fish swims across the screen to the 400th pixel, then reappears at the left of the screen and swims across the screen again, ad infinitum.

Because of the stacking order of the poles, the fish seems to swim in front of the blue pole, behind the red (middle) pole, and in front of the green pole.

Changing the Stacking Order of Fish and Poles

This example extends the previous example, Positioning and Moving the Fish and Poles.

In this extended version, when the fish reaches the far right, it turns around and swims back again. On the way back, it swims in front of the green pole, behind the red (middle) pole, and in front of the blue pole. To enable the fish to swim in front of a pole on the way out and swim behind it on the way back, you need to change the stacking order of the layers each time the fish changes direction.

Both fishes (one for each direction) are animated GIFs, and the three poles are static GIFs.

To run the <LAYER> version of the example, select:

fish2.htm lewin

To run the style sheet version of the example, select:

fish2css.htm lewin

To view the complete code for either version of the example, use the Page Source command of the View menu in the Navigator browser that is displaying the example.

The sections in the first part of this example are:

- Adding Another Layer to Contain the Reverse Fish Image
- Initializing the Fish to Have a Direction Variable
- Moving the Fish Backward and Forward
- Changing the Direction of the Fish
- Changing the Stacking Order of the Poles and the Fish
- Updating the Button That Gets the Fish Going

Adding Another Layer to Contain the Reverse Fish Image

When the fish reaches the right edge, the image of the fish needs to change to a fish swimming in the reverse direction. The change needs to occur very quickly, perhaps too quickly for there to be time for the new fish image to download across the network. If the image of the reverse fish does not download quickly enough, the image will continue coming in as the fish moves back across the screen. To start with, you'll see only bits of the fish.

To ensure that the fish is whole as soon as it starts swimming back, you can preload the fish image.The easiest way to do this is to create a new, hidden layer that contains the reverse fish image. Even if a layer is hidden, all its images are downloaded when the layer is loaded.

The following code creates a hidden layer containing an image of the fish swimming in the reverse direction.

```
<LAYER ID="fishB" VISIBILITY="hide">
 <IMG SRC=images2.gif>
</LAYER>
```

Initializing the Fish to Have a Direction Variable

The following function initializes the fish layer so that it has a `direction` variable which keeps track of which way the fish is swimming. To start with, the fish swims forward. The fish also has `forwardimg` and `backwardimg` properties that hold the appropriate fish images.

```
function initializeFish() {
 // create the backward fish image to force it to preload now
 var fish = document.fish;
 var fishB = document.fishB;
 fish.direction = "forward";
 fish.forwardimg = fish.document.images["fish"].src;
 fish.backwardimg = fishB.document.images["fishB"].src;
}
```

Moving the Fish Backward and Forward

The following code defines the function `movefish2()`, which moves the fish to the right, changes the image of the fish (so that it faces left), moves the fish back to the left, and repeats the process continuously.

In more detail, the function specifies that if the fish is moving forward and hasn't reached a horizontal position of 450, it keeps moving forward. If it has reached 450, it changes direction.

If it's moving backward and hasn't reached 10, it keeps moving backward. If it has reached 10, it changes direction.

Each time the fish changes direction, the function changes the stacking order of the layers, by calling either the `changePoles()` function or the `resetPoles()` function, depending on which way the fish is turning.

```
function movefish2() {
 var fish = document.fish;
 if (fish.direction == "forward") {
  if (fish.left < 450) {fish.moveBy(5, 0);}
  else {changePoles();changeDirection();}
 }
 else {
  if (fish.left > 10) {fish.moveBy(-5, 0);}
  else {resetPoles();changeDirection();}
 }
 setTimeout("movefish2()", 10);
 return;
}
```

Changing the Direction of the Fish

The `changeDirection()` function changes the image of the fish, so that it faces in the correct direction. The function also sets the value of the `direction` variable to the new direction.

```
function changeDirection () {
 var fish = document.fish;
 if (fish.direction == "forward") {
  fish.direction = "backward";
  fish.document.images["fish"].src = fish.backwardimg;
```

```
 }
 else {fish.direction = "forward";
  fish.document.images["fish"].src = fish.forwardimg;
 }
 return;
}
```

Changing the Stacking Order of the Poles and the Fish

The functions `changePoles()` and `resetPoles()` change the stacking order (z-order) of the layers. You can change the stacking order of a layer in the following ways:

- Use the `moveBelow()` layer to move a layer immediately below another one.
- Use the `moveAbove()` layer to move a layer immediately above another one.
- Directly set the value of the `zIndex` property of a layer.

To keep your stacking order straight, it is a good idea to consistently use one of these ways. If you mix them, it could be hard to keep track of the exact stacking order. For example, if you use `moveAbove()` to move the blue pole layer above the green pole layer, then you set the `zIndex` value of the fish layer to 3, you may not know where the fish is in the stacking order in relation to the green and blue poles.

The following functions, `changePoles()` and `resetPoles()`, consistently use the `moveAbove()` function to set the stacking order of the three layers containing the poles and the layer containing the fish.

```
function changePoles () {
 var redpole = document.redpole;
 var bluepole = document.bluepole;
 var greenpole = document.greenpole;
 var fish = document.fish;
```

```
 fish.moveAbove(redpole);
 bluepole.moveAbove(fish);
 greenpole.moveAbove(bluepole);
}

// reset the stacking order of the poles and the fish
function resetPoles () {
 var redpole = document.redpole;
 var bluepole = document.bluepole;
 var greenpole = document.greenpole;
 var fish = document.fish;
 greenpole.moveAbove(bluepole);
 fish.moveAbove(greenpole);
 redpole.moveAbove(fish);
}
```

Updating the Button That Gets the Fish Going

Here is the definition of the layer that contains the form:

```
<H1>Fish Example 2</H1>
<LAYER ID="fishlink" LEFT=10 TOP=100 >
 <FORM>
  <INPUT type=button value="Move the Fish"
  OnClick="initializeFish(); movefish2(); return false;">
 </FORM>
</LAYER>
```

This time, the `OnClick()` method initializes the fish to initialize the `direction` variable on the fish before it calls `movefish2()`.

Chapter

11

Nikki's Diner Example

This example illustrates the use of external files as the source for a layer. This example creates a web page for Nikki's Diner, which is a vegan restaurant that offers tasty daily specials. The web page contains some general information about the diner, and then offers a pop-up menu that lists the days of the week. When a user selects a particular day, the specials for that day are displayed.

To run the `<LAYER>` version of the example see:

diner.htm lewin

To run the style sheet version of the example see:

dinercss.htm lewin

To view the complete code for either version of the example, use the `Page Source` command of the `View` menu in the Navigator browser that is displaying the example.

The functions used in both versions are identical.

To view the files containing the daily specials see:

specials/mon.htm

specials/tues.htm

specials/wed.htm

specials/thurs.htm

specials/fri.htm

specials/sat.htm

specials/sun.htm

The specials for each day are written in separate files. There is a file for Monday's special, (`mon.htm`) another for Tuesday's special (`tues.htm`) and so on. These files contain HTML formatted text that describes the specials for that day.

The benefit of this system is that changing the specials for a particular day of the week is a trivial process. For example, to update the specials offered on Monday, Nikki simply has to change the text in the `mon.htm` file. She doesn't have to make any changes to the main file for the web page document.

Content in the External Files

The following code shows the entire contents of the file `mon.htm`:

```
<HR>
<H1 align=center > Monday</H1>
<HR>
<H2 align=center >Entrees</H2>
<P>Tofu, Artichoke, and Asparagus Surprise</P>
<P>Walnut and Carrot Risotto</P >
<P>Parsnip Casserole </P >
<P>Chef's Special Spicy Salad</P >
<H2 align=center >Desserts</H2>
<P>Gooseberry Tart</P >
<P>Strawberry Delight</P >
```

The content of the files `tues.htm`, `wed.htm`, and so on are similar.

The File for the Main Page

The file for Nikki's Diner's home page starts with some general information about the diner. Paragraphs in the general introduction are not indented, and the paragraphs in the layers are indented. This page uses style sheets to achieve this indentation effect.

```
<HTML>

<HEAD>

<TITLE>Welcome to Nikki's Diner</TITLE>

<STYLE TYPE="text/css">

<!--

 P {margin-left:50;}

 P.plainPara {margin-left:0};

-->

</STYLE></HEAD>

<BODY BGCOLOR="white">

<HR>

<H1 align = "center">Welcome to Nikki's Diner!</H1>

<HR>

<P CLASS=plainPara>Nikki's Diner is the best place for vegan food in
NetscapeVille. </P>

<P CLASS=plainPara>You can find us at the corner of Communicator Street
and Navigator Way. We're open from 10 am to 6 pm every day. We don't
take reservations, so just come on down. We guarantee that after you
visit us once, you'll be back on a regular basis!</P>

<P CLASS=plainPara>We have an extensive regular menu of tasty meals in
addition to our daily specials.</P>

<P CLASS=plainPara >You can use the following menu (no pun intended) to
view the Specials for any day this week. Our specials change every
week.</P>
```

Next comes an inflow layer containing a form that lets users pick a day of the week. This layer is indented 50 pixels to the left. Because it is an inflow layer, the natural position in the page will be at the end of the layer, when the layer has finished being drawn.

```
<LAYER ID="formlayer" LEFT=50>
<P Please select a day of the week:</P>
  <FORM NAME=form1>
   <SELECT name=menu1 onChange="showSpecials(this.selectedIndex); return
false;">
    <OPTION >Saturday
    <OPTION >Sunday
    <OPTION >Monday
    <OPTION >Tuesday
    <OPTION >Wednesday
    <OPTION >Thursday
    <OPTION >Friday
   </SELECT>
  </FORM>
</LAYER>
```

The next task is to create the layer where the daily specials will be shown.

The menu layer needs to have an absolute position, since changing the source on the fly works only for layers with absolute positions.

Since this is the first layer with an absolute position in the document, its position defaults to the current cursor position in the page, which happens to be beneath the inflow form layer.

You want the top value to default to the natural top position, so do not supply a value for `TOP`, but you want the left value to be 50 pixels in from the left edge. By default, Saturday's menu appears.

```
<LAYER ID="menu" LEFT=50 WIDTH=400 src="specials/sat.htm">
</LAYER>
```

The script is defined at the level of the document rather than inside a particular layer, since it involves both the form and the menu layer. The `showSpecial()` function assigns a source for the menu layer depending on which menu option was picked.

```
<SCRIPT>
function showSpecials(n) {
```

```
 var specials = document.menu;
 switch (n) {
  case 0: specials.src = "specials/sat.htm"; break;
  case 1: specials.src = "specials/sun.htm"; break;
  case 2: specials.src = "specials/mon.htm"; break;
  case 3: specials.src = "specials/tues.htm"; break;
  case 4: specials.src = "specials/wed.htm"; break;
  case 5: specials.src = "specials/thurs.htm"; break;
  default: specials.src = "specials/fri.htm";
  }
}
</SCRIPT>
</BODY>
</HTML>
```

Chapter

12

Expanding Colored Squares Example

This example illustrates how to expand and contract the clipping region of a layer, without changing the wrapping width of the layer. (The next example, Chapter 13, "Changing Wrapping Width Example,", illustrates how to capture mouse events so that the user can make a layer's wrapping width wider or narrower by dragging the mouse.)

This example illustrate these tasks:

- using of `onLoad` and `onMouseOver` event handlers for layers
- dynamically changing clipping regions of layers
- writing to layers
- changing the source of a layer
- using nested layers

The sections in this chapter are:

Running the Example

In this example, when the page initially loads, the user sees four colored squares as follows:

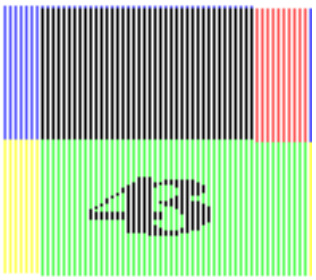

Figure 12.1 Initial appearance of the four colored squares

When the square is fully contracted, it displays a number. If the user moves the mouse over one of the squares, its content changes to a block of text and the square expands.The top-left square expands up and to the left, the top-right square expands to the top and to the right, and so on.

The following figure shows the four squares after the top-left and top-right squares are fully expanded.

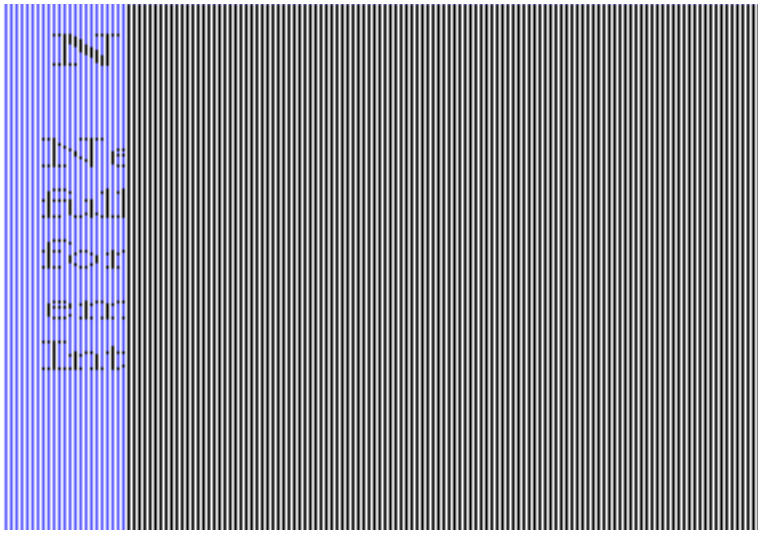

Figure 12.2 Two squares are fully expanded

While a square is expanding, further mouse-over events are blocked on that square until it has finished expanding. When it is fully expanded, if the user moves the mouse over it, then it contracts again. While it is contracting, all mouse-over events for that square are blocked until it has finished contracting. When it finishes contracting, it changes its content back to a number.

To run the example see:

squares.htm lewin

This example is provided only as a <LAYER> version.

To view the complete code for the example, use the Page Source command of the View menu in the Navigator browser that is displaying the example.

Creating the Colored Squares

Each colored square is in its own layer. The width of each layer is 200 and the height is 200. When the page loads you only see a 50x50 region of each layer because as soon as it is loaded, it calls a function that sets its clipping region so that only a small part of the square is visible.

Each square layer contains another layer that displays a number. The number needs to be in a layer so that can be placed it in the portion of the layer that is visible when the square is fully contracted.

The following figure shows where the number 1 would appear in the top-left square if it were not in a layer but were allowed to fall in its natural position in the parent layer. As you can see, when the red square is fully contracted, the number would not be visible.

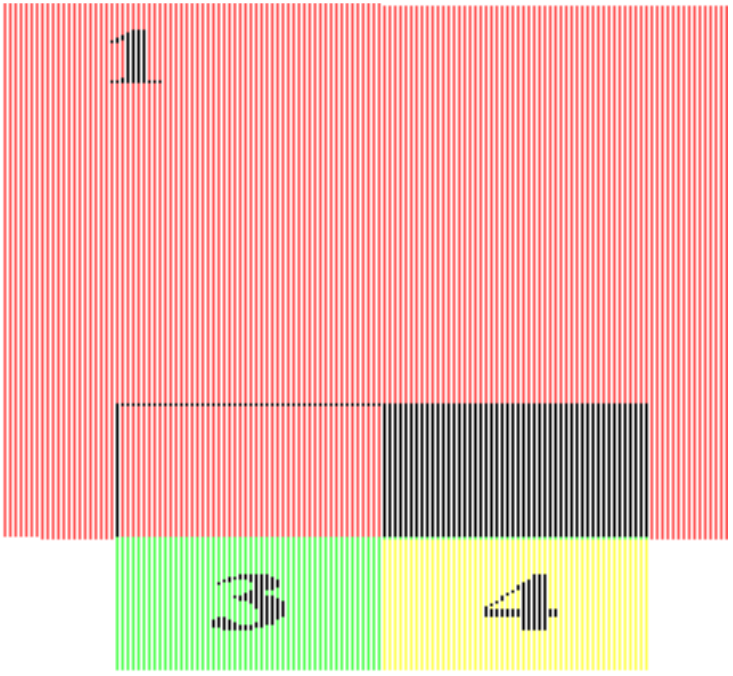

Figure 12.3 Position where the number 1 would appear if it were not in a layer

Definitions for the Layers

The following code defines the top-left layer:

```
<LAYER ID="topleftblock" top=50 left=50
 width=200 height=200
 BGcolor="#FF5555"
 onLoad = initializeTopLeft(0);
 onMouseOver=changeNow(0); >
 <LAYER TOP=160 LEFT=168>
    <H1>1</H1>
  </LAYER>
</LAYER>
```

This layer would be 200 pixels wide by 200 high. However, when this layer finishes loading, it calls its `onLoad` function, `initializeTopLeft()`.

Before considering the `initializeTopLeft()` function, quickly look at the global variables defined in the script. There are four variables that describe the minimum and maximum clipping values. The variable `delta` specifies the distance by which the clipping values change each time the `expand()` or `contract()` functions are called. (These functions will be discussed in detail soon.)

```
<SCRIPT>
var maxclip = 200;
var minclip = 0;
var maxclipcontracted = 150;
var minclipcontracted = 50;
var delta = 10;
```

The Initialization Functions

The `initializeTopLeft()` function does the following things:

- Sets the layer's status variable to `"waitingToExpand"`. (Notice that you can create the variable simply by using it.)
- Sets the `clip.top`, `clip.bottom`, `clip.right`, and `clip.left` values so that the visible region of the layer is a square measuring 50 pixels by 50 pixels in the bottom right corner, as illustrated in the following figure:

Figure 12.4 Clip values

- Sets the `dleft`, `dtop`, `dbottom`, and `dright` variables to indicate by how much the `clip.left`, `clip.top`, `clip.bottom` and `clip.right` variables need to change while the square is expanding.
- Sets the `myposition` variable to `topLeft`.
- Sets the `mysource` variable so that it specifies the source file to be used as the contents of the layer when it starts expanding.
- Sets the `mytext` variable so it contains the text that will be written to the layer when the layer is fully contracted.

The full definition for the `initializeTopLeft()` function is shown here:

```
function initializeTopLeft(n)
{
 var thislayer = document.layers[n];
 thislayer.status = "waitingToExpand";
 thislayer.clip.top = maxclipcontracted;
 thislayer.clip.left = maxclipcontracted;
 thislayer.clip.bottom = maxclip;
 thislayer.clip.right = maxclip;
 thislayer.dleft = -delta;
 thislayer.dtop = -delta;
 thislayer.dbottom = 0;
 thislayer.dright = 0;
 thislayer.myposition = "topLeft";
 thislayer.mysource="point1.htm"
 thislayer.mytext="<LAYER TOP=160 LEFT=168><H1>1</H1></LAYER>"
 return false;
}
```

Each of the other three layers has a similar definition, and a corresponding initialization function.

The Last Layer

Since the bottom squares can be dynamically expanded beyond the height of the page, add a last layer that is positioned below the bottom of the expanded bottom squares. This last layer has nothing in it, but it forces the Web page to increase its height to be big enough to include the expanded layers. Thus you will be able to use the scrollbar to scroll down to the bottom of the expanded layers if they do not initially fit on your screen.

If you we do not include this last layer, then the scrollbar will only allow you to scroll to the bottom of the contracted squares.

Here is the definition for the last layer:

```
<LAYER TOP=500>
<P></P>
</LAYER>
```

Moving the Mouse Over a Square

When you move the mouse over any of the colored squares, its `changeNow()` function is invoked. (This is because in the layer definition, the `onMouseOver` handler is set to `changeNow`().)

The basic aim of the `changeNow()` function is to start expanding the layer if it is fully contracted, or start contracting the layer if it is fully expanded. If the layer is already in the process of expanding or contracting, it ignores the new mouse over event.

The `status` variable indicates whether the layer is waiting to expand, waiting to contract, expanding or contracting. The status value for each layer is initialized to "`waitingToExpand`".

The `changeNow()` function simply checks the status of the layer and then calls the `expand()` function, the `contract()` function, or does nothing, depending on the layer's status. If the layer needs to start expanding, it first sets the layer's source to change the content of the layer to show text instead of just a number.

```
function changeNow (n)
{
 var thislayer = document.layers[n];
 if (thislayer.status == "waitingToExpand")
 {
  thislayer.src=thislayer.mysource;
  expand(n);
  }
 else if (thislayer.status == "waitingToContract")
  {contract(n);}
 return false;
}
```

The expand() Function

The `expand()` function sets the layer's status to `expanding`. Then it changes each of the `clip.left`, `clip.right`, `clip.top`, and `clip.bottom` variables by the values appropriate to the particular layer to increase the layer's visible region in the appropriate direction. It then checks if the layer is fully expanded and, if not, calls the `setTimeout()` function to reinvoke the `expand()` function.

If the layer has finished expanding, the `expand()` function sets the layer's status to `"waitingToContract"`.

Here is the code for the `expand()` function:

```
function expand (n)
{
 var thislayer = document.layers[n];
 thislayer.status = "expanding";
 // increase or decrease each clip value as appropriate
 thislayer.clip.left=thislayer.clip.left+thislayer.dleft;
 thislayer.clip.right=thislayer.clip.right+thislayer.dright;
 thislayer.clip.bottom=thislayer.clip.bottom+thislayer.dbottom;
 thislayer.clip.top=thislayer.clip.top+thislayer.dtop;
 // is the layer fully expanded?
 if (
  (((thislayer.myposition == "topLeft") |
   (thislayer.myposition == "bottomLeft")) &&
   (thislayer.clip.left > minclip)) ||
  (((thislayer.myposition == "topRight") |
   (thislayer.myposition == "bottomRight")) &&
   (thislayer.clip.right < maxclip)))
 // if not, call expand again
 {setTimeout("expand(" + n + ")", 50);}
 // if so, change the layer's status
 else {thislayer.status = "waitingToContract";}
 return false;
}
```

The contract() Function

The `contract()` function is very similar to the `expand()` function. The `contract()` function sets the layer's status to `contracting`. Then it changes each of the `clip.left`, `clip.right`, `clip.top`, and `clip.bottom` by the values appropriate to the particular layer to decrease the visible region in the appropriate direction. It then checks if the layer is fully contracted, and if not, calls the `setTimeout()` function to reinvoke the `contract()` function.

If the layer has finished contracting, the `contract()` function sets the layer's status to `"waitingToExpand"`. The other thing it does is to change the contents of the layer so that you can see the number of the layer. It does this by opening the layer's document, writing the data stored in the layer's `mytext` variable, and then closing the layer's document.

The value of the `mytext` variable was set during the initialization process. For each layer, it contains the HTML text for an embedded layer that displays the layer's number at a place that will be visible when the layer is fully contracted.

Here is the code for the `contract()` function:

```
function contract (n)
{
 var thislayer = document.layers[n];
 thislayer.status = "contracting";
 // increase or decrease each clip value as appropriate
 thislayer.clip.left=thislayer.clip.left-thislayer.dleft;
 thislayer.clip.right=thislayer.clip.right-thislayer.dright;
 thislayer.clip.bottom=thislayer.clip.bottom-thislayer.dbottom;
 thislayer.clip.top=thislayer.clip.top-thislayer.dtop;
 // is the layer fully contracted? True if
 // the square is the top OR bottom left AND its clip left
 //  is less than or equal to the minimum clip for contracted squares
 // OR if the square is the top OR bottom right AND its clip right
 // is greater than or equal the max clip for contracted squares
```

```
 if (
  (((thislayer.myposition == "topLeft") |
   (thislayer.myposition == "bottomLeft")) &&
   (thislayer.clip.left <= minclipcontracted)) ||
  (((thislayer.myposition == "topRight") |
   (thislayer.myposition == "bottomRight")) &&
   (thislayer.clip.right >= maxclipcontracted)))
 // if not, call contract again
 {setTimeout("contract(" + n + ")", 50);}
 // if it is fully contracted
 else {
  // change the status
  thislayer.status = "waitingToExpand";
  //open the document, write mytext to it, close again
  thislayer.document.write(thislayer.mytext);
  thislayer.document.close();
 }
 return false;
}
</SCRIPT>
```

Styles in the Document

Just to make the text in the squares look prettier, this file uses a style sheet to set left and right margins for paragraphs, and to center level-three headings:

```
<STYLE TYPE="text/css">
<!--
 P {margin-left:10%; margin-right:10%;}
 H3 {text-align:center; margin-top:4%;}
-->
</STYLE>
```

Chapter

13

Changing Wrapping Width Example

The previous example, Chapter 12, "Expanding Colored Squares Example," illustrates how to expand and contract the clipping region of a layer without changing the wrapping width of the layer.

This example illustrates how to capture mouse events so that the user can make a layer's wrapping width wider or narrower by dragging the mouse.

This example illustrates:

- how to capture mouse events for a layer
- how to change the wrapping width of a layer by using the `load()` function

The sections in this chapter are:

Running The Example

When the page loads, you'll see a a blue layer containing a block of text. You can change the wrapping width of the layer by moving the mouse into the layer, pressing the mouse button down, and moving the mouse to the left or right. The wrapping width of the layer increases when you move the mouse to the right, and decreases when you move the mouse to the left. When you release the mouse button, the layer stops tracking mouse events and no longer changes in accordance with the mouse.

To run the <LAYER> version of the example see:

wrapping.htm lewin

For the style sheet version of this example see:

wrapcss.htm lewin

Defining the Block of Content

The definition for the block of content is very simple. It sets the left position, sets the background color, sets the initial wrapping width, and specifies the source for the layer:

```
<LAYER NAME="layer1" LEFT=100
 WIDTH=300 BGCOLOR="#99bbFF"
 SRC="mytext.htm" >
</LAYER>
</BODY>
```

Capturing Events for the Layer

The first thing the script does is to define some variables that it needs. These include `layerWidth`, which is the initial width of the layer; `oldX` which keeps track of the previous x position of the mouse when it is dragged inside the layer; and `layer1`, which is the layer itself.

```
var layerWidth = 300;
var oldX;
var layer1 = document.layer1;
```

Next, the script specifies which events layer1 needs to capture:

```
layer1.document.captureEvents(
 Event.MOUSEUP|Event.MOUSEDOWN|Event.MOUSEDRAG);
```

Then it specifies that when the mouse is pressed down inside `layer1`, the `begindrag()` function is called, and when the mouse button is released (let up) inside layer1, the `enddrag()` function is called. (These functions will be defined shortly.)

```
layer1.document.onmousedown=begindrag;
layer1.document.onmouseup=enddrag;
```

The script specifies that after `layer1` has loaded, the `resetcapture()` function is invoked.

```
layer1.onLoad=resetcapture;
```

Next comes the definition of the `resetcapture()` function, which basically restates which events the layer needs to capture:

```
function resetcapture() {
 layer1.document.captureEvents(
 Event.MOUSEUP|Event.MOUSEDOWN|Event.MOUSEDRAG|Event.MOUSEMOVE);
}
```

Defining the Dragging Functions

When you press the mouse down in the layer, the layer's `onMouseDown` event handler is called, which in this case is the `begindrag()` function. The `begindrag()` function sets the layer's `onMouseMove` handler to `drag`, so that when you move the mouse while the button is pressed down, the `drag()` function is invoked. When you release the mouse button, the layer's `onMouseUp` event handler is invoked, which in this case is the `enddrag()` function.

When an event occurs, an event object is created to represent the event. This event object has a `PageX` variable, which indicates the x position in the page where the event occurred.

The begindrag() Function

The `begindrag()` function tells the layer that it needs to capture mouse-move events. It sets the `onmousemove` handler to `drag` so that the `drag()` function will be invoked when the mouse is moved. Then it gets the x position of the mouse-down event and stores it in the `oldX` global variable.

```
function begindrag(e) {
 layer1.document.captureEvents(Event.MOUSEMOVE);
 layer1.document.onmousemove=drag;
 oldX=e.pageX;
 return false;
}
```

The drag() Function

The `drag()` function calls the `changeWidth()` function, which changes the wrapping width of `layer1` by the distance that the mouse moved since the drag function was last called, or if applicable since the `begindrag()` function

was called. This distance is calculated by subtracting the x value of the previous event (stored in `oldX`) from the `pageX` value of the current event. Finally the `drag()` function updates the value stored in `oldX`.

```
function drag(e) {
 changeWidth(layer1, e.pageX - oldX);
 oldX = e.pageX;
 return false;
}
```

The only way to change the wrapping width of a layer is to reload the contents of the layer using the `load()` function. This function takes two arguments: the file to use as the content of the layer, and the new wrapping width of the layer.

The `changeWidth()` function increases the value of the `layerWidth` global variable by the amount that the mouse moved. If the distance that the mouse moved is not zero, the function calls the `load()` method on the layer to load the file `"mytext.htm"` and also to change the layer's wrapping width to the new layer width. Since the same file is loaded over and over, in effect the content does not seem to change, but the wrapping width constantly changes so that the content wraps neatly at the right edge of the layer.

```
function changeWidth(layer, delta)
{
 layerWidth = layerWidth + delta;
 if (delta != 0)
 layer.load("mytext.htm", layerWidth);
}
```

When you use `load()` to change the wrapping width, the value of `clip.right` automatically changes to show the full wrapping width, so long as you have not changed the value of `clip.right` from its default initial value. If you have specifically set the value of `clip.right`, then the right edge of the clipping region will not change, even if the wrapping width changes.

The enddrag() Function

When you release the mouse, the `enddrag()` function is called. The only thing this function does is set the layer's `onMouseMove` handler to 0, and release the mouse-move event. If the mouse-move event was not released, the layer would continue tracking all mouse move events.

```
function enddrag(e) {
 layer1.document.onmousemove=0;
 layer1.document.releaseEvents(Event.MOUSEMOVE);
 return false;
}
```

Part 3.Downloadable Fonts

Contents

Chapter

14

Using Downloadable Fonts

Font enhancements in Communicator include the ability to incorporate downloadable fonts into your web documents. By using downloadable fonts on your web pages, you can specify whatever fonts you want to enhance the appearance of your pages.

The fonts are contained in a font definition file that reside on the host web server with the HTML document. When the page is accessed by a browser, the font definition file is downloaded with the HTML file in the same way that a GIF or JPEG file would be. The font definition file is loaded asynchronously so that the HTML page doesn't have to wait while the fonts are loading.

The dowloaded font remains on the end user's system only while the page is in the browser's cache. End users cannot copy the fonts for their own future use.

This document contains the following sections:

Creating and Using Font Definition Files

Before you can create font definition files, make sure the fonts you wish to use in your web document are installed on your system. You can get fonts by creating them, purchasing them, or finding free fonts on the Internet. Be aware that fonts are subject to copyright laws, so be sure you have the right to use a font before you incorporate it as a downloadable font in your web documents.

As a first place to look for fonts to buy or download free, you can search the web using keywords such as "font buy"or "font free."

Creating Font Definition Files

When the desired fonts are installed on your system, the next step is to make a font definition file. To do this, you need a font definition file authoring tool, such as Typograph from HexMac, or the Font Composer Plugin for Communicator.

To download a font definition generation tool from HexMac, go to their web site at:

http://www.hexmac.com/ fontswin

The exact steps for creating a font definition file depend on the tool you are using. For example, in HexMac Typograph, you would open your document in Typograph, and use simple menus to select fonts and apply them to text. You then *burn* the file, which saves the document, creates a font definition file that contains the fonts used by the file, and also links the font defintion file into the document.

When creating a font definition file, you must specify the domain that is allowed to use these fonts. That is, only web pages served by the specified domain are allowed to download the font file. For example, for fonts to be downloaded with this document, which is served from `developer.netscape.com`, the domain for the font file is:

```
//developer.netscape.com
```

Linking Font Definition Files Into a Document

After you have created a font definition file, you can link it directly into documents either by using a style sheet or by using the <LINK> tag.

The following example links a font definition file using CSS syntax.

```
<STYLE TYPE="text/css"><!--
    @fontdef url(http://home.netscape.com/fonts/sample.pfr);
--></STYLE>
```

You can link a font definition file into a document by using a LINK tag whose REL attribute is FONTDEF, and whose SRC attribute is the pathname to the font definition file, as shown here:

```
<LINK REL=FONTDEF SRC="http://home.netscape.com/fonts/sample.pfr">
```

The source URL can be any valid URL.

Using Fonts in the Document

After linking a font definition file into a document, you can use the fonts that are contained in the font definition file anywhere in the document. You can either use the fonts as the value of the FACE attribute in the <FONT> tag, or you can use them as the value of the font family style sheet property, as discussed in the section "Font Family" in Chapter 5, "Style Sheet Reference."

The following code creates a style sheet that contains a style definition for all <H1> tags. All <H1> elements will be displayed in the Impress BT font. If that font is not available (for example, the font definition file cannot be located), the element uses the Helvetica font. If that font is not available, the generic sans serif font is used as a last resort.

```
<STYLE type="text/css">
<!--
H1 {font-family:"Impress BT", "Helvetica", sans-serif;}
-->
</STYLE>
```

The following example displays an <H1> elemet in the Impress BT font.

```
<H1> <FONT FACE="Impress BT">This H1 Uses Impress BT Font </FONT></H1>
```

For a further example of the use of downloadable fonts, open the following page:

fontdef1.htm fontswin

You can view the source code for the file `fontdef1.htm` to see how the fonts are used in the file.

Adding a New MIME Type to the Web Server

When you are ready to make your document available on the web, you need to put the font definition file in the place where the document expects to find it. The font definition file will be downloaded with documents that use it, so long as it is served from the domain for which the font definition file was created.

You will also need to add a new MIME type to your web server if it has not already been added.

Add the MIME type `application/font-tdpfr`, and specify its ending as `.pfr`.

Web servers cananot download font definition files unless they know about this MIME type.

New Attributes for the FONT Tag

The `<FONT>` tag takes new `POINT-SIZE` and `WEIGHT` attributes, in addition to the other attributes it already supports.

POINT_SIZE Attribute

The `POINT-SIZE` attribute indicates the point size of the font. For example:

```
<P><FONT FACE="BT Impress" POINT-SIZE=18>
This text appears in 18 pt monspace font.</FONT>
```

```
</P>
```

The `POINT_SIZE` attribute lets you set exact point sizes. (The existing `SIZE` attribute lets you set the font size relative to the existing size, for example, "+2" or "-2".)

WEIGHT Attribute

The `WEIGHT` attribute indicates the weight, or "boldness" of the font. The value is from 100 to 900 inclusive (in steps of 100), where 100 indicates the least bold value, and 900 indicates the boldest value.

If you use the `<B>` tag to indicate a bold weight, the maximum boldness is always used. The `WEIGHT` attribute allows you to specify degrees of boldness, rather than just "bold" or "not bold,"

For example:

```
<P>

<FONT FACE="MONOSPACE" POINT_SIZE=18 WEIGHT=600>

This text appears in 18 pt monospace font.It is fairly bold, but it
could be even bolder if it needed to be.</FONT>

</P>
```

Further Information

For more information about dynamic fonts, see:

http://home.netscape.com/comprod/products/communicator/fonts/index.html

Another information resource is:

http://www.bitstream.com/world/ fontswin

The following link takes you to a very informative article that contains information and recommendations about buying fonts:

http://www4.zdnet.com/macuser/mu_0696/desktop/desktop.html fontswin

The following link takes you to a paper published by the World Wide Web Consortium (W3C) discussing fonts and the web.

http://www.w3.org/pub/WWW/Fonts/ fontswin

Index

Symbols

A

B

C

D

E

F

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z