chapter 9

CSS modification

JAVASCRIPT ALLOWS YOU to modify the CSS presentation layer of your site. A style change is an excellent (and very common) way to draw your users' eyes to the page element on which you want them to focus.

In fact, seven of the example scripts use some form of CSS modification. For instance, Form Validation changes the styles of the incorrect form field, and XMLHTTP Speed Meter uses animations (i.e., many changes to the same style in a short time) to draw the eye of the user to the speed data. (Also, to be honest, as a bit of eye candy.) Dropdown Menu uses style changes to show and hide menu items. These changes all have the same purpose: to draw the eye of the user to these elements.

JavaScript can modify CSS in four ways:

- By modifying the `style` property of an element (`element.style.margin = '10%'`).
- By changing the class or id of an element (`element.className = 'error'`). The browser automatically applies the styles defined on the new class or id.
- By writing new CSS instructions in the document (`document.write('<style>.accessibility {display: none}</style>')`).
- By changing the style sheet of the entire page.

Most CSS modification scripts use either the `style` property or a class/id change. The `document.write` method is used only in specialized situations to enhance the page's accessibility. Finally, changing entire style sheets is rarely useful, both because it's not supported by all browsers, and because usually you want to single out specific elements for style changes.

However, I use all four methods in the example scripts. We'll look at all of them, and their proper context, in this chapter.

A: The style property

The first, and best-known, way of modifying CSS is through the `style` property that all HTML elements possess, and that accesses their inline styles. `style` contains one property for every inline CSS declaration. If you want to set an element's CSS `margin`, you use `element.style.margin`. If you want to set its CSS `color`, you use `element.style.color`. The JavaScript property always has a name similar to the CSS property.

> **INLINE STYLES!**
>
> Remember: the style property of an HTML element gives access to the inline styles of that element.

Let's review a bit of CSS theory. CSS offers four ways to define styles on an element. You can use inline styles, where you put your CSS directly in the `style` attribute of the HTML tag:

```
<p style="margin: 10%">Text</p>
```

In addition, you can embed, link to, or import a style sheet. However, since an inline style is more specific than any other kind of style, inline styles overrule styles defined in an embedded, linked, or imported style sheet. Because the `style` property gives access to these inline styles, it will always overrule all other styles. That's the great strength of this method.

However, when you try to read out styles, you might encounter problems. Take this example:

```
<p id="test">Text</p>
p#test {
    margin: 10%;
}
alert(document.getElementById('test').style.margin);
```

The test paragraph doesn't have any inline styles. Instead, the `margin: 10%` is defined in an embedded (or linked or imported) style sheet, and is therefore impossible to read through the `style` property. The alert remains empty.

In the next example, the alert will return a result of "10%", because `margin` has now been defined as an inline style:

```
<p style="margin: 10%" id="test">Text</p>
alert(document.getElementById('test').style.margin);
```

Thus, the `style` property is excellently suited for setting styles, but less useful for getting them. Later we'll discuss ways to get styles from an embedded, linked, or imported style sheet.

Dashes

Many CSS property names contain a dash, for instance `font-size`. In JavaScript, however, a dash means "minus," and therefore it cannot be used in a property name. This gives an error:

```
element.style.font-size = '120%';
```

Does the browser have to subtract the (undefined) variable `size` from `element.style.font`? If so, what does the `= '120%'` bit mean? Instead, the browser expects a camelCase property name:

```
element.style.fontSize = '120%';
```

The general rule is that all dashes are removed from the CSS property names, and that the character after a dash becomes uppercase. Thus `margin-left` becomes `marginLeft`, `text-decoration` becomes `textDecoration`, and `border-left-style` becomes `borderLeftStyle`.

Units

Many numerical values in JavaScript need a unit, just as they do in CSS. What does `fontSize=120` mean? 120 pixels? 120 points? 120 percent? The browser doesn't know, and therefore it doesn't do anything. The unit is necessary to clarify your intent.

Take the `setWidth()` function, which forms the core of the animations in XMLHTTP Speed Meter:

[XMLHTTP Speed Meter, lines 70-73]

```
function setWidth(width) {
    if (width < 0) width = 0;
    document.getElementById('meter').style.width = width + 'px';
}
```

The function is handed a value, and it should change the width of the meter to this new value. After a safety check that allows only 0 or larger numbers, it sets

DON'T FORGET YOUR 'PX'

Forgetting to append the `'px'` unit to a width or height is a common CSS modification error.

In CSS quirks mode, adding `'px'` is not necessary, since the browsers obey the old rule that a unitless number is a pixel value. In itself this is not a problem, but many Web developers have acquired the habit of leaving out units when they change widths or heights, and encounter problems when they work in CSS strict mode.

the `style.width` of the element to the new width. Then it adds `+ 'px'`, because without that, the browsers wouldn't know how to interpret the number, and would do nothing.

Getting styles

WARNING Browser incompatibilities ahead

As we saw, the `style` property cannot read out styles set in embedded, linked, or imported style sheets. Because Web developers occasionally need to read out these styles nonetheless, both Microsoft and W3C have created ways of accessing non-inline styles. The Microsoft solution works only in Explorer, while the W3C standard works in Mozilla and Opera.

Microsoft's solution is the `currentStyle` property, which works exactly like the `style` property, except for two things:

- It has access to all styles, not just the inline ones, and therefore reports the style that is actually applied to the element.
- It is read-only; you cannot set it.

For instance:

```
var x = document.getElementById('test');
alert(x.currentStyle.color);
```

Now the alert shows the current `color` style of the element, regardless of where it's defined.

W3C's solution is the `window.getComputedStyle()` method, which works similarly but with a more complicated syntax:

```
var x = document.getElementById('test');
alert(window.getComputedStyle(x,null).color);
```

`getComputedStyle()` always returns a pixel value, even if the original style was, for instance, `50em` or `11%`.

As always when we encounter incompatibilities, we need a bit of code branching to satisfy all browsers:

```
function getRealStyle(id,styleName) {
    var element = document.getElementById(id);
    var realStyle = null;
    if (element.currentStyle)
            realStyle = element.currentStyle[styleName];
    else if (window.getComputedStyle)
            realStyle = window.➥
            getComputedStyle(element,null)[styleName];
    return realStyle;
}
```

You use this function as follows:

```
var textDecStyle = getRealStyle('test','textDecoration');
```

Remember that `getComputedStyle()` will always return a pixel value, while `currentStyle` retains the unit specified in the CSS.

Shorthand styles

WARNING Browser incompatibilities ahead

Whether you get inline styles through the style property, or other styles through the function we just discussed, you'll encounter problems when you try to read out shorthand styles.

Take this border declaration:

```
<p id="test" style="border: 1px solid #cc0000;">Text</p>
```

Since this is an inline style, you'd expect this line of code to work:

```
alert(document.getElementById('test').style.border);
```

Unfortunately, it doesn't. The browsers disagree on the exact value they show in the alert.

- Explorer 6.0 gives `#cc0000 1px solid`.
- Mozilla 1.7.12 gives `1px solid rgb(204,0,0)`.
- Opera 9 gives `1px solid #cc0000`.
- Safari 1.3 doesn't give any border value.

The problem is that `border` is a shorthand declaration. It secretly consists of no less than twelve styles: width, style, and color for the top, left, bottom, and right borders. Similarly, the `font` declaration is shorthand for `font-size`, `font-family`, `font-weight`, and `line-height`, so it exhibits similar problems.

RGB()

Note the special `color` syntax Mozilla uses: `rgb(204,0,0)`. This is a valid alternative to the traditional `#cc0000`; and you can use either syntax in CSS and JavaScript.

How is the browser supposed to handle such shorthand declarations? The example above seems pretty straightforward; you would intuitively expect the browser to return `1px solid #cc0000`, exactly as the inline style attribute says. Unfortunately, shorthand properties are more complicated than that.

Consider the following case:

```
p {
    border: 1px solid #cc0000;
}
<p id="test" style="border-color: #00cc00;">Test</p>
alert(document.getElementById('test').style.borderRightColor);
```

All browsers report the correct color, even though the inline style doesn't contain a `border-right-color` but a `border-color` declaration. Apparently the browsers consider the color of the right border to be set when the color of the entire border is set; not an unreasonable line of thought.

As you see, browsers have to make rules for these unusual situations, and they have chosen slightly different approaches to handle shorthand declarations. In the absence of a specification for the exact handling of shorthand properties, it's impossible to say which browsers are right or wrong.

B: Changing classes and ids

JavaScript allows you to change the class or id of an element. When you do that, the browser automatically updates the styles of the element.

After the welter of tricky details and browser differences we encountered while discussing the `style` property, class and id changes are a quiet oasis of logic and perfect browser harmony. Consider this example:

```
p {
    color: #000000; /* black */
}
p.emphasis {
```

CLASSNAME, NOT CLASS!

Note that JavaScript uses `className`, because `class` is a reserved word, kept apart for a future in which JavaScript might start to support Java-like classes.

```
    color: #cc0000; /* red */
}
<p id="test">Test</p>
```

Initially, the paragraph doesn't have a class, and its color is therefore black. However, one line of JavaScript is enough to change its styles:

```
document.getElementById('test').className = 'emphasis';
```

Instantly the text becomes red. To change it back, you do the following:

```
document.getElementById('test').className = '';
```

You remove the class, and the paragraph reverts to the standard `p{}` rule.

For a practical example, take a look at Textarea Maxlength. The counter has this structure and presentation (the structure is generated by JavaScript, but that doesn't matter):

```
<div class="counter"><span>12</span>/1250</div>

div.counter {
    font-size: 80%;
    padding-left: 10px;
}
span.toomuch {
    font-weight: 600;
    color: #cc0000;
}
```

When the script sees that the user has exceeded the maximum length, it changes the class of the counter span to `toomuch`:

[Textarea Maxlength, lines 20-23]

```
if (currentLength > maxLength)
    this.relatedElement.className = 'toomuch';
else
    this.relatedElement.className = '';
```

Now the counter `<span>` becomes bold and red.

An id change works exactly the same way:

```
p {
    color: #000000; /* black */
}
p#emphasis {
    color: #cc0000; /* red */
}

<p>Test</p>

document.getElementsByTagName('p')[0].id = 'emphasis';
```

Again, the paragraph becomes red. Nonetheless, I advise you not to change ids too much. Apart from serving as CSS hooks, often they are also JavaScript hooks, and changing them may have odd side effects. In practice you can implement every CSS modification you need as a class change, and you don't have to work with ids.

Adding classes

Often you do not *set* the class of an element to a new value. Instead, you *add* a class value, because you don't want to remove any styles the element might already have. Since CSS allows for multiple classes, the styles of the new class

are added to the element, without removing any instructions that already-existing classes give it.

The `writeError()` and `removeError()` functions of Form Validation are good examples. I generally use a few classes for form fields, because the graphic design often uses two or even three widths for input fields. When a form field contains an error, I want to add a special warning style, but I don't want to disturb any style that the element might already have. Therefore, I cannot simply overwrite the old class value; I'd lose my special widths.

Take this situation:

```
<input class="smaller" name="name" />
input.smaller {
    width: 75px;
}
input.errorMessage {
    border-color: #cc0000;
}
```

Initially, the input has a width of 75px. If the script *sets* the class to 'errorMessage' and discards the old value, the form field would get a red border color but lose its width, and that would be very confusing to the user.

Therefore I *add* the class `errorMessage`:

{Form Validation, lines 105-106]

```
function writeError(obj,message) {
    obj.className += ' errorMessage';
```

This code takes the existing `className` and adds the new class to it, preceded by a space. This space is meant to separate the new class value from any class value the object might already have. Now the form field gets a red border color in addition to its 75px width—exactly what we want. The form field now applies both classes, as if the HTML were:

```
<input class="smaller errorMessage" name="name" />
```

> **CLASS NAMES AND SPACES IN MOZILLA**
>
> You may have noticed that `removeError()` removes the class value `"errorMessage"` without the leading space. That's because of a browser bug. When you add `" errorMessage"` to a class that *doesn't* have a value yet, Mozilla discards the leading space. If we subsequently do `replace(/errorMessage/,'')`, Mozilla doesn't remove the class; it can't find the string `" errorMessage"` because the leading space is not there.

Removing classes

Once the user has corrected her mistake, the class `errorMessage` should be removed, but any original class, such as `smaller`, should not be touched. The `removeError()` function provides this functionality:

[Form Validation, lines 119-120]

```
function removeError() {
    this.className = this.className.replace(/errorMessage/,'');
```

It takes the class of the element and replaces the string `'errorMessage'` by `''` (an empty string). The `"errorMessage"` bit is taken from the class value, but other values are not touched. The form field loses its red border color, but retains its 75px width.

C: Writing CSS into the page

The third method of CSS modification is a rather specialized one that should be used *only* to enhance the accessibility of a page. It works as follows:

```
document.write('<style>element {property: value}</style>');
document.write('<link rel="stylesheet" ➥
href="specialstyles.css" />');
```

The purpose of this method is to define styles that should only be used when JavaScript is enabled. If JavaScript is disabled, or if the browser fails its

ALTERNATIVE: ADDING A <LINK /> TAG

You can also use the W3C DOM to create an extra element that holds the special styles. For instance, you could create a new `<link />` tag, give it the correct `href`, and add it to the document. Since it avoids the use of a `document.write`, some people prefer this method.

However, it has its drawbacks, too. It's impossible to add a `<link />` tag to the document when it hasn't yet been loaded completely; the `<head>` element, to which you must append the `<link />`, is not yet available. Therefore, you must either wait until after the load event, which may cause elements to flicker, or add an extra hard-coded `<link />` tag and give it its proper `href` when JavaScript turns out to be supported.

I leave it to others to decide on the semantic value of empty `<link />` tags, but personally I don't like these potentially useless elements. I prefer the `document.write` method. Feel free to use `<link />` tags, though, if you like them better.

object-detection checks, the `document.write` is never executed, and the styles are never applied.

Dropdown Menus uses this method:

[Dropdown Menus, lines 1-4]

```
var compatible = (document.getElementsByTagName && ➥
document.createElement);
if (compatible)
    document.write('<link rel="stylesheet" ➥
    href="navstyles.css" />')
```

These are the first lines of the script. Note that they are not contained within any function; they are executed as soon as the browser has parsed them, and as we'll see in a moment this is a vital part of such CSS modifications.

The script checks whether the browser supports the W3C DOM, as discussed in 3C. If the browser survives the check, the `document.write` line adds a special style sheet, navstyles.css, to the page. The reason for this special construction is accessibility.

This is the most important part of that style sheet:

```
ul.menutree ul {
    display: none;
    position: absolute;
}
```

When applied without checking the browser support, these styles are actively dangerous. `display: none` hides the submenus, and without JavaScript there's no way to make them visible. The menu becomes inaccessible.

Even if this problem were somehow solved, noscript users would see a confused jumble of submenus, because they're all positioned absolutely in their parent `<li>` and overlap each other when they're all visible. Therefore, I also want to hide the `position: absolute` until I'm certain the browser supports the necessary JavaScript to handle the submenus properly.

FIGURE 9.1
Even without `display: none`, the `position: absolute` makes for inaccessible and unusable dropdown menus. We have to hide this style.

Sandwich Picker does the opposite. The Sandwich Picker page contains this paragraph:

```
<p class="noscript">Unfortunately your browser doesn't support
(enough) JavaScript to use the advanced form. You can always
<a href="mailto:correct@address.nl">mail</a> us to order some
sandwiches.</p>
```

This provides noscript users with a way to order sandwiches. Obviously, this text should be hidden if the browser supports the script, since it'd only confuse "scripted" users. Therefore, the script adds an extra style:

[Sandwich Picker, lines 4-7, condensed]

```
if (W3CDOM) {
    document.write('<style>.noscript{display: none}</style>');
```

If the browser supports sufficient JavaScript, the noscript text is hidden.

As we discussed in 2G, the usability of this solution is not all that it could have been. Nonetheless, technically it's a correct example of hiding elements from the advanced, scripted interface.

Execute immediately

When using this method, it's very important that you apply the special styles *as soon as possible*. If you don't, because you wait for the load event, your users might see an ugly flickering when elements that were visible are suddenly hidden, or vice versa. By adding the special styles up front, the styles are already there when the browser starts loading the HTML, and all elements have the right show/hide instructions at the moment they're created and rendered.

That's why `document.write` is right at the top of the JavaScript file, outside any function. I want it executed while the browser parses the JavaScript file, and before it starts loading the HTML.

DOCUMENT.WRITE AND PAGE LOADING

It's important to repeat a feature of `document.write` from 6F: if you use it *after* the page has been loaded completely, the browser must create a new page to contain the new content, and it destroys the old page in the process. Therefore, the user sees a blank page with a cryptic line of code instead of your carefully crafted interface.

This is another reason to execute the `document.write` immediately.

Incidentally, XMLHTTP Speed Meter does it wrong. The script should hide the form's Submit button when sufficient JavaScript is supported, but it does so only at onload, which is too late:

[XMLHTTP Speed Meter, lines 19-22]

```
window.onload = function () {
    var supportCheck = [check support];
    if (!supportCheck) return;
    document.getElementById('submitImage').style.display = 'none';
```

If the page is slow in building, visitors will first see the submit image, but then all of a sudden it's hidden. That is confusing, and I should have used the `document.write()` trick here, too.

Used correctly, this method is a powerful accessibility tool. Used incorrectly, it can cause serious trouble. Always think out all the scenarios before using `document.write` to add styles, and test your pages with JavaScript disabled.

D: Changing entire style sheets

WARNING Browser incompatibilities ahead

Changing entire style sheets works in Internet Explorer for Windows and Mozilla, and in no other browsers. Therefore this technique isn't ready for prime time yet, apart from the fact that editing an entire style sheet rarely

makes sense. However, we'll take a look at it anyway, since I use this technique in Edit Style Sheet.

Theory

A document contains one or more style sheets, each of which contain one or more rules. Every rule contains one selector and an unlimited number of style declarations. Just as with the inline styles, these style declarations are accessible through the `style` property of the rule.

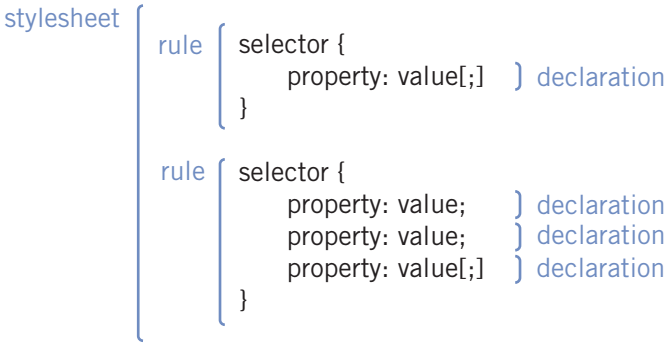

FIGURE 9.2 Every style sheet contains one or more rules. Every rule contains a selector and one or more declarations, each of which consists of a name/value pair.

In addition, Explorer and Mozilla support the addition and removal of entire rules from the style sheet. Although Edit Style Sheet doesn't use this feature, we'll treat it below.

Although the addition and removal of rules has been specified by W3C, Explorer supports only its own version. So a bit of object detection is necessary when you want to use these features.

document.styleSheets

The first requirement for successful style-sheet editing is support for the `document.styleSheets` nodeList. If the browser doesn't support that, the script won't work.

Therefore, Edit Style Sheet's `initStyleChange()` function starts by checking support:

[Edit Style Sheet, lines 5-7]

```
function initStyleChange() {
    if (!document.styleSheets) return;
    var sheets = document.styleSheets;
```

If the browser doesn't support `document.styleSheets` at all, the function ends immediately. Otherwise, the script creates a `sheets` variable that refers to the `document.styleSheets` nodeList.

`document.styleSheets` contains all style sheets in the document, whether they're embedded or linked. Take this bit of HTML:

```
<head>
<link type="text/css" rel="stylesheet" href="main.css" />
<link type="text/css" rel="stylesheet" href="colors.css" />
<style type="text/css">
.specialCase {
    color: #cc0000;
}
</style>
</head>
```

Now `document.styleSheets` contains three style sheets: the linked main.css, the linked colors.css, and the embedded .specialCase rule. As usual, they're numbered in the order they appear in the source code, and the first one has index 0. Therefore, the embedded style sheet is `document.styleSheets[2]`.

Edit Style Sheet has to search for the style sheet that users are allowed to edit. In my example script, it's called colors.css:

[Edit Style Sheet, lines 8-13]

```
for (var i=0;i<sheets.length;i++) {
    var ssName = sheets[i].➥
    href.substring(sheets[i].href.lastIndexOf('/')+1);
```

```
        if (ssName == 'colors.css')
                var currentSheet = sheets[i];
    }
    if (!currentSheet) return;
```

The script goes through all style sheets and searches to their `href` property to find the style-sheet name after the last slash. If this name is equal to colors.css, the script has found the correct style sheet, and stores it in `currentSheet`.

If `currentSheet` doesn't have a value after this loop, colors.css is missing, and the function ends.

cssRules[] and rules[]

WARNING Browser incompatibilities ahead

Edit Style Sheet needs access to all the rules in the style sheet. Mozilla uses the W3C-specified `cssRules[]` nodeList, while Explorer needs the Microsoft-proprietary `rules[]` nodeList. Fortunately, these two nodeLists are exactly the same except for their names:

[Edit Style Sheet, lines 14-18]

```
    if (currentSheet.cssRules)
        sheetRules = currentSheet.cssRules;
    else if (currentSheet.rules)
        sheetRules = currentSheet.rules;
    else return;
```

Therefore, a simple object detection is enough. If the browser supports `cssRules[]`, store this nodeList in `sheetRules`; if it supports `rules[]`, store that nodeList in `sheetRules`. If the browser supports neither, the script won't work, and it ends.

Note that `sheetRules` is a global variable; other functions will need this information, too.

selectorText

Every rule has a `selectorText`, which contains, unsurprisingly, the rule selector as a string.

When the user selects a new rule to edit from the first select, the `assignRule()` function searches for this rule and opens it for editing:

[Edit Style Sheet, lines 48-55]

```
function assignRule() {
    var selector = this.value;
    for (var i=0;i<sheetRules.length;i++)
            if (sheetRules[i].selectorText.toLowerCase() ➥
            == selector.toLowerCase())
                    currentRule = sheetRules[i];
}
```

It takes the desired selector (i.e., the select box's `value`), and then goes through all the rules in `sheetRules`. When it finds a rule with a matching `selectorText`, it points `currentRule` to this rule. `currentRule`, too, is a global variable, since other functions need access to the rule that's currently being edited.

One word of warning: some browsers (Explorer and Safari at the time of writing) return an UPPERCASE selector text. Therefore, the above function compares `selectorText.toLowerCase()` to `selector.toLowerCase()`.

style

Every rule has a `style` property that works exactly like the `style` property of individual HTML elements we discussed in 9A. If we wanted to change the `color` of `currentRule` to green we could do this:

```
currentRule.style.color = '#00cc00';
```

Now all elements that take their style information from this rule will change their color to green.

The `assignStyles()` function is called whenever the user changes a style declaration through the main form. It sets the correct style of the current rule to the correct value:

[Edit Style Sheets, lines 57-64]

```
function assignStyles() {
    if (!currentRule) return;
    var styleName = this.name;
    var styleValue = this.value;
    if (this.type == 'checkbox' && !this.checked)
            styleValue = '';
    currentRule.style[styleName] = styleValue;
}
```

First it checks for a `currentRule`, and if there is none, it exits. Then it takes the name and value of the form field the user has changed:

```
<input type="checkbox" name="fontWeight" value="bold" />
```

Checkboxes need special treatment: if they are not checked, the CSS property value they set should be empty. We already discussed the last line of this function in 5G. It sets the correct style property of the current rule (`currentRule.style[styleName]`) to the correct value.

Now the browser implements the revised styles, and the user immediately sees the effect of her changes.

cssText and submitting the style sheet

Since Edit Style Sheet never left the prototype phase, I never devised a way to send the edited style sheet back to the server. Nonetheless, the script is useless without some sort of submission that allows the server to store and use the updated style sheet.

The simplest way would be to copy the entire text content of the style sheet, store it in a hidden textarea, and send it to the server. A server-side program

would create a neat file and store it as colors.css. Thus the style sheet would become available to all users of the site.

WARNING Browser incompatibilities ahead

The problem is that there's no simple cross-browser way of reading the entire text of the style sheet. The W3C DOM CSS module contains the `cssText` property, but the browsers don't entirely agree on the scope of this property.

`cssText` contains style declarations as a string. But which style declarations? There are three ways to use `cssText`:

```
1. document.styleSheets[0].cssText;
2. document.styleSheets[0].cssRules[0].cssText;
3. document.styleSheets[0].cssRules[0].style.cssText;
```

The first example would give the entire style sheet as a string; that's exactly what we need for Edit Style Sheets. If we pasted `document.styleSheets[0].cssText` into a hidden text area, we'd be ready.

Unfortunately, Mozilla doesn't support this use of `cssText`. For this reason, I was sorely tempted to restrict this script to Explorer Windows, but its implementation was delayed several times, and in the end I didn't solve this conundrum.

BROWSER COMPATIBILITY

At the time of writing, the first use of cssText was supported only by Explorer (Windows and Mac), the second one only by Explorer Mac and Mozilla, while only the third one is supported by all browsers. These compatibility patterns may change, though.

Inserting and deleting rules

You can insert new rules into a style sheet, or remove rules from it. This feature is not used in Edit Style Sheet—I definitely don't want a user who knows little of CSS to insert new rules that might mess up the entire page.

WARNING Browser incompatibilities ahead

Nonetheless, inserting and deleting rules is important enough to treat. Unfortunately, this area of JavaScript again contains some browser incompatibilities. Mozilla uses the W3C-specified `insertRule()` and `deleteRule()` methods, while Explorer uses the Microsoft-proprietary `addRule()` and `removeRule()` methods. Safari and Opera don't support these actions at all.

As soon as you insert a rule, the style declarations in it are immediately applied to all relevant elements. Deleting a rule immediately removes these style declarations from all relevant elements.

insertRule() and addRule()

When you are inserting a new rule, the browsers expect three bits of information:

1. The selector of the new rule.
2. The style declarations of the new rule, as a string.
3. The position of the new rule (first rule of the style sheet, last rule, somewhere in between?).

The W3C `insertRule()` and Microsoft `addRule()` methods need this information, but they differ in the way they want it delivered.

W3C's `insertRule()` expects two arguments:

1. The entire rule (selector + declarations) as a string.
2. The position of the new rule (0 = first rule in the style sheet, etc.).

Example:

```
var x = document.styleSheets[0];
x.insertRule('PRE {font: 0.9em verdana}',2)
```

Now this CSS rule is added to the first style sheet of the page, as the third rule (with index 2).

```
pre {
    font: 0.9em verdana;
}
```

Microsoft's `addRule()` expects three arguments:

1. The selector as a string.
2. The declarations as a string.
3. The position of the new rule.

Example:

```
var x = document.styleSheets[0]
x.addRule('PRE', 'font: 0.9em verdana',2)
```

This gives exactly the same result as the previous code example: the same CSS rule is added as the third rule in the style sheet.

deleteRule() and removeRule()

If you want to delete a rule, you should specify which one by giving its index number. So in order to delete the rule we just inserted, we should do this:

```
var x = document.styleSheets[0]
x.deleteRule(2); // W3C
x.removeRule(2); // Microsoft
```

Now the third rule in the first style sheet is deleted.

Helper functions

In order to defeat these rule-related browser incompatibilities, we have to create two helper functions. Deletion is easy:

```
function deleteCSS(sheet,index) {
    if (sheet.deleteRule)
            sheet.deleteRule(index);
    else if (sheet.removeRule)
            sheet.removeRule(index);
}
```

Inserting a rule is slightly more complicated because of the different arguments the functions expect:

```
function insertCSS(sheet,selector,declarations,index) {
    if (sheet.insertRule) {
            var toBeInserted  = selector + ➥
            ' {' + declarations + '}';
            sheet.insertRule(toBeInserted,index);
    }
    else if (sheet.addRule)
            sheet.addRule(selector,declarations,index);
}
```

E: Comparison

Now that we have studied all four CSS-modification methods, we can ask ourselves which one is the best. The `document.write` method should be used only in specialized accessibility-related cases. Changing an entire style sheet is rarely useful. Therefore two methods remain. Should you use the `style` property, or change class or id values?

To get some answers, let's take a look at the example scripts. Site Survey doesn't modify CSS at all, Edit Style Sheet changes the entire style sheet, and Usable Forms uses the `document.write` method. Only XMLHTTP Speed Meter changes `style` properties, while four scripts—Textarea Maxlength, Sandwich

Picker, Form Validation, and Dropdown Menu—use class changes. As you might guess, I feel the class change is usually the superior CSS-modification technique.

In my opinion, excessive reliance on the `style` property violates the separation of presentation and behavior, and besides it's more complicated to code. Styles belong in the CSS presentation layer—their natural habitat—not in either the HTML structural layer or the JavaScript behavior layer.

When you change a style, you are saying, "Give this element a new presentation." Giving this command is obviously a job for JavaScript. However, *defining* the new presentation is a job for CSS. The new styles of the element, as well as the old ones, should therefore be defined in the CSS file.

Thus presentation and behavior remain separate: JavaScript gives the command to change styles, but it's up to the CSS to define the new styles.

Example

Let's take a look at a practical example. Suppose I'd used the `style` property instead of a class change in Form Validation:

[Form Validation, lines 105-6 and 119-120, changed]

```
function writeError(obj,message) {
    obj.style.border = '1px solid #cc0000';

function removeError() {
    this.style.border = '';
```

Of course this works fine. But suppose later on the graphic designer decides the text also has to become red. You'd have to add two lines to your script:

```
function writeError(obj,message) {
    obj.style.border = '1px solid #cc0000';
    obj.style.color = '#cc0000';

function removeError() {
    this.style.border = '';
    this.style.color = '';
```

And so on. Each extra style for form fields with errors would take two lines of JavaScript. In contrast, a class-change script is easier to maintain, since an extra style requires only one line of CSS.

Of course, adding two lines of JavaScript hardly counts as complex programming. Nonetheless, implementing these changes would still require someone who knows his way in the JavaScript code. A graphic designer/CSS wizard without any JavaScript experience would not be able to do it.

CSS developers who know nothing about JavaScript *are* able to edit a style sheet, and this is not an inconsiderable advantage in a larger company, where people tend to have more specialized jobs. As a JavaScript developer, you don't want every tiny style change to have to go through you. Therefore, decreasing the complexity of the JavaScript layer is to your own advantage, too.

So there are a few reasons why I believe that in general a class change is superior to a `style` property change. Nonetheless, like any good general rule, this one has exceptions. The two most important ones are simple show/hide scripts and animations.

F: Showing and hiding elements

In many cases, you want to show and hide elements based on user actions. These cases may be complicated, like Usable Forms, or they may be extremely simple, such as a menu that folds in or out when the user clicks on a header.

HIDE ELEMENTS IN JAVASCRIPT!

As we saw in 2F, for accessibility's sake we should give the initial commands to hide content *in JavaScript*. That way, if a visitor to your site doesn't have JavaScript enabled, the content is never hidden, and although the page may lose some usability, it's still perfectly accessible.

Simple show/hide scripts

Simple show/hide scripts display or conceal only one element per user action. Most of these scripts change the `style.display` property of the element.

Mostly this method is used out of tradition: when people started writing show/hide scripts, class changes weren't yet possible in Netscape 4 (an important browser back then). Besides, show/hide scripts generally change only one CSS property: `display`.

Therefore, this is a typical example of a show/hide script. It uses style-property changes throughout, and doesn't touch the class names of the elements:

```
<h3>Header</h3>
<div class="content">
    <p>Content</p>
    <p>Content</p>
</div>

window.onload = function () {
    var x = document.getElementsByTagName('div');
    for (var i=0;i<x.length;i++) {
            if (x[i].className != 'content') continue;
            x[i].style.display = 'none';
            var header = x[i].previousSibling;
            if (header.nodeType != 1)
                    header = header.previousSibling;
            header.relatedTag = x[i];
            header.onclick = openClose;
    }
}
function openClose() {
    var currentValue = this.relatedTag.style.display;
    var newValue = (currentValue == 'none') ? 'block' : 'none';
    this.relatedTag.style.display = newValue;
}
```

The script goes through all s in the document, and ignores those that don't have a class="content". Then it sets their display to none (styles that hide content should be set in JavaScript!).

Then the script finds the header the is related to. In principle, this is the previousSibling of the , but as we saw in 8H there may be empty text nodes between the header and the . Therefore, the script goes back through the document tree until it finds a node with nodeType 1 (i.e., an element node). This is the header it's looking for.

Once it finds this header, it sets its relatedTag property to point to the , and the header gets an onclick event handler.

When the user clicks on the header, the openClose() function reads out the current display of the header's relatedTag (i.e., the that should be opened or closed), and sets its display to the opposite value.

GENERATING THE DIVS

Sometimes you don't want the extra div class="content" in your HTML code. It has no semantic meaning, and editors of the HTML page could suddenly add extra headers. In addition, if the page is updated through a CMS, it would have to generate an extra div class="content" around the correct content. You'd just have to hope the CMS you're working with could do that.

Enter the W3C DOM. Don't put any divs in your HTML, but generate them when the page has been loaded. Go through all children of the <body>. As soon as you find a header, append a new div after it. Then move all nodes you find after the header to this generated div, until you encounter a new header. Then generate a new div, and continue with the content after the new header.

Try expanding the show/hide script with this new functionality. It's a useful DOM exercise.

Note a few important points:

- The crucial `display: none` is set in JavaScript, not in CSS. If a noscript browser visits the page, all blocks remain open and accessible.
- We set the inline style of the `<div>` to `display: none` so that we can read out the current `display` of the block through `style.display`.
- When the script has loaded `(onload)` we relate every header to the content block it should open. We discussed the creation of such relations in 4C.

This simple example of a show/hide script can be used in many circumstances.

Showing and hiding table rows

In my script I used `<div>`s as example tags, but of course you can write a show/hide script for any tag you like. Showing and hiding `<ul>`s or `<li>`s in a navigation could also be useful; in fact, Dropdown Menu does exactly that (though in a more complicated way than the simple script we just reviewed).

WARNING Browser incompatibilities ahead

One case deserves special attention: showing and hiding table rows. Hiding table rows is done through `display: none`, obviously, but how do you show them? Table elements ought to have the special `display` value `table-row`, but unfortunately Explorer uses `display: block` instead.

TABLE DISPLAY VALUES

According to the CSS specification tables, table rows and table cells should not use `display: block`, but rather `display: table`, `display: table-row`, and `display: table-cell`, respectively. However, Explorer uses `block` throughout.

All browsers are strict about what they allow. If they expect `table-row`, they'll choke on `block`, and vice versa. For instance, the following line makes Explorer restore the `<tr>` to view, but the other browsers literally make the `<tr>` a block that doesn't pay attention to the rest of the table:

```
var tr = [the tr you want to show];
tr.style.display = 'block';
```

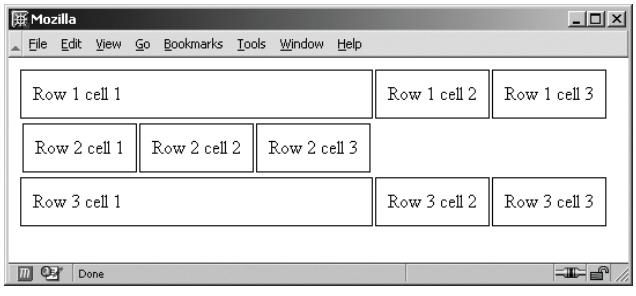

FIGURE 9.3
I gave the middle `<tr> display: block`. Mozilla takes me at my word; it makes the `tr` a block, and the vertical alignment of `<td>`s is lost.

Conversely, if you use the correct `table-row` value, the other browsers restore the `<tr>`, but Explorer Windows gives an error:

```
tr.style.display = 'table-row';
```

What now? Do we have to use a browser detect to decide between `block` and `table-row`? Fortunately, no. Every browser has a default style sheet that declares all default styles of all elements, and this style sheet includes the default `display` value of a `<tr>`.

This default style sheet contains either `block` or `table-row`, depending on the browser. If you allow this sheet to take over once more, the problem is solved; all browsers will restore the `<tr>` to view.

How do you do this? As we saw at the start of this chapter, the `style` property sets the inline style of an element. If we remove the inline value entirely, the browser once again takes its `display` instructions from the default style sheet, and the problem is solved:

```
tr.style.display = '';
```

By removing the inline style entirely, the default style sheet takes over and the `<tr>` becomes visible in all browsers. Better still, you don't have to know whether the browser needs `'block'` or `'table-row'`.

G: Animations

Animations draw the eye of the user to the element that's animated. If that's a good thing, the animation serves a good purpose; but if not, it doesn't.

XMLHTTP Speed Meter is the only example script that uses an animation. Although it's there to provide some eye candy, it also serves a real purpose. By making the animation fluctuate within a range of download speeds, the interface conveys the message "Your download speed will be somewhere around here," without specifying an exact speed. This is very important to the client—an ISP who simply cannot guarantee an exact speed, and wants its clients to be aware of that fact.

How animations work

Animations are repeated changes to the same style of the same element. With a bit of luck, the style changes follow each other so rapidly that the user perceives the animation as one continuous movement instead of a lot of small steps.

Nonetheless, JavaScript animations may become jerky for several reasons: the script isn't fast enough; the user's computer is slow; the page has many scripts running in addition to the animation; or the animation uses too many tiny steps. If perfect animations are an absolute requirement for your site, Flash remains the best tool.

Animations are one area in which changing the `style` property is better than a class change. If XMLHTTP Speed Meter used class changes, I'd have to define dozens of classes: one for each possible animation step. That would make the CSS presentation layer extremely complicated.

XMLHTTP Speed Meter uses two nested `<div>`s. The outer one contains the grey background image, and the inner one the black image. Initially, the inner

 has width: 0 and is thus invisible. The animation script changes the width of the black , and since its background has exactly the same position as the outer background, it appears to become a black meter running over a greyed background:

```
<div class="grey">
    <div id="meter">
    </div>
</div>
div.grey {
    background: url(pix/bg_meter_grey.gif); // grey bg
    height: 23px;
    width: 280px;
}
div#meter {
    background: url(pix/bg_meter_black.gif); // black bg
    height: 23px;
}
```

[XMLHTTP Speed Meter, lines 70-73]

```
function setWidth(width) {
    if (width < 0) width = 0;
    document.getElementById('meter').style.width = width + 'px';
}
```

FIGURE 9.4
Initially, the meter has width: 0 (invisible); later, it has width: 56px and obscures part of the grey background.

Changing the `width` is pretty easy (as long as you remember to append a correct unit). However, the `width` shouldn't change once, but many times over. How do we cleanly issue a lot of commands to change the `width`?

setTimeout and setInterval

In order to create a proper animation, you need either `setTimeout` or `setInterval`. Before discussing the use of these methods, let's take a peek at the problem they solve.

If you command the browser to change widths multiple times without any pause, it will change the on-screen rendering only after the entire script has been run. Suppose we want to animate a change from 0 to 56px width, with 14px intervals:

```
function changeWidths() {
    setWidth(14);
    setWidth(28);
    setWidth(42);
    setWidth(56);
}
```

Now the browser faithfully changes the `width` of the element to 14, 28, 42, and 56 pixels. However, it doesn't show any of these changes until the function has ended. Therefore, as far as the user is concerned, the black bar is invisible at first (`width: 0`), and then all of a sudden it has a substantial size (`width: 56px`), without any steps in between. This is no animation—you might just as well have set `width: 56px` immediately.

We have to pause between the steps to give the browser the chance to catch up and to show every single step on the screen before proceeding to the next one. We have to time our commands: do `setWidth(14)`, wait, do `setWidth(28)`, wait, etc.

As we discussed in 6E, JavaScript contains two methods that allow you to time your commands: `setTimeout` and `setInterval`. They come in especially handy in animations.

Using setTimeout

We're going to use `setTimeout` to define a slight waiting period between the `width` changes. (Why not `setInterval`? We'll discuss that later.)

```
function changeWidths() {
    setTimeout('setWidth(14)',100);
    setTimeout('setWidth(28)',200);
    setTimeout('setWidth(42)',300);
    setTimeout('setWidth(56)',400);
}
```

Now the animation works. The browser is commanded to run `setWidth()` four times: once with argument 14 after 100 milliseconds, once with argument 28 after 200 milliseconds, etc. As far as the browser is concerned, after executing the function once, it considers the function ended, and shows the results on-screen.

Now the user sees a simple animation, each step of which makes the black bar 14 pixels wider, with a 100 millisecond pause between each step. The style change has become a true animation.

USING THE RIGHT INTERVAL TIME

An important part of creating a good animation is choosing an appropriate interval for your steps. In theory, the smaller the time, the better the animation, since small steps make an animation more fluid. Unfortunately, if you make your steps too small, browsers start to have trouble.

10 milliseconds seems to be a practical limit; it's about the smallest delay time browsers can handle. But even if you stay above this limit—setting, say, 500 animation steps with a 20-millisecond interval—your script may use too many browser resources, since it runs 500 times in a 2-second period. Of course, the exact behavior of the browser depends on the speed of the computer it runs on, so some browsers might still show such an animation correctly.

I generally use a delay time of 50 milliseconds at minimum, and restrict the number of steps to a moderate amount, say 10 to 50.

When creating an animation, I always allow the client to set the exact animation time. Clients frequently think an animation is slightly too slow or too fast, and by allowing them to modify one variable by themselves, I save myself valuable development time.

So let's add a variable `moveTime`, which defines the pause between the animation steps:

```
var moveTime = 100; // in milliseconds
function changeWidths() {
    setTimeout('setWidth(14)',moveTime*1);
    setTimeout('setWidth(28)',moveTime*2);
    setTimeout('setWidth(42)',moveTime*3);
    setTimeout('setWidth(56)',moveTime*4);
}
```

That's a useful addition, but the function is still quite ugly. Looking for ways to further sanitize it, we see that what we're really doing is giving a repetitive command, and such commands can be worked into a `for{}` loop:

```
var moveTime = 100; // in milliseconds
function changeWidths() {
    for (var i=1;i<5;i++) {
            setTimeout('setWidth(' + (i*14) + ')',moveTime*i);
    }
}
```

This is much better. `i` loops from 1 to 4. The width that the black meter should be set to is always `i*14` (14, 28, 42, 56), and the timeout time is always `moveTime*i` (100, 200, 300, 400). Now the animation function has become much more standardized, and all timeouts are set at once, reducing the clutter in your script.

setTimeout or setInterval?

Why do I use `setTimeout`, not `setInterval`? In my opinion, the two methods, though similar, are meant for different situations, especially when it comes to animations. Before continuing, let's review the difference:

```
setTimeout('theFunction()',100);
setInterval('theFunction()',100);
```

The first line means "Execute function `theFunction()` 100 milliseconds from now." The second line means "Execute function `theFunction()` 100 milliseconds from now, and *continue* executing it every 100 milliseconds."

`setInterval` is best for animations that continue for an indefinite time (for instance, until the user does something). `setTimeout`, on the other hand, is best suited for animations that have a fixed start and end point.

The animation that moves the speed meter to a new value has a fixed start point (the old value) and a fixed end point (the new value.) Therefore this animation requires `setTimeout`.

Once the XMLHTTP Speed Meter has arrived at its new value, however, a second animation kicks in. This is the fluctuation that continues until the user does something. Since the fluctuation doesn't have a fixed start and end point but should continue until further notice, this animation requires `setInterval`.

Examples

XMLHTTP Speed Meter initializes the two kinds of animations. When a new speed arrives from the server, `moveToNewSpeed()` is called. We'll ignore the calculations and focus on setting and cancelling the timeouts and intervals:

[XMLHTTP Speed Meter, lines 78-86, condensed]

```
function moveToNewSpeed(Mbit) {
    for (var i=0;i<animationSteps.length;i++)
            clearTimeout(animationSteps[i]);
    animationSteps.length = 0;
    // calculations
    clearInterval(fluctuationInterval);
```

As soon as `moveToNewSpeed()` is called, it clears all timeouts and intervals that may exist; all animations should stop what they're doing and await new orders.

All timeouts are stored in the `animationSteps[]` array. This is done in order to easily remove them in the `clearTimeout` routine at the start of this function. The function cannot be sure in advance of the exact number of timeouts, since this varies with the start and end points of the desired animations. Storing them in an array is therefore the best solution; the array can become as long as necessary.

[XMLHTTP Speed Meter, lines 90-94]

```
        do {
                animationSteps[timeoutCounter] = ➥
                setTimeout('setWidth(' + (pos*7) + ')', ➥
                timeoutCounter*moveTime);
                timeoutCounter++;
        } while ([perform some calculations])
```

Now it calls the necessary `setTimeouts` in a way similar to the `changeWidths()` function we discussed previously. `timeoutCounter` counts the timeouts that are already set, so that every timeout gets its own entry in the `animationSteps` array. `pos*7` is the desired width, so we feed this value to `setWidth()`. We already discussed the `do/while()` loop in 5H.

[XMLHTTP Speed Meter, lines 96-97]

```
        setTimeout('initFluctuation()',timeoutCounter*moveTime);
    }
```

Then the script sets a last timeout to initialize the fluctuation. `timeoutCounter` has increased since the last timeout, so `initFluctuation()` is called after the last time `setWidth()` has been called, and the fluctuation takes over seamlessly:

[XMLHTTP Speed Meter, lines 102-107, condensed]

```
    function initFluctuation() {
        // calculations
        fluctuationInterval = ➥
        setInterval('fluctuate()',fluctuationTime);
    }
```

`initFluctuation()` performs some calculations, and then sets an interval that calls `fluctuate()` every once in a while. The exact time between fluctuations is set by the client and stored in `fluctuationTime`:

[XMLHTTP Speed Meter, lines 109-116, condensed]

```
function fluctuate() {
    // calculations
    setWidth(currentFluctuation*7);
}
```

`fluctuate()`, finally, performs some more calculations and calls `setWidth()`. This function continues to be called until new data arrives from the server and the interval is cancelled by `moveToNewSpeed()`.

In this way, both animations are set and cleared in the most efficient way possible.

H: Dimensions and position of elements

To wrap up this chapter, we will discuss how you can find an element's real dimensions and position in pixels.

Element dimensions

In some situations, you need to find the exact dimension of an element—and that includes the browser window. You can use a few properties that, though not part of any standard, are defined for all HTML elements in all browsers. They all yield a width or height in pixels.

- The `clientWidth` and `clientHeight` properties give the width and height of the visible part of an element (i.e., CSS `width` + `padding`). They don't take borders or scrollbars into account, nor any possible scrolling.
- The `offsetWidth` and `offsetHeight` properties give the total width and height that the element takes up in the page. The only difference from the previous property pair is that these two take into account the borders and scrollbars of an element.

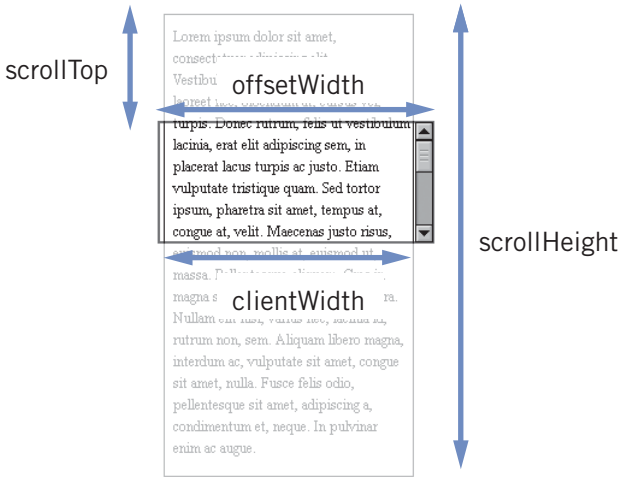

FIGURE 9.5
What `scrollTop`, `offsetWidth`, `scrollHeight`, and `clientWidth` measure. (`scrollLeft`, `offsetHeight`, `scrollWidth`, and `clientHeight` work the same, but in the other dimension.)

- The `scrollWidth` and `scrollHeight` properties give the total width and height of the element as if it had `overflow: visible`. If this width and height are larger than the `clientWidth` and `clientHeight`, the element needs scrollbars.
- The `scrollTop` and `scrollLeft` properties give distance (in pixels) that the element has scrolled. When you set these properties, the page scrolls to the new coordinates.

Therefore, the following gives the real width in pixels of the element with id="test":

```
document.getElementById('test').clientWidth;
```

To get the width of the borders and the scrollbars, use `offsetWidth` instead.

Finding the browser window's dimensions

Since these measurement-related properties are defined for all elements, they also work on `<body>` and `<html>`. In fact, it's on these elements that they're most commonly used. If you take the `clientWidth` and `clientHeight` of `<body>` or `<html>`, you find the visible width and height of the browser window, and that's useful in many scripts (though not in any of the example scripts).

As we saw in 8B, the <html> and <body> tags are represented by document.documentElement and document.body. Therefore, one of these lines will give you the browser window's width (without the scrollbars):

```
document.documentElement.clientWidth;
document.body.clientWidth;
```

The second line is meant for Explorer versions earlier than 6, while the first line is meant for all other browsers. The difference is the interpretation of the <body> tag. In ancient times, the <body> element was the topmost visible element, and the <html> element remained hidden. However, modern browsers made <body> a normal block-level element, while <html> encompasses the entire browser window.

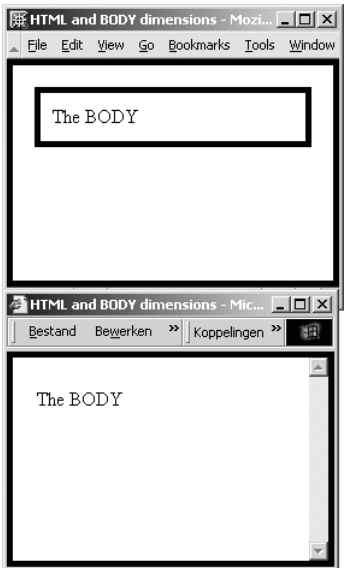

FIGURE 9.6
In Mozilla, the <html> element encompasses the entire browser window, while the <body> element is a normal block-level element. In Explorer 5.5, the <body> element encompasses the browser window, while the <html> element is invisible.

Hence, to read out the window's width and height, you have to query the clientWidth of the <html> element in modern browsers, but of the <body> element in older ones. Fortunately, that's simple:

```
var windowWidth = document.documentElement.clientWidth
        || document.body.clientWidth;
```

OLDER PROPERTIES

In all browsers but Explorer, the older property pairs `window.innerWidth/Height` and `pageXOffset/pageYOffset` give the browser window's width and height and the scrolling offset, respectively. However, since all these browsers also support the `clientWidth/Height` and `scrollTop/Left` properties, there's no more need to use the old ones.

The window width is `document.documentElement.clientWidth`, but if that property is 0 or doesn't exist, we take `document.body.clientWidth`.

Element position

Occasionally, you want to find an element's position in the window. Edit Style Sheet needs this information when it wants to position the color picker right next to the link that says 'pick'; it has to know the position of this link.

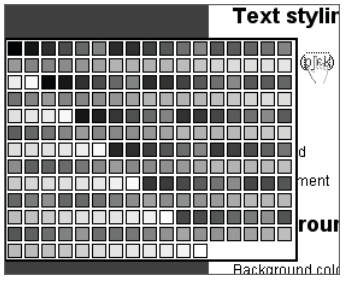

FIGURE 9.7
The color picker should be positioned next to the link the user clicked on. To do this, the script needs to find the position of this link in the browser window.

Any element has two properties, `offsetLeft` and `offsetTop`, that give the offset of the element in pixels. The question is, of course, the offset relative to which element? The browsers disagree sharply on this. Explorer calculates the position of the Pick link relative to its `parentNode` (the `<label>` tag), while Mozilla calculates it relative to the `div#container`, because that's the first ancestor to have a CSS `position` other than `static`.

At first, this seems to be one of those unsolvable cross-browser incompatibilities. Fortunately, all browsers agree that the property `offsetParent`

PARENTAL PROBLEMS

The `offsetParent` function is not perfect, and although it'll work fine in 90% of the cases, sometimes it needs a bit of help.

One drawback is that the `<body>` and `<html>` elements don't have an `offsetParent` themselves, and thus no `offsetWidth/Height`. If you give either element a CSS `margin` or `padding`, this value may not be added to the calculated offset. Fortunately, I did not define any `margin` or `padding` to the `<body>` or `<html>` element in Edit Style Sheet.

In addition, the function doesn't work perfectly on elements with `position: relative` in Explorer.

See http://www.quirksmode.org/js/findpos.html for an overview of the problems you can expect, as well as a few test cases.

refers to the element relative to which the offset is calculated. In Explorer, the link's `offsetParent` refers to the `<label>`, but in Mozilla it refers to the `div#container`.

That means that we can jump from `offsetParent` to `offsetParent` without having to worry about their exact identity. We just follow the trail of `offsetParent`s until we end up at the `<body>` or `<html>` tag.

We first calculate the link's offset relative to its `offsetParent`, and then we go to this `offsetParent` and add its offset relative to its own `offsetParent`, etc. When we're finished, we've found the link's offset relative to the `<body>` or `<html>` tag.

The `findPos()` function does so:

[Edit Style Sheet, lines 181-191]

```
function findPos(obj) {
    var curleft = curtop = 0;
    if (obj.offsetParent) {
            while (obj.offsetParent) {
```

```
                curleft += obj.offsetLeft;
                curtop += obj.offsetTop;
                obj = obj.offsetParent;
            }
        }
        return [curleft,curtop];
    }
```

The return value of this function is an array (see 5L). The function first sets `curleft` and `curtop` to 0. If the `offsetParent` property does not exist, it returns [0,0]: calculation impossible.

Otherwise it enters a `while()` loop. While the current object has an `offsetParent`, it adds the `offsetTop/Left` of the current object to `curleft` and `curtop`, and then moves on to the `offsetParent` of the current object. The `while()` loop makes sure that it continues to do so until there is no more `offsetParent`.

Once this loop has ended, we find the element's offset relative to the `<body>` or `<html>` element.

`findPos()`'s results are used as follows:

[Edit Style Sheet, lines 138-142, condensed]

```
    function placeColorPicker() {
        var coors = findPos(this);
        colorPicker.style.top = coors[1] - 20 + 'px';
        colorPicker.style.left = 0;
```

`findPos()` returns an array, so `coors` becomes an array. In this specific example, the color picker ignores any left value but instead sets its `left` to 0. The element's `top`, though, is taken from `findPos()`'s results, 20 pixels are subtracted, and the 'px' unit is appended. Now the color picker appears to the left of the link the user clicked on.