

8

Print Magic: Using the DOM and CSS to Save the Planet

ian
lloyd

accessify.com

Ian Lloyd runs Accessify.com, a site dedicated to promoting web accessibility and providing tools for web developers. His personal site, Blog Standard Stuff, ironically, has nothing to do with standards for blogs (it's a play on words), although an occasional standards-related gem can be found there.

Ian works full-time for Nationwide Building Society, where he tries his hardest to influence standards-based design ("to varying degrees!"). He is a member of the Web Standards Project, contributing to the Accessibility Task Force. Web standards and accessibility aside, he enjoys writing about his trips abroad and recently took a "year out" from work and all things web (but then ended up writing more in his year off than he ever has). He finds most of his time being taken up by a demanding old lady (relax, it's only his old Volkswagen camper van).

Ian recently wrote his first book for SitePoint, entitled *Build Your Own Web Site the Right Way Using HTML & CSS* (in which he teaches web standards-based design to the complete beginner).



A printing technique is born

Saving the planet? What's that all about? Well, I'll be getting to that in due course. First, I want to take you back in time. OK, so it's only a month ago as I write this, but it's the point at which I got the inspiration for what you'll be reading in this chapter.

I was sitting at my desk at work, busy with something or other, when my colleague said to me, "Is there any way that you can get a web page to print only one specific part?"

"Oh yes," I replied, "That's easy—you can use print CSS styles to do that. Which part of the page do you want to print and which parts don't you want to print?"

"Well, that depends," he replied, preparing to throw the proverbial curveball. "We've got this great big long page of interest rates, but depending on the account the person has, they'll want a different section to be printed."

I mulled it over. I remembered that on some browsers it was possible to select a portion of a page and then print just that section, an option offered in the Print dialog box, but this was hardly ideal, as it required the user to know that option existed. Also, it's not available on all browsers. For the über-techy types, it would be possible to use a Greasemonkey¹ script to print a given selection, but that would probably account for 0.001% of our intended audience at a push. What my colleague was after was something that the browser simply did not offer in any obvious format. So I told him, "Sorry, you can't do that. I'm afraid that unless they know about highlighting a section and then choosing 'print selection,' it's not possible."

Then I got back to whatever pixels I was pushing around the screen at the time. However, the idea refused to go away, nagging at the back of my mind. I personally hate it when you want to print one section of a page and end up printing seven sheets of nonsense, six of which immediately get tossed. Couldn't there be a way of solving this problem and cutting down on wasted printouts? And

then it came to me: a mixture of print CSS; some good, solid, and semantic markup; and some unobtrusive JavaScript might just achieve what he was after. So I got started on a proof of concept.

The basic idea

Knowing that it was possible to dynamically change the appearance of a web page as viewed on screen using the DOM, JavaScript, and CSS, I wondered whether the same theory could apply to print. There was nothing to suggest that this could not happen. I recalled reading Aaron Gustafson's article in A List Apart (<http://www.alistapart.com>), where he demonstrated a method for displaying footnotes, and I knew that there was some scope for this kind of behavior. I just wasn't sure just how "dynamic" this could be. The only one way to find out was to try it. So, I began with this idea:

For any given page that has specific sections that might need to be printed in isolation, such as the interest rates page that prompted my investigation, use JavaScript to dynamically switch the display property off or on (that's to say block or none), but *only* for the print view.

The other aims for this technique included

- The technique must be unobtrusive.² It should fit in the category of progressive enhancement,³ and all the behavior should be controlled from a shared JavaScript file.
- It must look good, not just work.
- It has to be based on totally clean and valid markup.
- The purpose of the technique needs to be really obvious to the user.

Thankfully, all of these aims are achievable, as I will demonstrate in the following pages.

-
1. Greasemonkey is a Firefox extension that allows you to bolt on extra functionality to the browser. For more information, a good starting point is Dive Into Greasemonkey (<http://diveintogreasemonkey.org/install/what-is-greasemonkey.html>).
 2. Unobtrusive scripts do not require inline event handlers, such as onclick or onmouseover, in the HTML. All scripting is centralized, making site-wide management and maintenance far easier (in a similar way to how using a shared style sheet centralizes the management of the site's presentation).
 3. Progressive enhancement is an approach to web design that provides access to a basic version of your page to all, but adds an extra layer of behavior or styling (or both) to browsers that are up to the job. You can find out more about this technique and links to related articles at http://en.wikipedia.org/wiki/Progressive_Enhancement.

Preparing the foundations

With a clear final objective and some aims defined, I began work on a simple proof of concept. This is the part of the process where the visual effect looks a bit ugly but the functional parts get worked out, so don't worry—I'll be making it look pretty later in the chapter. I knew what the ultimate aim was (to apply it to the long-winded interest rates page that needs carving up for print), but to begin with, I created a dummy page that contained the basics.

Sectioning the page

The key to this technique is knowing which parts of the page you do and don't want to print. As I saw it, this could be done using CSS classes, and the obvious choice is to use a classname of section or, if you want to make it absolutely clear of the purpose, a classname like `print_section`.⁴ So, here's a simple HTML page with some sections marked up (with just a sprinkling of CSS applied to help identify the boundaries):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>A sectioned-up web page</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<style type="text/css">
.print_section
{
    border:1px solid gray;
    background:#e1e1e1;
    margin:0 0 15px 0;
    padding:10px;
}
</style>
</head>

<body>
<h1>Sectioned up page</h1>

<div class="print_section">
<h2>Section 1</h2>
<p>This is section 1. Nothing much to see here, just a bunch of meaningless words.</p>
</div>

<div class="print_section">
<h2>Section 2</h2>
<p>This is section 2. You'd be better off watching paint dry than reading the content here.</p>
</div>
```

4. If we were using XHTML 2.0, we could opt for using a section element. See www.w3.org/TR/xhtml2/mod-structural.html#edef_structural_section for more information.

```

<div class="print_section">
  <h2>Section 3</h2>
  <p>This is the final section. Hurrah!</p>
</div>

</body>
</html>

```

Figure 8-1 shows this dummy page.

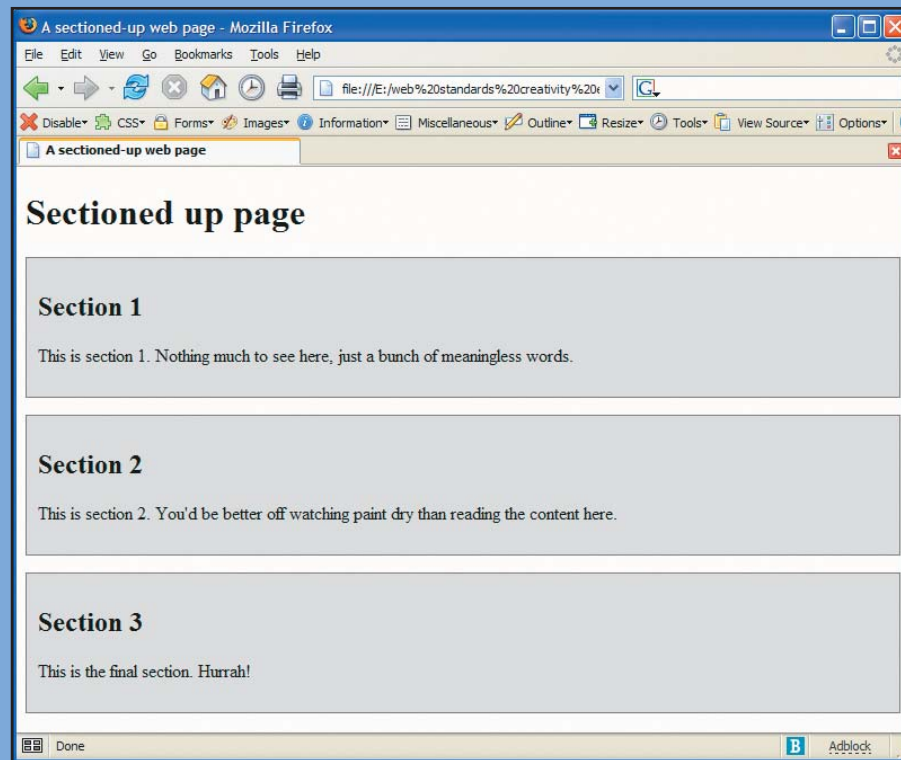


Figure 8-1. Sectioning the page, ready for printing

This would be the basic page (ignoring the placeholder content that I've used for the dummy page) that a visitor would see. Now the trick is to add in the extra layers of goodness on top for those browsers that are up to the job—progressive enhancement in action.

Identifying the sections

The basic structure of the page is there, but it needs something else if this technique is to work. How are we to magically toggle the appearance of these sections if we can't identify which section we're dealing with at any given time? The clue is in that question: *identify*, or *id*, the sections. Yep, we just have to add an *id* attribute to the sections, like so:

```
<div id="sect1" class="print_section">
  <h2>Section 1</h2>
  <p>This is section 1. Nothing much to see here, just a bunch of ➡
    meaningless words.</p>
</div>

<div id="sect2" class="print_section">
  <h2>Section 2</h2>
  <p>This is section 2. You'd be better off watching paint dry than ➡
    reading the content here.</p>
</div>

<div id="sect3" class="print_section">
  <h2>Section 3</h2>
  <p>This is the final section. Hurrah!</p>
</div>
```

This will have no effect on the visual side of things for the time being, but it will provide the hook that we're going to need for our JavaScript behaviors. Now let's get to the nuts and bolts of it.

Pseudocode first

Earlier in the chapter, I described in plain English what the technique should do. Now I'm going to veer into the pseudocode world and describe the order of events that are going to take place here.

The browser begins loading the page, and includes a piece of JavaScript in an external .js file that contains all the behavioral rules. When the page has loaded, the JavaScript kicks in and does the following:

1. First, it goes through the entire document looking for *div* elements and creates an array
2. Next, it goes through that array of *div* elements one by one. If the *div* element has a classname of *print_section*, it does the following:
 - (a) It creates a button or link that will say *Print this section* or similar.
 - (b) It then applies a behavior to that button. The behavior when clicked is to first check the *id* of the link that was clicked. If the *id* matches the current section (as the browser trawls through all the printable sections in the document), make that section visible (in the printed view). If the *id* of the link clicked does not match up (as the browser trawls through all the printable sections in the document), it hides the current section (again, only for print).

Section 3

This is the final section. Hurrah!

3. Bring up the Print dialog box in the browser, as shown in Figure 8-2. Unfortunately, it's not possible to avoid this step and get it to go *straight* to print. The user will still have one more button to click. In the Firefox example in Figure 8-2, it's the OK button.

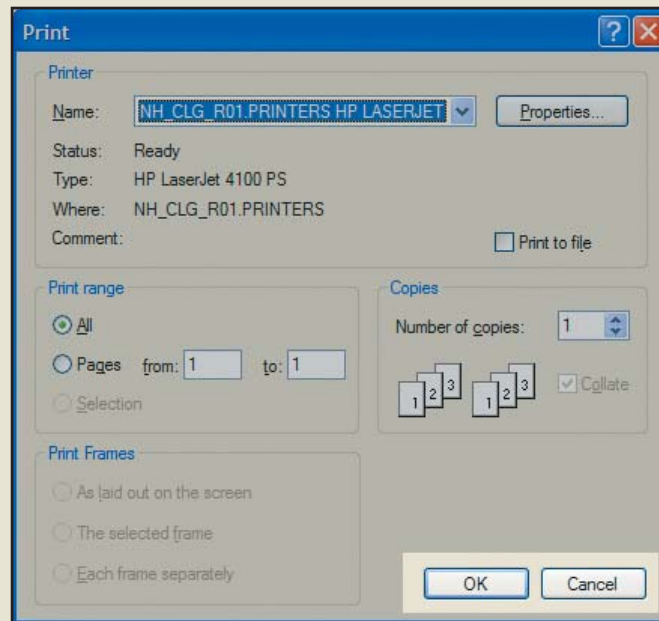


Figure 8-2. The Print dialog box appears before printing.

4. When the visual elements for each section are created and their associated behaviors are defined, they then get inserted into the page.

All of these steps should happen very quickly. Unless the web page is very long or there is an element on the web page that takes a lot longer to download than others (for example, an image from an external website that is running, like a dog, and not a greyhound at that), the user should not notice a jarring change in the page's appearance.

The reason why a single slow-loading element on a page can delay this effect is because the JavaScript runs after the page has loaded. This is because the onload event happens once everything on the page has been downloaded—all images, scripts, and CSS files linked to—rather than just the page's markup.

So, that's the plan of attack with the dummy page. Let's look at some of the real code that's going to help us achieve this.

Event planning

The first thing that needs doing is setting up the page load events. Now there are many different ways of doing this, and just as many opinions about which method is the best (see <http://tinyurl.com/ct2vy>). My bet is that many of the readers of this book are at a level where they are quite capable of making up their own mind and can adapt the suggested `addEventListener` function that I'm using (from Scott Andrew: <http://tinyurl.com/qcmdrd>). However, even if this is not the case, and you, dear reader, have no idea what I'm talking about, fear not—it really is worrying too much about the smallest of details. Move along, nothing to see here, nothing to see . . .

The `addEventListener` function primes the web page. It tells the browser what it needs to run (which JavaScript function) and when (on what event—for example, when the page loads or when something gets clicked). Here it is:

```
function addEvent(elm, evType, fn, useCapture)
{
  if(elm.addEventListener)
  {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  }
  else if (elm.attachEvent)
  {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  }
  else
  {
    elm['on' + evType] = fn;
  }
}
```

Now if that looks like gobbledygook to you, don't worry. You don't need to understand everything it does or how. You simply need to know that it has been created to work around some cross-browser JavaScript issues, and it allows you to call it using three parameters:

- What element you are dealing with (`elm`)
- What the event is (click, load, focus) (`evType`)
- What JavaScript function you want it to do when that element has that event triggered (`fn`)

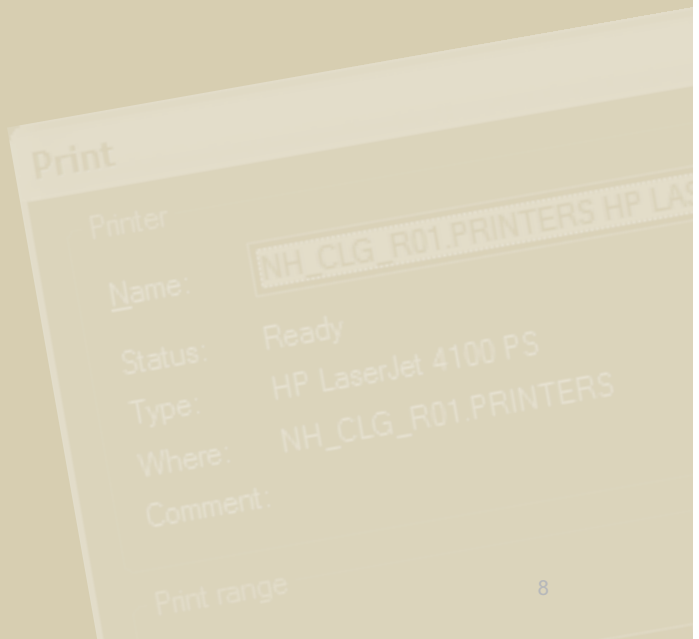
And here's an example of how you would call it:

```
addEvent(window, 'load', addPrintLinks, false);
```

Or, in plain English, when the window (or, to be precise, the document in this window) has loaded, please run the `addPrintLinks` function. This function has yet to be built, so just to test that it works, I've placed a simple alert there for the time being:

```
<script type="text/javascript">
  function addPrintLinks()
  {
    alert("Here's where the cool stuff will happen");
  }
  addEvent(window, 'load', addPrintLinks, false);
</script>
```

The alert looks like Figure 8-3.



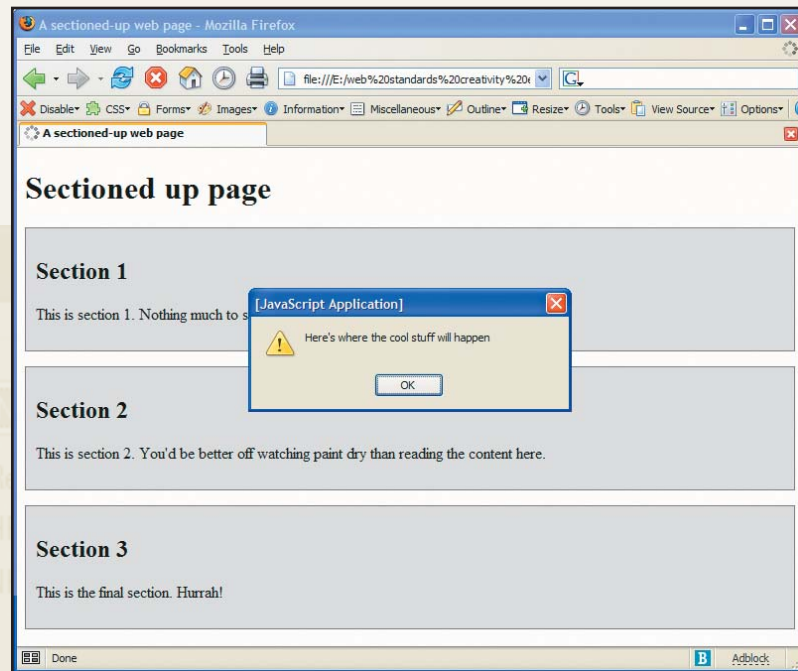


Figure 8-3. The alert placeholder in action

From pseudocode to real code

Having set out the steps required to make this work, let's take a look at the actual code that will do it (I'll quote from my earlier descriptions):

"The browser begins loading the page, and includes a piece of JavaScript in an external .js file that contains all the behavioral rules."

In the previous examples, I showed the scripts embedded in the page itself. It's now time to place the scripts in their own file and refer to them in the web page, like so:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>A sectioned-up page</title>
  <script type="text/javascript" src="print_sections.js"></script>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  ...
```

"When the page has loaded . . ."

That's our AddEvent part, mentioned previously. The addEvent function should be in the .js file, as is the following call to that function:

```
addEvent(window, 'load', addPrintLinks, false);
```

“... the JavaScript kicks in and does the following:

First, it goes through the entire document looking for div elements and creates an array.”

```
function addPrintLinks()
{
  var el = document.getElementsByTagName("div");
  for (i=0;i<el.length;i++)
    { //loop through the array of divs and do something cool
      ...
    }
}
```

Creating new print links for the page

“Next, it goes through that array of div elements one by one. If the div element has a classname of print_section...”

```
function addPrintLinks()
{
  var el = document.getElementsByTagName("div");
  for (i=0;i<el.length;i++)
    {
      if (el[i].className=="print_section")
        { //focus on only divs that are print sections
          ...
        }
    }
}
```

“... it does the following:

It creates a button or link that will say Print this section or similar.

It then applies a behavior to that button.”

Quite a bit of JavaScript is required to fulfill these aims, and here it is (with comments in the appropriate places explaining what each part is doing):

```
function addPrintLinks()
{
  var el = document.getElementsByTagName("div");
  for (i=0;i<el.length;i++)
    {
      if (el[i].className=="print_section")
        {
          // create the anchor element
          var newLink = document.createElement("a");
          // give it some text content
          var newLinkText = document.createTextNode("print this section only");
```

```

// create a container for the link to
// go in and give it a class
var newLinkPara = document.createElement("p");
newLinkPara.setAttribute("class", "printbutton");

// set up the 'print this section' link

newLink.setAttribute("href", "#");
// the print button will need a unique ID
// (this will be used to tell the function
// what section is to be shown or hidden )
var btId = "printbut_" + el[i].id;
newLink.setAttribute("id", btId);
// add the text for the link to the anchor element
newLink.appendChild(newLinkText);
// add the anchor element to the paragraph element
newLinkPara.appendChild(newLink);

//add the behaviors for the new link
newLink.onclick = togglePrintDisplay;
newLink.onkeypress = togglePrintDisplay;
}
}
}

```

At this point, we've created the link that will be used to do the printout. It's been placed in a paragraph, but that paragraph has not yet been inserted into the document. It's currently in limbo—floating around in the browser's memory somewhere—but we'll be getting to that in just a moment. Carrying on with the pseudo-to-real code translations then:

“The behavior [of the link] when clicked is to first check the `id` of the link that was clicked. If the `id` matches the current section (as the browser trawls through all the printable sections in the document), make that section visible (in the printed view). If the `id` of the link clicked does *not* match up (as the browser trawls through all the printable sections in the document), it hides the current section (again, only for print).

Bring up the Print dialog box in the browser:”

The behavior for the link was attached using these lines of script:

```

newLink.onclick = togglePrintDisplay;
newLink.onkeypress = togglePrintDisplay;

```

This means that when the link is clicked or when the link has focus and a keypress event is detected, the browser should run a new function called `togglePrintDisplay`.

Adding the behavior

Let's take a look at what's in the `togglePrintDisplay` function. I'll show it all in one go first, and then I'll step through the constituent parts to explain what's happening:

```

function togglePrintDisplay(e)
{
var whatSection = this.id.split("_");
whatSection = whatSection[1];
var el = document.getElementsByTagName("div");

```

```

for (i=0;i<el.length;i++)
{
  if (el[i].className.indexOf("print_section")!= -1)
  {
    el[i].removeAttribute("className");
    if (el[i].id==whatSection)
    {
      //show only this section for print
      el[i].setAttribute("className","print_section print");
      el[i].setAttribute("class","print_section print");
    }
    else
    {
      //hide the sections from printout
      el[i].setAttribute("className","print_section noprint");
      el[i].setAttribute("class","print_section noprint");
    }
  }
}
if (window.event)
{
  window.event.returnValue = false;
  window.event.cancelBubble = true;
}
else if (e)
{
  e.stopPropagation();
  e.preventDefault();
}
window.print();
}

```

And here it is step by step. First, the function gets the `id` from the link that was clicked. The `id` should be something like `printBut_sect2`. It was created programmatically (grabbing the parent container's `id` of `sect2` and appending it to the newly created button) earlier, like so:

```
var btId = "printbut_" + el[i].id;
```

Of course, it's possible to use shorter variable names, but I'm aiming for readability here. Note that you cannot begin an `id` with a number; hence, I used an underscore in the `id` value. It makes it pretty easy to retrieve the value later, as shown in the following snippet. We use a `split` function, splitting at the underscore. This creates an array with two values: `printBut` and the part we really want, `sect2` in the following example.

Those two values are passed into a variable `whatSection`. We really want only the second part of that array, though. Because arrays start at 0, the second item in the array is referred to using `[1]`; hence, we get the information we want using `whatSection[1]`. That value is then passed into the `whatSection` variable (overwriting its previous array values; `whatSection` now ceases to be an array and becomes a string).

```

function togglePrintDisplay(e)
{
  var whatSection = this.id.split("_");
  whatSection = whatSection[1];
}

```

Now that we have the id we're looking for, we need to once again trawl through the document, looking for all the div elements, specifically the div elements that have a classname of `print_section`:

```
var el = document.getElementsByTagName("div");
for (i=0;i<el.length;i++) //loop through all the divs found
{
    if (el[i].className.indexOf("print_section")!=-1)
    {
```

So, at this stage, we're looking at a div that has in its classname somewhere the value `print_section`. Note that there is a reason why I've chosen to use the `indexOf` method in the preceding bit of script to match the `className` attribute. In the next part, I'll combine the `print_section` classname with either a `print` or `noprint` classname, so a straight `==` comparison would not work. Onwards and upwards then.

First of all, we get rid of any `className` attributes found for the element that's currently being looked at (remember, we're looping through a collection), and then check the id we ascertained earlier. If the id matches up (for example, `sect1`, `sect2`), then we know that the button clicked relates to this div, and we can deal with it accordingly by setting a class of `print`, as well as reinstating the `print_section` classname (otherwise, this script would fail after its first use for any given page):

```
    el[i].removeAttribute("className");
    if (el[i].id==whatSection)
        { //if this is the section to be printed, set a 'print' classname
          el[i].setAttribute("className","print_section print");
          el[i].setAttribute("class","print_section print");
        }
    else
        {
            // otherwise hide the section from printout
            el[i].setAttribute("className","print_section noprint");
            el[i].setAttribute("class","print_section noprint");
        }
    }
}
```

Note that there seems to be a bit of doubling of effort here: setting both the `class` attribute and the `className` attribute. This is a workaround to solve differences between the way Internet Explorer and other browsers handle things. It's not a *big* overhead, and hardly worth creating a new custom function to deal with it (which you could do if you felt so inclined).

The final part of the script is simply there to cancel the browser carrying out the default action of the element clicked. In the example here, the element I've chosen to fire up the printing functions is a link (or anchor, to be precise), which I gave an `href` attribute of `#`. Normally, a link with this attribute would call the same page again. If this happened in our page, when choosing the print button, the page would reload and appear to jump, which is not the desired effect. The following code stops the default action from taking place:

```
if (window.event)
{
    window.event.returnValue = false;
    window.event.cancelBubble = true;
}
else if (e)
{
    e.stopPropagation();
    e.preventDefault();
}
```

It is generally considered bad practice to use a link, stick an href of # in there, and then use the link for another purpose. So why the heck am I doing it? How can I justify these actions? Well, the reason that it's normally a bad idea is that a link written this way fails when JavaScript is switched off. However, we're using JavaScript to create this technique; therefore, there's absolutely no danger of this situation happening.

Another solution might be to dynamically write in buttons rather than a link. However, that itself presents some issues. First of all, this may limit your styling options (you have more CSS control over a link element than an input of type button or submit). Secondly, well, it's not really a form is it? And if you dynamically write in form buttons, you need to wrap a form element around it. Also, if it is a form, should you use an input of type button or submit? No form data is being submitted for processing, after all. So, I opted for an anchor. Somebody shoot me!

Finally, we call the Print dialog box. All the classnames have been changed—thus affecting their display—and so we're pretty much good to go:

```
window.print();
} //end of the togglePrintDisplay function
```

Writing the new links into the document

We're not quite done yet, though. The `togglePrintDisplay` function was called from the `addPrintLinks` function, and that first script wasn't quite completed yet. There was the small matter of inserting the buttons (contained in paragraphs) into the web page after it loaded. That's achieved with this line:

```
el[i].insertBefore(newLinkPara,el[i].firstChild);
```

Recap: what these scripts do

And now we're pretty much done. So to recap:

1. The page loads, and `addPrintLinks` runs.
2. `addPrintLinks` calls another function, `togglePrintDisplay`, where the behaviors are set.
3. When the behavior is set for each element required, `addPrintLinks` resumes, appending the new links into the web page.

Here are the finished scripts in their entirety:

```
function addEvent(elm, evType, fn, useCapture)
{
  if(elm.addEventListener)
  {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  }
  else if (elm.attachEvent)
  {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  }
  else
```

```

        {
            elm['on' + evType] = fn;
        }
    }

function addPrintLinks()
{
    var el = document.getElementsByTagName("div");
    for (i=0;i<el.length;i++)
    {
        if (el[i].className=="print_section")
        {
            var newLink = document.createElement("a");
            var newLinkText = document.createTextNode("print this section ➡
                only");
            var newLinkPara = document.createElement("p");
            newLinkPara.setAttribute("class","printbutton");

            //set up the 'print this section' link
            var btId = "printbut_" + el[i].id;
            newLink.setAttribute("id",btId);
            newLink.appendChild(newLinkText);
            newLink.setAttribute("href","#");
            newLinkPara.appendChild(newLink);

            //add the behaviors for the new link
            newLink.onclick = togglePrintDisplay;
            newLink.onkeypress = togglePrintDisplay;

            //insert the para and the two links into the DOM
            el[i].insertBefore(newLinkPara,el[i].firstChild);
        }
    }
}

function togglePrintDisplay(e)
{
    var whatSection = this.id.split("_");
    whatSection = whatSection[1];
    var el = document.getElementsByTagName("div");
    for (i=0;i<el.length;i++)
    {
        if (el[i].className.indexOf("print_section")!=-1)
        {
            el[i].removeAttribute("className");
            if (el[i].id==whatSection)
            {
                //show only this section for print
                el[i].setAttribute("className","print_section print");
                el[i].setAttribute("class","print_section print");
            }
        }
        else
        {

```

```

        //hide the sections from printout
        el[i].setAttribute("className","print_section noprint");
        el[i].setAttribute("class","print_section noprint");
    }
}
}
if (window.event)
{
    window.event.returnValue = false;
    window.event.cancelBubble = true;
}
else if (e)
{
    e.stopPropagation();
    e.preventDefault();
}
}
window.print();
}
addEvent(window, 'load', addPrintLinks, false);

```

What about the CSS?

Wow, with all the talk of scripting, you might have forgotten that there's another very important aspect to this, and that's the print CSS styles. The scripts change the classnames of the various elements in the document, but somewhere we need to define the appearance of these items. The following CSS needs to be added to the document. The first part is purely cosmetic (for the link—to make it appear “buttony”), but the important part is highlighted in bold:

```

.print_section p.printbutton
{
    float:left;
}
.print_section p.printbutton a
{
    text-decoration:none;
    background:white;
    display:block;
    float:left;
    margin:3px;
    padding:10px;
    border:1px solid red;
}
@media print
{
    .noprint, .printbutton
    {
        display:none;
    }
    .print
    {
        display:block;
    }
}

```


The styles are limited to the print display by being contained inside the curly brackets after @media print. The toggling of display styles cannot affect the screen view (or other media types) because of this.

With all of this stitched together, the finished result looks like Figure 8-4.

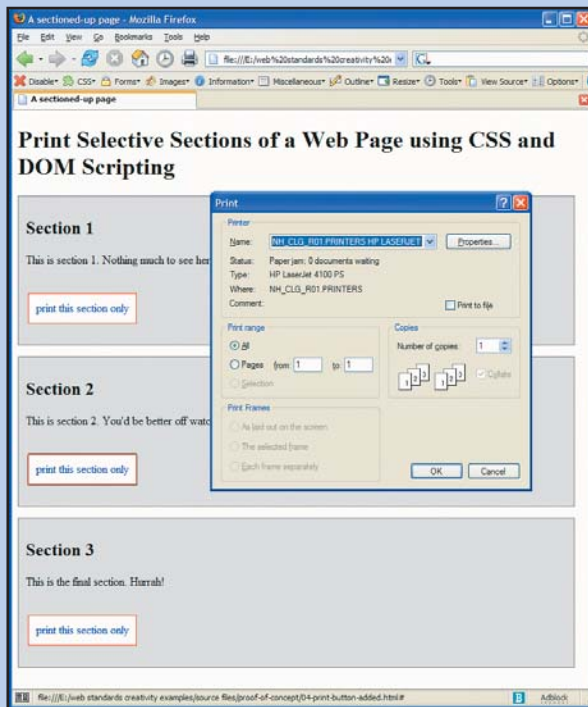


Figure 8-4. The finished prototype page

If the OK button were clicked, just one section of the page would be printed. In this example, it's section 2, as the print preview facility in Firefox demonstrates (see Figure 8-5).

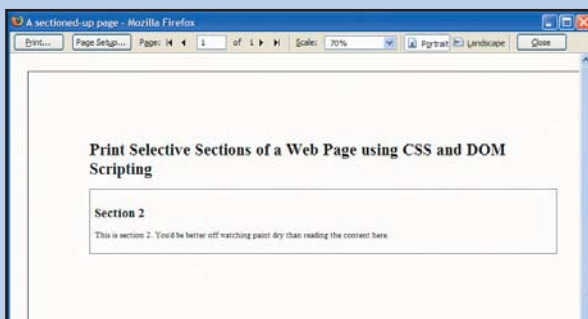


Figure 8-5. The functionality to print one section is working!

A couple of refinements

Before we move out of the proof-of-concept phase into practical application, there is a small issue to address. Consider this scenario:

What would happen if the user clicked a Print this section link but then hit Cancel. Perhaps the user wants to print the whole page after all?

If you check when the various events are triggered and what they do, you'll note there's a problem. By clicking the browser's print button after first activating one of the print section buttons, only the partial section will print. This is because the CSS displays have already been set, and the browser's print button will not affect this.

One solution is to write out two links: one that says print this section and another that says print the whole page. The idea is to stop people from veering up to their browser print button if they've just used the print this section button. Alternatively, you could create another function that reset the displays of each section to the default when a certain predictable event took place. For example, this could happen when the mouse pointer passes over a page header area, which it almost certainly would do if it were moving from a part of the page up to the top of the browser, as shown in Figure 8-6.

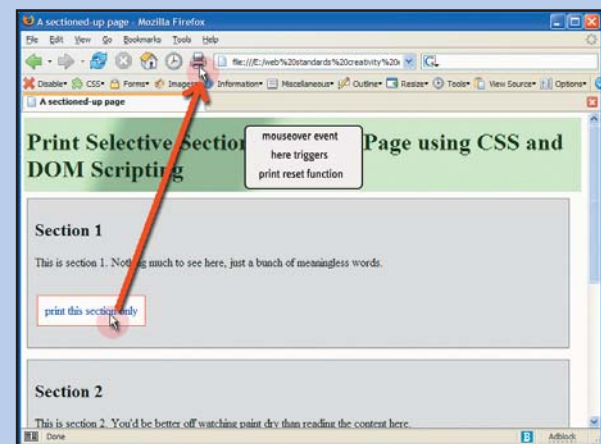


Figure 8-6. Resetting the display using a mouseover event that detects if the user gravitates towards the browser's print button

The following is the code required for my first solution (the simpler one of adding a second print button for each section). It's in the `addPrintLinks` function (additions shown in bold):

```
var btId = "printbut_" + el[i].id;
newLink.setAttribute("id",btId);
newLink.appendChild(newLinkText);
newLink.setAttribute("href","#");
newLinkPara.appendChild(newLink);

//set up the print all link
newLink2.setAttribute("href","#");
var bt2Id = "printall_" + el[i].id;
newLink2.setAttribute("id",bt2Id);
newLink2.appendChild(newLinkText2);
newLink2.setAttribute("href","#");
newLinkPara.appendChild(newLink2);

//add the behaviors for the new links
newLink.onclick = togglePrintDisplay;
newLink.onkeypress = togglePrintDisplay;
newLink2.onclick = printAll;
newLink2.onkeypress = printAll;
```

Note that the new links will call a new function named `printAll`. This is a variation of the `togglePrintDisplay` function (I copied, pasted, and amended it accordingly). This function simply runs through all the printable sections and resets the display so that they are visible for print purposes:

```
function printAll(e)
{
  var el = document.getElementsByTagName("div");
  for (i=0;i<el.length;i++)
  {
    if (el[i].className.indexOf("print_section")!=-1)
    {
      el[i].setAttribute("className","print_section print");
      el[i].setAttribute("class","print_section print");
    }
  }
  if (window.event)
  {
    window.event.returnValue = false;
    window.event.cancelBubble = true;
  }
  else if (e)
  {
    e.stopPropagation();
    e.preventDefault();
  }
  window.print();
}
```

It wouldn't take too much to abstract the original function (`togglePrintDisplay`) so that it could handle both the print some and print all scenarios, rather than having two separate functions. However, I've opted to keep them separate for now, mainly for clarity of reading.

Sliding in the code

In the proof-of-concept example, I placed all the functions in one single .js file. This included the addEvent function. If you are likely to use the addEvent function for more than one purpose across the site, it's probably better to split the .js file into two:

- A file containing the addEvent function (which can be used anywhere on the site for whatever purpose)
- Another file containing the print-handling functions (as it needs to be called only on pages that get this printing treatment)

So that's what I did for the practical application (the interest rates page in question can be found at www.nationwide.co.uk/savings/rates.htm (but please don't study the rest of the markup too closely, you might upset yourself—I don't have total control over what goes live, honest!). Here are the links to the JavaScript in the head of the document:

```
<script src="/_common_scripts/add_event.js" type="text/javascript">
</script>
<script src="/_common_scripts/print_sections_handler.js" type="text/javascript"></script>
</head>
```

That's the behavior added in. Now what's needed is to add in the extra markup around the relevant sections on the page as it currently stands. Here's an example using one of the smallest sections (just two table rows) with the new markup that we need for the print functions to work:

```
<div id="sect4" class="print_section">
<table cellpadding="4" cellspacing="0" width="100%">
<thead>
<tr>
<th>accounts for over 65s</th>
<th>interest tier</th>
<th>AER %</th>
<th>AER* %</th>
<th>gross p.a. %</th>
<th>net p.a. %</th>
<th>&nbsp;</th>
</tr>
</thead>
<tbody>
<tr>
<td class="textbottom" valign="top">
<a href="/pdf/P8865_Sep05.pdf" target="_blank">
Monthly Income 65+</a><br>passbook account
with a guaranteed regular income for the over 65's
<br><br><small>effective from:&nbsp;&nbsp;&nbsp;01/09/2005</small>
</td>
<td>1+</td>
<td>4.55</td>
<td>4.55</td>
<td>4.55</td>
<td>3.64</td>
<td><a href="#">more info</a></td>
</tr>
</tbody>
</table>
```

Product	Interest tier	AER %	AER* %	gross p.a. %	net p.a. %	More Info
online only accounts	1+	4.55	n/a	4.55	3.64	more info
e-Savings						
Instant access online savings account				4.65	3.72	more info
1 Year e-Bond (Annual Interest)	£1 - £3,000,000	4.65	n/a	4.65	3.64	more info
1 Year e-Bond (Monthly Interest)	£1 - £3,000,000	4.65	n/a	4.55	3.64	more info
2 Year e-Bond (Annual Interest)	£1 - £3,000,000	4.70	n/a	4.70	3.76	more info
2 Year e-Bond (Monthly Interest)	£1 - £3,000,000	4.70	n/a	4.60	3.68	more info
3 Year e-Bond (Annual Interest)	£1 - £3,000,000	4.70	n/a	4.70	3.76	more info
3 Year e-Bond (Monthly Interest)	£1 - £3,000,000	4.70	n/a	4.60	3.68	more info
Other accounts						
Instant access accounts	1+	1.20	4.45	4.45	3.56	more info
Bonus Saver						
passbook account for regular saving				3.85	3.08	more info
InvestDirect						
postal account with cash card	£1 - £24,999	3.95	n/a	3.95	3.16	more info
	£25,000 - £49,999	4.05	n/a	4.05	3.24	more info
CashBuilder						
Instant access account with passbook or cash card	£1 - £9,999	1.20	n/a	1.20	0.96	more info
	£10,000 - £24,999	1.40	n/a	1.40	1.12	more info
	£25,000 - £49,999	1.80	n/a	1.80	1.44	more info
	£50,000+	2.10	n/a	2.10	1.68	more info
ISAs						
Members' ISA Bond	£1 - £9,999	4.80	n/a	4.80	n/a	more info
Tax Free First Cash ISA rewarding long term members	£10,000+	4.85	n/a	4.85	n/a	more info
Instant Access ISA	£1 - £9,999	4.50	n/a	4.50	n/a	more info
Tax free instant access passbook account	£10,000+	4.55	n/a	4.55	n/a	more info
Flexi Maturity ISA Bond	£1 - £9,999	4.80	n/a	4.80	n/a	more info

```

        600,500);return false;">apply in branch</a><br>
        <a href="http://www.nationwide.co.uk/pdf/P8865_Sep05.pdf"
        target="_blank">view leaflet</a>
    </td>
</tr>
</tbody>
</table>
</div>

```

Each section on the page needs to be marked up in this way, with an id of sect#, where # is the section number.

The markup now has the necessary hooks for the script to grab and write in the print links. But how do these look? Well, a whole lot better than those shown in the proof of concept! Figure 8-9 shows a proposed design of the links in context.



Figure 8-9. The print link design

Predictably, the design was put together in Photoshop, but very much with an idea for how it can be built. In the example in Figure 8-9, the link buttons could be created using images (and the script would need to be adapted so that it writes out an `img` element rather than text). Figure 8-10 shows the two approaches: the first set of links is HTML text over a background image, and the second is based on `img` elements.



Figure 8-10. The two approaches to creating the link buttons

For the purposes of building on the proof-of-concept stage and doing a like-for-like translation, I am going to show how you do this using a combination of CSS background images and real text, rather than using `img` elements.

Important: The downside of this approach, I should warn you, is that when (or if) you scale text up on the page, the background image for the button will not scale accordingly, and text may break out of the button area, as shown in Figure 8-11. Writing out an `img` element gets around that problem, but I'll leave that to you to experiment with (it's really not a stretch to adapt the script to do this, honest!).



Figure 8-11. The effect of scaling up to maximum size in Internet Explorer when text sits on top of a button

Styling the print links

Regardless of the approach taken for the links, you'll need to apply some CSS. In the proof of concept, I used a paragraph, which contained the two links. This can be used to include a background image (the gray section in the images in Figure 8-10 that includes a printer), while the links themselves need the backgrounds. Figure 8-12 shows the constituent parts.

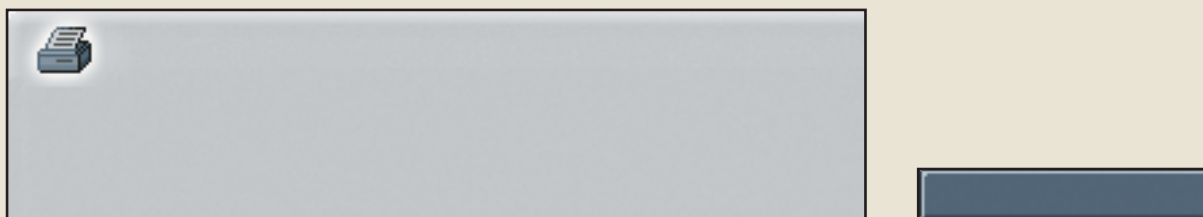


Figure 8-12. The background image and button used to style the print links

Before looking at the CSS, just to recap: the HTML for these buttons (once generated via the JavaScript and inserted into the DOM) is exactly as per the proof of concept. The bold markup shown here is the part that will be dynamically generated:

```
<div id="sect1">
  <p class="printbutton">
    <a href="#" id="printbut_sect1">print this section</a>
    <a href="#" id="printall_sect1">print whole page</a>
  </p>
  <table><!--table stuff goes here --></table>
</div>
```



To get it looking like the proposed design, the CSS needs to push the whole lot over to the right, apply the necessary backgrounds, and allow enough space to the left of the first link so the little printer icon is visible. Here's the CSS that lets us do this:

```
p.printbutton
{
    background-image:url(print-button-para-bg.gif);
    background-repeat:no-repeat;
    background-color:#576374;
    float:right;
    padding:6px 0 0 40px;
}
#sect1 p.printbutton a
{
    background-image:url(print-button-bg.gif);
    background-repeat:no-repeat;
    padding:2px 0;
    margin:0 10px 0 0;
    color:white;
    text-decoration:none;
    width:110px;
    height:19px;
    font-weight:bold;
    font-size:x-small;
    display:block;
    float:left;
    text-align:center;
}
```

Pulling it all together

The scripts have been inserted in the document head, the styles for the generated links have been defined, and the page has had all the necessary ids applied that will make the technique work. Figure 8-13 shows the finished result as seen on screen (in the Windows version of Firefox).



Never mind all that—what about saving the planet?

Ah, that! I said I would return to the topic, which is all about saving paper, avoiding needless printouts. With the piece of JavaScript wizardry you’ve seen in this chapter doing its thing, users can now print just the part of the page that is of interest to them, and here’s the proof. First, Figure 8-14 shows the results of a printout from the page before the script was applied—the “before.”

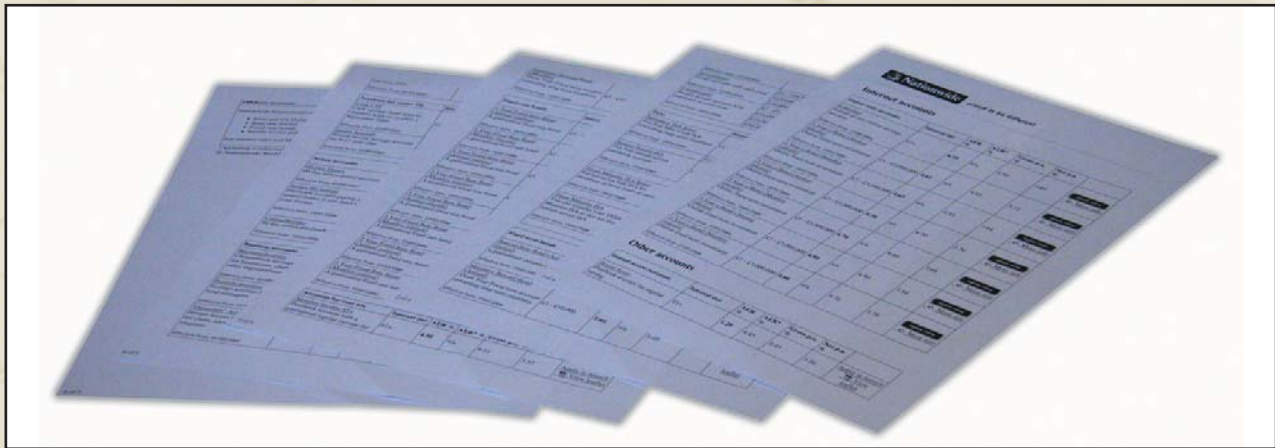


Figure 8-14. Phew—that’s a lot of paper!

And Figure 8-15 shows the results after applying the script—the “after.”

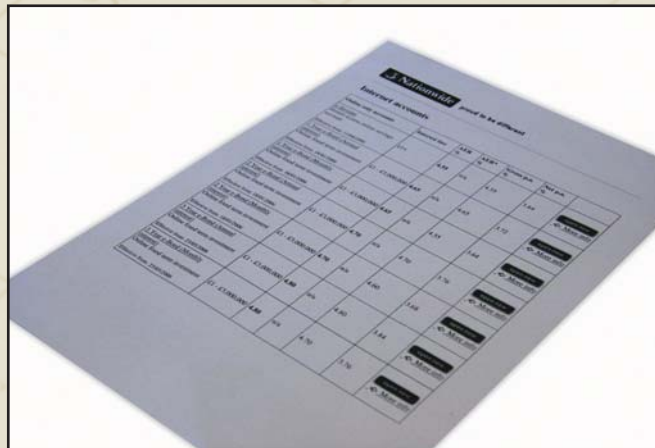


Figure 8-15. Much better!

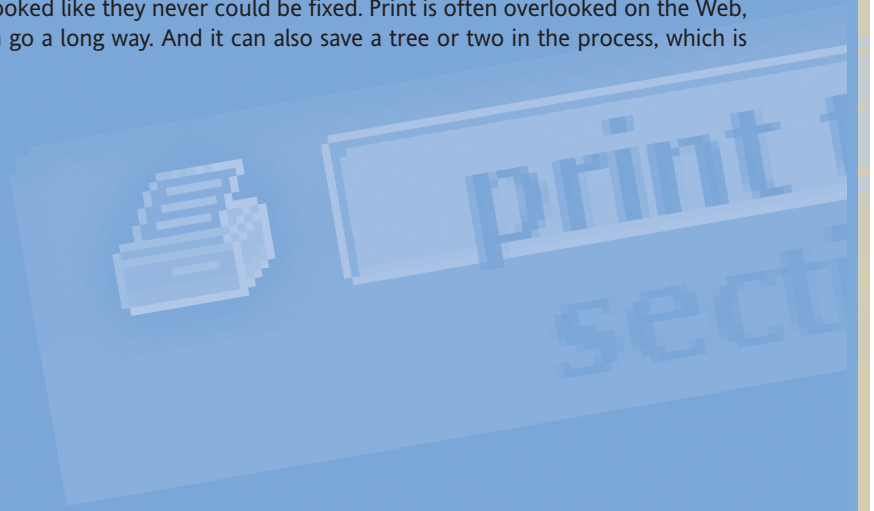
The eagle-eyed among you will notice that the controls that appear on the page to selectively print certain sections are not showing on the printout. Well, why would you need them on a printout? You can't click them or do anything on paper, so these dynamically generated controls are switched off in the print style sheet, like so:

```
@media print
{
  p.printbutton
  {
    display:none;
  }
}
```

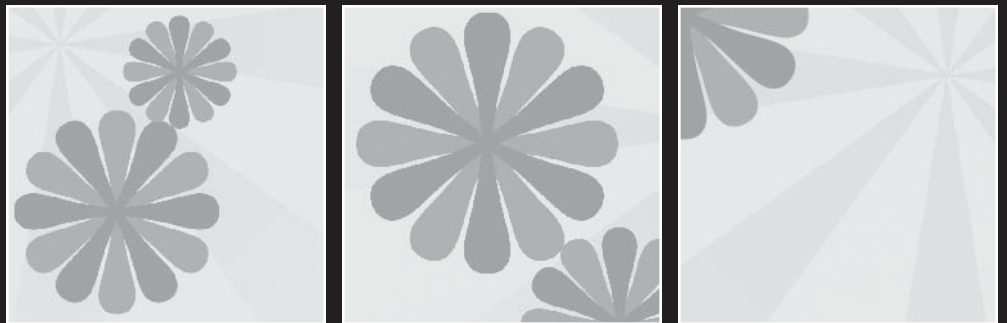
You could find many different uses for the technique, but essentially, it's handy wherever there's a large amount of content of which only a portion might sometimes need to be printed. The original posting on my personal site, where the technique was first put forward, is still up and available for comment (see <http://tinyurl.com/p9tqb>), including any suggestions that people have to improve the technique.

Conclusion

If there is one conclusion I would draw from this exercise, it's that it's all too easy to dismiss an idea on the basis that the browser can't do it. With the DOM, CSS, and JavaScript, you can custom-build functions to suit wildly inventive ideas and provide solutions to long-standing browser issues that looked like they never could be fixed. Print is often overlooked on the Web, but this need not be the case. Some fine-tuning can go a long way. And it can also save a tree or two in the process, which is all good.



3 THE DOCUMENT OBJECT MODEL



What this chapter covers:

- The concept of nodes
- Four very handy DOM methods: `getElementById`, `getElementsByTagName`, `getAttribute`, and `setAttribute`

It's time to meet the DOM. I'd like to introduce you to the Document Object Model and show you the world through its eyes.

D is for document

The Document Object Model can't work without a document. When you create a web page and load it in a web browser, the DOM stirs into life. It takes the document that you have written and turns it into an object.

In normal everyday English, the word “object” isn't very descriptive. It simply means thing. In programming languages, on the other hand, “object” has a very specific meaning.

Objects of desire

At the end of the last chapter, I showed you some examples of **objects** in JavaScript. You'll remember that objects are self-contained bundles of data. Variables associated with an object are called **properties** of the object, while functions that can be executed by an object are called **methods** of the object.

There are three kinds of objects in JavaScript:

- User-defined objects created from scratch by the programmer. We won't be dealing with these.
- Native objects like `Array`, `Math`, and `Date` that are built in to JavaScript.
- Host objects that are provided by the browser.

From the earliest days of JavaScript, some very important host objects have been made available for scripting. The most fundamental of these is the window object.

This object is nothing less than a representation of the browser window itself. The properties and methods of the window object are often referred to as the Browser Object Model, although perhaps Window Object Model would be more semantically correct. The Browser Object Model has methods like `window.open` and `window.blur`. These methods, incidentally, are responsible for all those annoying pop-up and pop-under windows that now plague the Web. No wonder JavaScript has a bad reputation!

Fortunately, we won't be dealing with the Browser Object Model very much. Instead, I'm going to focus on what's inside the browser window. The object that handles the contents of a web page is the document object.

For the rest of this book, we're going to be dealing almost exclusively with the properties and methods of the document object.

That explains the letter D (document) and the letter O (object) in DOM. But what about the letter M?

Dial M for model

The M in DOM stands for Model, but it could just as easily stand for Map. A model, like a map, is a representation of something. A model train represents a real train. A street map of a city represents the real city. The Document Object Model represents the web page that's currently loaded in the browser window. The browser provides a map (or model) of the page. You can use JavaScript to read this map.

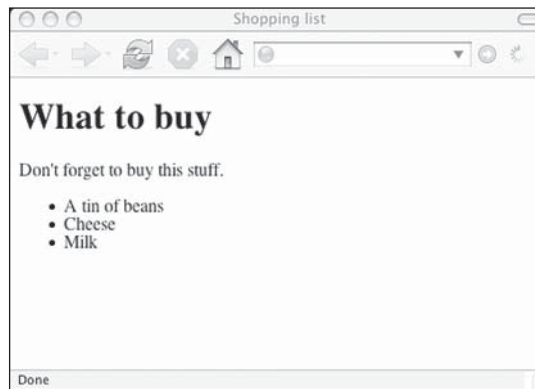
Maps make use of conventions like direction, contours, and scale. In order to read a map, you need to understand these conventions—and it's the same with the Document Object Model. In order to gain information from the model, you need to understand what conventions are being used to represent the document.

The most important convention used by the Document Object Model is the representation of a document as a tree. More specifically, the document is represented as a family tree.

A family tree is another example of a model. A family tree represents a real family, describes the relationships between family members, and uses conventions, like *parent*, *child*, and *sibling*. These can be used to represent some fairly complex relationships: one member of a family can be a parent to others, while also being the child of another family member, and the sibling of yet another family member.

The family tree model works just as well in representing a document written in (X)HTML.

Take a look at this very basic web page:



```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
➡ charset=utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li>Cheese</li>
      <li>Milk</li>
    </ul>
  </body>
</html>

```

This can be represented by the model in Figure 3-1.

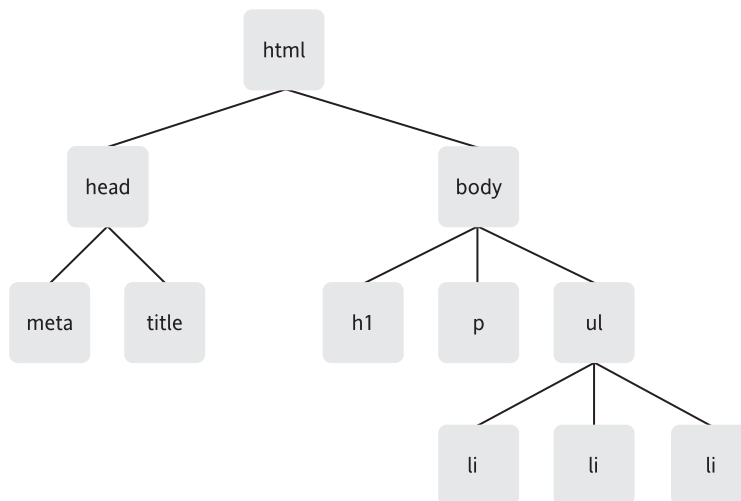


Figure 3-1. The element tree of a basic web page

Let's walk through the web page structure to see what it's made up of, and show why it's represented so well by the model shown previously. After the Doctype declaration, the document begins by opening an `<html>` tag. All the other elements of the web page are contained within this element, meaning it is a parent. Because all the other elements are inside, the `<html>` tag has no parent itself. It also has no siblings. If this were a tree, the `<html>` tag would be the root.

The root element is `html`. For all intents and purposes, the `html` element *is* the document.

If we move one level deeper, we find two branches: `<head>` and `<body>`. They exist side by side, which makes them siblings. They share the same parent, `<html>`, but they also both contain children, so they are parents themselves.

The `<head>` element has two children: `<meta>` and `<title>` (siblings of one another). The children of the `<body>` element are `<h1>`, `<p>`, and `` (all siblings of one another). If we drill down deeper still, we find that `` is also a parent. It has three children, all of them `` elements.

By using this simple convention of familial relationships, we can access lots of information about the relationship between elements.

For example, what is the relationship between `<h1>` and `<p>`? The answer is that they are siblings.

What is the relationship between `<body>` and ``? `<body>` is the parent of ``. `` is a child of `<body>`.

If you can think of the elements of a document in terms of a tree of familial relationships, then you're using the same terms as the DOM. However, instead of using the term “family tree,” it's more accurate to call a document a **node tree**.

Nodes

The term **node** comes from networking, where it is used to denote a point of connection in a network. A network is a collection of nodes.

In the real world, everything is made up of atoms. Atoms are the nodes of the real world. But atoms can themselves be broken down into smaller, subatomic particles. These subatomic particles are also considered nodes.

It's a similar situation with the Document Object Model. A document is a collection of nodes, with nodes as the branches and leaves on the document tree.

There are a number of different types of nodes. Just as atoms contain subatomic particles, some types of nodes contain other types of nodes.

element nodes

The DOM's equivalent of the atom is the **element node**.

When I described the structure of the shopping list document, I did so in terms of elements such as `<body>`, `<p>`, and ``. Elements are the basic building blocks of documents on the Web, and it's the arrangement of these elements in a document that gives the document its structure.

The tag provides the name of an element. Paragraph elements have the name `p`, unordered lists have the name `ul`, and list items have the name `li`.

Elements can contain other elements. All the list item elements in our document are contained within an unordered list element. In fact, the only element that isn't contained within another element is the `<html>` element. It's the root of our node tree.

text nodes

Elements are just one type of node. If a document consisted purely of empty elements, it would have a structure, but the document itself wouldn't contain much content. On the Web, where content is king, most content is provided by text.

In our example, the `<p>` element contains the text "Don't forget to buy this stuff." This is a **text node**.

In XHTML, text nodes are always enclosed within element nodes. But not all elements contain text nodes. In our shopping list document, the `` element doesn't contain any text directly. It contains other element nodes (the `` elements), and these contain text nodes.

attribute nodes

There are quite a few other types of nodes. Comments are a separate node type, for instance. But I'd just like to mention one more node type here.

Attributes are used to give more specific information about an element. The title attribute, for example, can be used on just about any element to specify exactly what the element contains:

```
<p title="a gentle reminder">Don't forget to buy this stuff.</p>
```

In the Document Object Model, `title="a gentle reminder"` is an **attribute node**, as shown in Figure 3-2. Because attributes are always placed within opening tags, attribute nodes are always contained within element nodes.

Not all elements contain attributes, but all attributes are contained by elements.

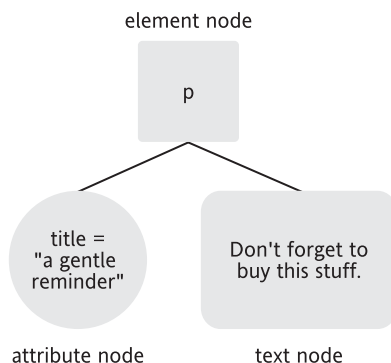


Figure 3-2. An element node that contains an attribute node and a text node

In our example document, you'll see that the unordered list (``) has been marked with the `id` attribute. You're probably familiar with the `id` and `class` attributes from using CSS. Just in case you haven't got that much familiarity with CSS, now we'll briefly recap the very basics of CSS.

Cascading Style Sheets

The DOM isn't the only technology that interacts with the structure of web pages. **Cascading Style Sheets** are used to instruct a browser how to display the contents of a document.

Like JavaScript, styles can be declared either in the `<head>` of a document (between `<style>` tags), or in an external style sheet. The syntax for styling an element with CSS is similar to that of JavaScript functions:

```
selector {
  property: value;
}
```

Style declarations can be used to specify the colors, fonts, and sizes used by the browser to display elements:

```
p {
  color: yellow;
  font-family: "arial", sans-serif;
  font-size: 1.2em;
}
```

Inheritance is a powerful feature of CSS. Like the DOM, CSS view the contents of a document as a node tree. Elements that are nested within the node tree will inherit the style properties of their parents.

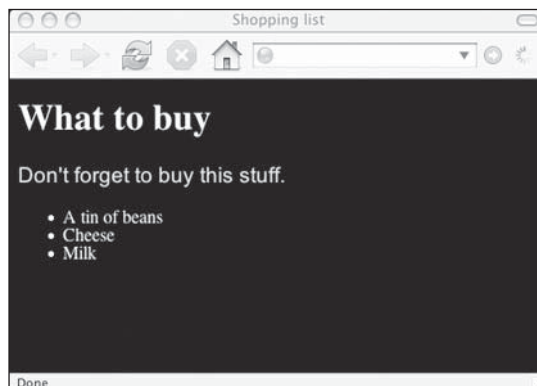
For instance, declaring colors or fonts on the `body` element will automatically apply those styles to all the elements contained within the body:

```
body {
  color: white;
  background-color: black;
}
```

Those colors will be applied not just to content contained directly by the `<body>` tag, but also by elements nested within the body.

The image at right is a basic web page with styles applied:

When you're applying styles to a document, there are times when you will want to target specific elements. You might want to make one paragraph a certain size and color, but leave other paragraphs unaffected. To get this level of precision, you'll need to insert something into the document itself to mark the paragraph as a special case.



To mark elements for special treatment, you can use one of two attributes: `class` or `id`.

class

The `class` attribute can be applied as often as you like to as many different elements as you like:

```
<p class="special">This paragraph has the special class</p>
<h2 class="special">So does this headline</h2>
```

In a style sheet, styles can then be applied to all the elements of this class:

```
.special {
    font-style: italic;
}
```

You can also target specific types of elements with this class:

```
h2.special {
    text-transform: uppercase;
}
```

id

The `id` attribute can be used once in a web page to uniquely identify an element:

```
<ul id="purchases">
```

In a style sheet, styles can then be applied specifically to this element:

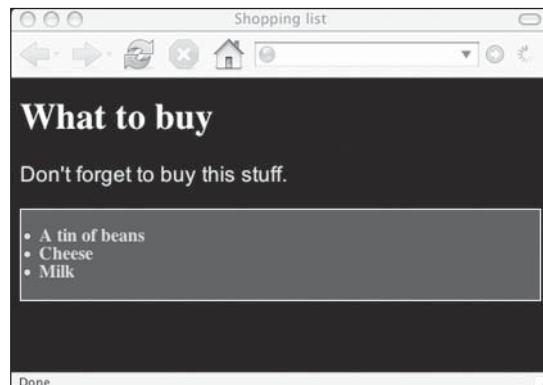
```
#purchases {
    border: 1px solid white;
    background-color: #333;
    color: #ccc;
    padding: 1em;
}
```

Although the `id` itself can only be applied once, a style sheet can use the `id` to apply styles to elements nested within the uniquely identified element:

```
#purchases li {
    font-weight: bold;
}
```

The image at right is an example of styles applied to a list with a unique `id`:

The `id` attribute acts as a kind of “hook” that can be targeted by CSS. The DOM can use the same hook.



getElementById

The DOM has a method called **getElementById**, which does exactly what it sounds like: it allows you to get straight to the element node with the specified id. Remember that JavaScript is case-sensitive so `getElementById` must always be written with case preserved. If you write **GetElementById** or **getElementbyid**, you won't get the results you expect.

This method is a function associated with the document object. Functions are always followed by parentheses that contain the function's arguments. `getElementById` takes just one argument: the id of the element you want to get to, contained in either single or double quotes.

```
document.getElementById(id)
```

Here's an example:

```
document.getElementById("purchases")
```

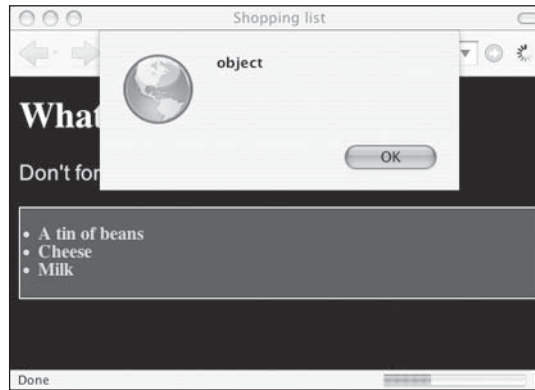
This is referencing the unique element that has been assigned the HTML id attribute "purchases" in the document object. This element also corresponds to an object. You can test this for yourself by using the `typeof` operator. This will tell you whether something is a string, a number, a function, a Boolean value, or an object.

I don't recommend this way of adding JavaScript to a document but, purely for testing purposes, insert this piece of JavaScript into the shopping list document. Put it right before the closing `</body>` tag:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
    ➡ charset=utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li>Cheese</li>
      <li>Milk</li>
    </ul>
    <script type="text/javascript">
      alert(typeof document.getElementById("purchases"));
    </script>
  </body>
</html>
```

When you load the XHTML file in a web browser, you will be greeted with an annoying pop-up box stating the nature of `document.getElementById("purchases")`. It is an object.

Here, an alert dialog reveals the nature of an element node:



In fact, every single element in a document is an object. Using the DOM, you can “get” at every single one of these elements.

Obviously you shouldn't give a unique id to every single element in a document. That would be overkill. Fortunately, the document object provides another method for getting at elements that don't have unique identifiers.

getElementsByTagName

If you use the method `getElementsByTagName`, you have instant access to an array populated with every occurrence of a specified tag. Like `getElementById`, this is a function that takes one argument. In this case, the argument is the name of a tag:

```
element.getElementsByTagName(tag)
```

It looks very similar to `getElementById`, but notice that this time you can get elements, plural. Be careful when you are writing your scripts that you don't inadvertently write `getElementById` or `getElementByTagName`.

Here it is in action:

```
document.getElementsByTagName("li")
```

This is an array populated with all the list items in the document object. Just as with any other array, you can use the `length` property to get the total number of elements.

Delete the alert statement you placed between `<script>` tags earlier and replace it with this:

```
alert(document.getElementsByTagName("li").length);
```

This will tell you the total number of list items in the document: three, in this case. Every value in the array is an object. You can test this yourself by looping through the array and using `typeof` on each value. For example, try this with a `for` loop:

```
for (var i=0; i < document.getElementsByTagName("li").length; i++) {
    alert(typeof document.getElementsByTagName("li")[i]);
}
```

Even if there is only one element with the specified tag name, `getElementsByTagName` still returns an array. The length of the array will simply be 1.

Now you will begin to notice it is starting to become tedious typing out `document.getElementsByTagName("li")` every time, and that the code is starting to get hard to read. You can reduce the amount of unnecessary typing and improve readability by assigning a variable to contain `document.getElementsByTagName("li")`.

Replace the `alert` statement between the `<script>` tags with these statements:

```
var items = document.getElementsByTagName("li");
for (var i=0; i < items.length; i++) {
    alert(typeof items[i]);
}
```

Now you'll get three annoying alert boxes, each one of them saying the same thing: `object`.

You can also use a wildcard with `getElementsByTagName`, which means you can make an array with every single element. The wildcard symbol (the asterisk) must be contained in quotes to distinguish it from the multiplication operator. The wildcard will give you the total number of element nodes in a document:

```
alert(document.getElementsByTagName("*").length);
```

You can also combine `getElementsByTagName` with `getElementById`. So far, we've only applied `getElementsByTagName` to the `document` object, but if you're interested in finding out how many list items are inside the element with the id "purchases", you could apply `getElementsByTagName` to that specific object:

```
var shopping = document.getElementById("purchases");
var items = shopping.getElementsByTagName("*");
```

Now the `items` array contains just the elements contained by the "purchases" list. In this case, that happens to be the same as the total number of the list items in the document:

```
alert (items.length);
```

And, if any further proof were needed, you can test that each one is an object:

```
for (var i=0; i < items.length; i++) {
    alert(typeof items[i]);
}
```

Taking stock

By now, you are probably well and truly fed up with seeing alert boxes containing the word “object”. I think I’ve made my point: every element node in a document is an object. Not only that, but every single one of these objects comes with an arsenal of methods, courtesy of the DOM. Using these pre-supplied methods, you can retrieve information about any element in a document. You can even alter the properties of an element.

Here’s a quick summary of what you’ve seen so far:

- A document is a tree of nodes.
- There are different types of nodes: elements, attributes, text, and so on.
- You can get straight to a specific element node using `getElementById`.
- You can get straight to a collection of element nodes using `getElementsByTagName`.
- Every one of these nodes is an object.

Now I want to show you some of the properties and methods associated with these objects.

getAttribute

So far, you’ve seen two different ways of getting to element nodes, using either `getElementById` or `getElementsByTagName`. Once you’ve got the element, you can find out the values of any of its attributes. You can do this with the **getAttribute** method.

`getAttribute` is a function. It takes only one argument—the attribute that you want to get:

```
object.getAttribute(attribute)
```

Unlike the other methods you’ve seen, you can’t use `getAttribute` on the document object. It can only be used on an element node object.

For example, you can use it in combination with `getElementsByTagName` to get the title attribute of every `<p>` element:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i < paras.length; i++ ) {
    alert(paras[i].getAttribute("title"));
}
```

If you include this code at the end of our shopping list document and then load the page in a web browser, you’ll be greeted with an alert box containing the text, “a gentle reminder”.

In our shopping list, there is only one `<p>` element and it has a title attribute. If there were more `<p>` elements and they didn’t have title attributes, then `getAttribute("title")` would return the value **null**. In JavaScript, null means that there is no value. You can test this for yourself by inserting this paragraph right after the existing paragraph:

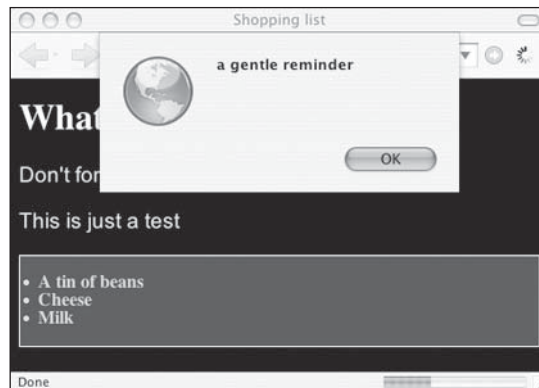
```
<p>This is just a test</p>
```

Now reload the page. This time, you'll see two alert boxes. The second one is either completely empty or simply says "null", depending on how the browser chooses to display null values.

We can modify our script so that it only pops up a message when a title attribute exists. We will add an if statement to check that the value returned by `getAttribute` is not null. While we're at it, let's use a few more variables to make the script easier to read:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
    var title_text = paras[i].getAttribute("title");
    if (title_text != null) {
        alert(title_text);
    }
}
```

Now if you reload the page, you will only see the alert box that contains the value, "a gentle reminder". Here is an example of this:



We can shorten the code even more. Whenever you want to check that something isn't null, you're really checking to see if it exists. A shorthand way of doing that is to use it as the condition in an if statement. `if (something)` is a shorthand way of writing `if (something != null)`. The condition of the if statement will be true if something exists. It will be false if something doesn't exist.

We can tighten up our code by simply writing `if (title_text)` instead of `if (title_text != null)`. While we're at it, we can put the alert statement on the same line as the if statement so that it reads more like English:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
    var title_text = paras[i].getAttribute("title");
    if (title_text) alert(title_text);
}
```

setAttribute

All of the methods you've seen so far have dealt with retrieving information. `setAttribute` is a bit different. It allows you to change the value of an attribute node.

Like `getAttribute`, this method is a function that only works on element nodes. However, `setAttribute` takes two arguments:

```
object.setAttribute(attribute,value)
```

In this example, I'm going to get the element with the id "purchases", and give it a title attribute with the value "a list of goods":

```
var shopping = document.getElementById("purchases");
shopping.setAttribute("title","a list of goods");
```

You can use `getAttribute` to test that the title attribute has been set:

```
var shopping = document.getElementById("purchases");
alert(shopping.getAttribute("title"));
shopping.setAttribute("title","a list of goods");
alert(shopping.getAttribute("title"));
```

Loading the page will now give you two alert boxes. The first one, which is executed before `setAttribute`, is empty or else displays "null". The second one, which is executed after the title attribute has been set, says "a list of goods".

In that example, we set an attribute where previously none had existed. The `setAttribute` method created the attribute and then set its value. If you use `setAttribute` on an element node that already has the specified attribute, the old value will be overwritten.

In our shopping list document, the `<p>` element already has a title attribute with the value "a gentle reminder". You use `setAttribute` to change this value:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
    var title_text = paras[i].getAttribute("title");
    if (title_text) {
        paras[i].setAttribute("title","brand new title text");
        alert(paras[i].getAttribute("title"));
    }
}
```

This will apply the value "brand new title text" to the title attribute of every `<p>` element in the document that already had a title attribute. In our shopping list document, the value "a gentle reminder" has been overwritten.

It's worth noting that, even when a document has been changed by `setAttribute`, you won't see that change reflected when you use the view source option in your web browser. This is because the DOM has dynamically updated the contents of the page after it has loaded. The real power of the DOM is that the contents of a page can be updated without refreshing the page in the browser.

What's next?

I've shown you four methods provided by the Document Object Model:

- `getElementById`
- `getElementsByTagName`
- `getAttribute`
- `setAttribute`

These four methods will be the cornerstones for many of the DOM scripts you're going to write.

The DOM offers many more methods and properties. There's `nodeName`, `nodeValue`, `childNodes`, `nextSibling`, and `parentNode`, to name just a few. But I'm not going to explain those just yet—I'll explain each one in turn as and when they're needed. I'm mentioning them now just to whet your appetite.

You've read through a lot of theory in this chapter. I hope by now you're itching to test the power of the DOM using something other than alert boxes. I think it's high time we applied the DOM to a case study.

Next, you're going to build a JavaScript image gallery using the four DOM methods introduced in this chapter.

A XHTML REFERENCE

This section of the reference guide details, in alphabetical order, generally supported elements and associated attributes. This is not intended as an exhaustive guide; rather, its aim is to list those elements important and relevant to current web design. Archaic deprecated elements such as `font` and `layer` are therefore ignored, as well as many attributes once associated with the `body` element, but the guide includes still occasionally useful deprecated and nonstandard elements and attributes such as `embed` and `target`.

Note that in the following pages, various styles are used for the attribute names and values. For the sake of clarity, quote marks have been omitted, but never forget that XHTML attributes must be quoted. Therefore, where you see the likes of id=name in this reference section, the final output would be id="name".

Standard attributes

Standard attributes are common to many elements. For brevity , they are listed in full here rather than in the XHTML element table later in the chapter . For each element in the forthcoming table, I simply state which groups of standard attributes are applicable to the element

Core attributes

Attribute	Description
class=classname	Specifies a CSS class to define the element’s visual appearance.
id=name	Defines a unique reference ID for the element.
style=style (deprecated)	Sets an inline style. Deprecated in XHTML 1.1, so it should be used sparingly and with caution.
title=string	Specifies the element’s title. Often used with links to provide a tooltip expanding on the link’s purpose or the target’s content.

Not valid in these elements: base, head, html, meta, param, script, style, and title.

Keyboard attributes

Attribute	Description
accesskey=character	Defines a keyboard shortcut to access an element. The short-cut must be a single character. Most commonly used with navigation links. See also Chapter 5, “Using accesskey and tabindex.”
tabindex=number	Defines the tab order of an element. Most commonly used with form input elements. Setting the value to 0 excludes the element from the tabbing order. The maximum value allowed is 32767. The tabindex values on a page needn’t be consecutive (for instance, you could use multiples of 10, to leave space for later additions). See also Chapter 5, “Using accesskey and tabindex.”

Language attributes

Attribute	Description
<code>dir=dir</code>	Specifies the text rendering direction: left-to-right (ltr, the default) or right-to-left (rtl).
<code>lang=language</code> (<i>deprecated</i>)	<p>Specifies the language for the tag's contents, using two-letter primary ISO639 codes and optional dialect codes. Included for backward compatibility with HTML. Used together with <code>xml:lang</code> (see below) in XHTML 1.0, but deprecated in XHTML 1.1.</p> <p>Examples:</p> <p><code>lang="en"</code> (English)</p> <p><code>lang="en-US"</code> (US English)</p> <p>ISO639 codes include the following: ar (Arabic), zh (Chinese), nl (Dutch), fr (French), de (German), el (Greek), he (Hebrew), it (Italian), ja (Japanese), pt (Portuguese), ru (Russian), sa (Sanskrit), es (Spanish), and ur (Urdu).</p>
<code>xml:lang=language</code>	Replaces lang in XHTML 1.1, but both should be used together in XHTML 1.0 to ensure backward compatibility with HTML and older browsers. <code>xml:lang</code> takes precedence over lang if set to a different value.

Not valid in these elements: base, br, frame, frameset, hr, iframe, param, and script.

Event attributes

A

As of HTML 4.0, it's been possible to trigger browser actions by way of HTML events. Again, these are listed in full here and referred to for the relevant elements of the XHTML element table. In XHTML, all event names must be in lowercase (e.g., onclick, not onClick).

Core events

Attribute	Description
<code>onclick=script</code>	Specifies a script to be run when the user clicks the element's content area
<code>ondblclick=script</code>	Specifies a script to be run when the user double-clicks the element's content area

continues

Attribute	Description
<code>onkeydown=script</code>	Specifies a script to be run when the user presses a key while the element's content area is focused
<code>onkeypress=script</code>	Specifies a script to be run when the user presses and releases a key while the element's content area is focused
<code>onkeyup=script</code>	Specifies a script to be run when the user releases a pressed key while the element's content area is focused
<code>onmousedown=script</code>	Specifies a script to be run when the user presses down the mouse button while the cursor is over the element's content area
<code>onmousemove=script</code>	Specifies a script to be run when the user moves the mouse cursor in the element's content area
<code>onmouseout=script</code>	Specifies a script to be run when the user moves the mouse cursor off the element's content area
<code>onmouseover=script</code>	Specifies a script to be run when the user moves the mouse cursor onto the element's content area
<code>onmouseup=script</code>	Specifies a script to be run when the user releases the mouse button on the element's content area

Not valid in these elements: base, bdo, br, frame, frameset, head, html, iframe, meta, param, script, style, and title.

Form element events

These events are generally restricted to form elements, although some other elements accept some of them.

Attribute	Description
<code>onblur=script</code>	Specifies a script to be run when the element loses focus
<code>onchange=script</code>	Specifies a script to be run when the element changes
<code>onfocus=script</code>	Specifies a script to be run when the element is focused

Attribute	Description
<code>onreset=script</code>	Specifies a script to be run when a form is reset
<code>onselect=script</code>	Specifies a script to be run when the element is selected
<code>onsubmit=script</code>	Specifies a script to be run when a form is submitted

Window events

These events are valid only in the following elements: `body` and `frameset`.

Attribute	Description
<code>onload=script</code>	Specifies a script to be run when the document loads
<code>onunload=script</code>	Specifies a script to be run when the document unloads

Although `onresize` is part of DOM2, it's not recognized by the XHTML specification. If an `onresize` event is required, it cannot be applied directly to the `body` element. Instead, you must declare it in the document head using `window.onresize=functionName`.

XHTML elements and attributes

The following pages list XHTML elements, associated attributes, and descriptions for all. Unless otherwise stated, assume an element is allowed in pages with XHTML Strict, XHTML Transitional, or XHTML Frameset DTDs. Do not use elements or attributes with DTDs that don't allow them. For instance, the `target` attribute cannot be used with XHTML Strict—doing so renders the page invalid.

Some elements are shown with a trailing forward slash. These are empty tags. Instead of having a start tag, content, and an end tag, these elements have a combined form. This takes the form of a start tag with an added trailing forward slash. Prior to the slash, a space is usually added. For instance, `
` denotes a line break.

Element	Attribute	Description	Standard attributes
<!-- ... -->		Defines a comment. See also Chapter 2, “Commenting your work.”	No attributes
<!DOCTYPE> (required)		Specifies a DTD for the document. This is required for a valid XHTML document. See also Chapter 2, “DOCTYPE declarations explained.”	No attributes
<a>		Defines an anchor. Can link to another document by using the href attribute, or create an anchor within a document by using the id or name attributes. Despite the number of available attributes, some aren't well supported. Generally, href, name, title, and target are commonly used, along with class and id for use as CSS or scripting hooks. See also Chapter 5, “Creating and styling web page links.”	Core attributes, keyboard attributes, language attributes Core events, onblur, onfocus
	href=URL	Defines the link target.	
	name=name (deprecated)	Names an anchor. Due to be replaced by id in future versions of XHTML. When defining a fragment identifier in XHTML 1.0, id must be used.	
	rel=relationship	Specifies the relationship from the current document to the target document. Common values include next, prev, parent, child, index, toc, and glossary. Also used within link elements to define the relationship of linked CSS documents (e.g., to establish default and alternative style sheets).	

Element	Attribute	Description	Standard attributes
	<code>rev=relationship</code>	Specifies the relationship from the target document to the current document. Common values include next, prev, parent, child, index, toc, and glossary.	
	<code>target=_blank _parent _self _top [name]</code> (deprecated)	Defines where the target URL opens. Primarily of use with frames, stating which frame a target should open in. Commonly used in web pages to open external links in a new window—a practice that should be avoided, because it breaks the browser history path. <i>Unavailable in XHTML 1.0, so cannot be used with XHTML 1.0 Strict documents. However, target is available in XHTML 1.1 using the target module.</i>	
	<code>type=MIME type</code>	Specifies the MIME type of the target. For instance, if linking to a plain text file, you might use the following: <code></code>	
<code><abbr></code>		Identifies the element content as an abbreviation. This can be useful for nonvisual web browsers. For example: <code><abbr title="Doctor">Dr.</abbr></code> See also Chapter 3, “Acronyms and abbreviations.”	Core attributes, language attributes Core events

A

continues

Element	Attribute	Description	Standard attributes
<acronym>		<p>Identifies the element content as an acronym. This can be useful for nonvisual web browsers. For example:</p> <pre><acronym title="North ➡ Atlantic Treaty ➡ Organization">NATO </acronym></pre> <p>See also Chapter 3, “Acronyms and abbreviations.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<address>		<p>Used to define addresses, signatures, or document authors. Typically rendered in italics, with a line break above and below (but no additional space).</p> <p>See also Chapter 8, “Contact details structure redux.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<applet> (deprecated)	<p><i>align=position</i></p> <p><i>alt=string</i></p> <p><i>archive=URL</i></p>	<p>Adds an applet to the web page. Deprecated in favor of the object element, but still required for embedding some Java applets. <i>This element cannot be used with an XHTML Strict DOCTYPE. Likewise, all of the element’s attributes are deprecated and cannot be used with the XHTML Strict DOCTYPE.</i></p> <p>Defines text alignment around the element. Possible values are left, right, top, middle, and bottom.</p> <p>Alternate text for browsers that don’t support applets.</p> <p>Defines a list of URLs with classes to be preloaded.</p>	<p>Core attributes, keyboard attributes, language attributes</p> <p>Core events</p>

Element	Attribute	Description	Standard attributes
	<code>code=URL</code> (required)	Specifies either the name of the class <i>file</i> that contains the applet's compiled applet subclass or the path to get the class <i>file</i> , including the class file itself. This attribute is required if the <code>object</code> attribute is missing, and vice versa. If both are present, they must use the same class name. Note: the value is case-sensitive.	
	<code>codebase=URL</code>	Base URL of the applet.	
	<code>height=number</code> (required)	Pixel height of the applet. This attribute is <i>required</i> .	
	<code>hspace=number</code>	Sets horizontal space around the applet.	
	<code>name=name</code>	Sets a unique name for this instance of the applet, which can be used in scripts.	
	<code>object=name</code>	Defines a resource's name that contains a serialized representation of the applet.	
	<code>vspace=number</code>	Sets vertical space around the applet.	
	<code>width=number</code> (required)	Pixel width of the applet. This attribute is <i>required</i> .	
<code><area /></code>		Defines a clickable area within a client-side image map. Should be nested within a <code>map</code> element (see separate <code><map></code> entry). See also Chapter 5, "Image maps."	Core attributes, keyboard attributes, language attributes Core events, <code>onblur</code> , <code>onfocus</code>
	<code>alt=string</code> (required)	Provides alternate text for nonvisual browsers. This attribute is <i>required</i> .	

A

continues

Element	Attribute	Description	Standard attributes
	<code>coords=</code> <i>coordinates list</i>	<p>Specifies coordinates for the clickable image map area. Values are defined as a comma-separated list. The number of values depends on the shape attribute value:</p> <p>For <code>rect</code>, four values are required, defining the coordinates on the x and y axes of the top-left and bottom-right corners.</p> <p>For <code>circle</code>, three values are required, with the first two defining the x and y coordinates of the hot-spot center, and the third defining the circle's radius.</p> <p>For <code>poly</code>, each pair of x and y values defines a point of the hot-spot's.</p>	
	<code>href=URL</code>	The link target.	
	<code>nohref=nohref</code>	Enables you to set the defined area to have no action when the user selects it. <code>nohref</code> is the only possible value of this attribute.	
	<code>shape=rect </code> <code>circle poly </code> <code>default</code>	Defines the shape of the clickable region.	
	<code>target=_blank </code> <code>_parent _self </code> <code>_top [name]</code> <i>(deprecated)</i>	Defines where the target URL opens. <i>Cannot be used in XHTML Strict.</i>	

Element	Attribute	Description	Standard attributes
		<p>Renders text as bold.</p> <p>This element is a physical style, which defines what the content looks like (presentation only), rather than a logical style, which defines what the content is (which is beneficial for technologies like screen readers). It's recommended to use the logical element in place of (see separate entry).</p> <p>See also Chapter 3, "Styles for emphasis (bold and italic)."</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<base />	<p>href=<i>URL</i> (required)</p> <p>target= <i>blank</i> <i>_parent</i> <i>_self</i> <i>_top</i> [<i>name</i>] (deprecated)</p>	<p>Specifies a base URL for relative URLs on the web page.</p> <p>Defines the base URL to use. This attribute is <i>required</i>.</p> <p>Defines where to open page links. Can be overridden by inline target attributes. <i>Cannot be used in XHTML Strict.</i></p>	
<bdo>	<p>dir=<i>ltr</i> <i>rtl</i> (required)</p>	<p>Overrides the default text direction.</p> <p>Defines text direction as left to right (<i>ltr</i>) or right to left (<i>rtl</i>). This attribute is <i>required</i>.</p>	<p>Core attributes, language attributes</p>

A

continues

Element	Attribute	Description	Standard attributes
<big>		<p>Increments text size to the next size larger as compared to surrounding text. Because the size differential is determined by the browser, precise text size changes are better achieved via span elements and CSS. Some browsers misinterpret this tag and render text as bold.</p> <p>See also Chapter 3, “The big and small elements.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<blockquote>		<p>Defines a lengthy quotation. To validate as XHTML Strict, enclosed content must be set within a block-level element (such as <p></p>).</p> <p>Although it is common for web designers to use this element to indent content, the W3C strongly recommends using CSS for such things.</p> <p>See also Chapter 3, “Block quotes, quote citations, and definitions,” and “Creating drop caps and pull quotes using CSS.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
	<code>cite=URL</code>	Defines the online location of quoted material.	
<body> (required)		Defines the document’s body and contains the document’s contents. This is a <i>required</i> element for XHTML web pages. (In HTML, it is optional and implied when absent. However, it’s good practice to always include the element.)	<p>Core attributes, language attributes</p> <p>Core events, onload, onunload</p>
 		Inserts a single line break.	Core attributes

Element	Attribute	Description	Standard attributes
<button>		Defines a push button element within a form. Works similarly to buttons created with the input element, but offers greater rendering scope. This is because all content becomes the content of the button, enabling the creation of buttons with text and images. For example: <button type="submit"> Order now! <img ➡ src="go.gif" alt="Go" </>	Core attributes, keyboard attributes, language attributes Core events, onBlur, onFocus
	disabled=disabled	Disables the button. disabled is the only possible value of this attribute.	
	name=name	Defines the button's name.	
	type=button reset submit	Identifies the button's type.	
	value=string	Defines the button's initial value.	
<caption>		Defines a caption for a table. Seldom used, but recommended because it enables you to associate a table's title with its contents. Omitting the caption may mean the table's contents are meaningless out of context. See also Chapter 6, "Captions and summaries."	Core attributes, language attributes Core events
<cite>		Defines content as a citation. Usually rendered in italics. See also Chapter 3, "Block quotes, quote citations, and definitions."	Core attributes, language attributes Core events

A

continues

Element	Attribute	Description	Standard attributes
<code>		<p>Defines content as computer code sample text. Usually rendered in a monospace font.</p> <p>See also Chapter 3, “Logical styles for programming-oriented content,” and the “Displaying blocks of code online” exercise.</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<col />		<p>Defines properties for a column or group of columns within a colgroup. Attributes defined within a col element override those set in the containing colgroup element. col is an empty element that contains attributes only. The following example sets the column widths of the table’s first three columns to 10, 30, and 50 pixels, respectively:</p> <pre><colgroup span="3"> <col width="10"></col> <col width="30"></col> <col width="50"></col> </colgroup></pre> <p>See also the <colgroup> entry.</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
	align=left right justify center (deprecated)	<p>Defines the horizontal alignment of table cell content. It’s recommended that you use the CSS text-align property instead (see its entry in the CSS reference) to do this.</p>	
	span=n	<p>Defines how many successive columns are affected by the col tag. Use only when the surrounding colgroup element does not specify the number of columns.</p> <p>The following example creates a colgroup with five columns, with each of the middle three columns 30 pixels wide:</p> <pre><colgroup> <col width="10" /> <col width="30" span="3" /> <col width="50" /> </colgroup></pre>	

Element	Attribute	Description	Standard attributes
	<i>valign=top middle bottom baseline (deprecated)</i>	Specifies the vertical alignment of table cell content. It's recommended that you instead use the CSS vertical-align property (see its entry in the CSS reference) to do this.	
	<i>width=percentage/number</i>	Defines the width of the column. Overrides the width settings in colgroup.	
<colgroup>		Defines a column group within a table, enabling you to define formatting for the columns within. See the <col /> entry for examples. See also Chapter 6, "Scope and headers."	Core attributes, language attributes Core events
	<i>align=left right justify center (deprecated)</i>	Defines the horizontal alignment of the table cell content within the colgroup. It's recommended that you instead use the CSS text-align property (see its entry in the CSS reference) to do this.	
	<i>span=number</i>	Defines how many columns the colgroup should span. Do not use if any of the col tags within the colgroup also use span, because a colgroup definition will be ignored in favor of span attributes defined within the col elements.	
	<i>valign=top middle bottom baseline (deprecated)</i>	Specifies the vertical alignment of the table cell content within the colgroup. It's recommended that you instead use the CSS vertical-align property (see its entry in the CSS reference) to do this.	

continues

Element	Attribute	Description	Standard attributes
	<i>width=percentage/ number</i>	Defines the width of columns within the colgroup. Can be overridden by the width settings of individual col elements.	
<dd>		Defines a definition description within a definition list. See the <dl> entry for an example. See also Chapter 3, “Definition lists,” and the “Displaying blocks of code online” exercise.	Core attributes, language attributes Core events
		Indicates deleted text. Usually appears in strikethrough format. See also Chapter 3, “Elements for inserted and deleted text.”	Core attributes, language attributes Core events
	<i>cite=URL</i>	Defines the URL of a document that explains why the text was deleted.	
	<i>datetime=date</i>	Defines the date and time that the text was amended. Various formats are possible, including YYYY-MM-DD and YYYY-MM-DDThh:mm:ssTZD (where TZD is the time zone designator). See www.w3.org/TR/1998/NOTE-datetime-19980827 for more date and time formatting information.	
<dfn>		Defines enclosed content as the defining instance of a term. Usually rendered in italics. See also Chapter 3, “Block quotes, quote citations, and definitions.”	Core attributes, language attributes Core events

Element	Attribute	Description	Standard attributes
<div>		<p>Defines a division within a web page. Perhaps one of the most versatile but least understood elements. Used in combination with an <code>id</code> or <code>class</code>, the <code>div</code> tag element allows sections of a page to be individually styled and is the primary XHTML element used for the basis of CSS-based web page layouts.</p> <p>See also Chapter 7, “Workflow for CSS layouts.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<dl>		<p>Defines a definition list. Contains pairs of term and definition elements, as follows:</p> <pre><dl> <dt>Windows</dt> <dd>Operating system ➔ made by Microsoft.</dd> <dt>Mac OS</dt> <dd>Operating system ➔ made by Apple.</dd> </dl></pre> <p>See also Chapter 3, “Definition lists,” and the “Displaying blocks of code online” exercise.</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<dt>		<p>Defines a definition term within a definition list. See the <code><dl></code> entry for an example.</p> <p>See also Chapter 3, “Definition lists,” and the “Displaying blocks of code online” exercise.</p>	<p>Core attributes, language attributes</p> <p>Core events</p>

A

continues

Element	Attribute	Description	Standard attributes
		<p>Defines enclosed content as emphasized. Generally renders as italics in a browser and is preferred over the use of <i></i>. See separate <i> entry.</p> <p>See also Chapter 3, “Block quotes, quote citations, and definitions.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<embed> (nonstandard)		<p>Embeds an object. Nonstandard and not supported by any XHTML DOCTYPE. If this is included in a web page, the page will not validate. Poor browser support for the W3C preferred alternative, object, left developers with little choice other than to use this nonstandard element when embedding Flash or other multimedia into a web page. Support for object has now improved, although embed may still be required in some circumstances.</p>	
	align= <i>left right top bottom</i>	Defines the alignment of the embedded object in relation to the surrounding text.	
	height= <i>number</i>	Defines the height of the object in pixels.	
	hidden= <i>yes no</i>	Hides the player or media file when set to yes. Defaults to no.	
	hspace= <i>number</i>	Sets horizontal space around the object.	
	name= <i>name</i>	Sets a name for the object.	
	pluginspage= <i>URL</i>	Defines a URL for information on installing the relevant plug-in.	

Element	Attribute	Description	Standard attributes
	<i>src=URL (required)</i>	Provides the location of the object to be embedded. This attribute is <i>required</i> .	
	<i>type=MIME type</i>	Specifies the MIME type of the plug-in required to run the file.	
	<i>vspace=number</i>	Sets vertical space around the object.	
	<i>width=number</i>	Defines the width of the object in pixels.	
<fieldset>		Creates a group of related form elements by nesting them within the <code>fieldset</code> element. Usually used in tandem with the <code>legend</code> element to enhance form accessibility (see the <legend> entry for more information). See also Chapter 8, “Improving form accessibility.”	Core attributes, language attributes Core events
	<i>accesskey=character</i>	Defines a keyboard shortcut to access an element.	
<form>		Indicates the start and end of a form. Cannot be nested within another <code>form</code> element. Generally, the <code>method</code> and <code>action</code> attributes are most used. See also Chapter 8, “Working with forms.”	Core attributes, language attributes Core events, <code>onreset</code> , <code>onsubmit</code>
	<i>accept=content-type list</i>	Specifies a comma-separated list of MIME types that the server processing the form can handle correctly.	
	<i>accept-charset=charset list</i>	Specifies a comma-separated list of character sets for form data.	

A

continues

Element	Attribute	Description	Standard attributes
	<p><code>action=URL</code> (required)</p> <p><code>enctype=encoding</code></p> <p><code>method=get post</code></p> <p><code>name=name</code> (deprecated)</p> <p><code>target=_blank _parent _self _top [name]</code> (deprecated)</p>	<p>The URL of the form processing application where the data is sent once the form is submitted. This attribute is <i>required</i>.</p> <p>The MIME type used to encode the form's content before it's sent to the server, so it doesn't become scrambled. Defaults to <code>application/x-www-form-urlencoded</code>. Other options are <code>multipart/form-data</code>, which can be used when the user is able to upload files, and <code>text/plain</code>, which can be used when using a <code>mailto:</code> value for the action instead of a server-side script to parse the form data.</p> <p>Specifies the http method used to submit the form data. The <code>post</code> value is most commonly used.</p> <p>Defines the form's name. <i>Cannot be used in XHTML Strict.</i></p> <p>Defines where the target URL is opened. <i>Cannot be used in XHTML Strict.</i></p>	
<code><frame></code>		<p>Defines a frame. <i>This element and its attributes must only be used with the XHTML Frameset DTD, and not with XHTML Strict or XHTML Transitional.</i></p> <p>See also Chapter 7, "Working with frames."</p>	Core attributes

Element	Attribute	Description	Standard attributes
	<code>frameborder=0 1</code>	Defines whether frame borders are present (<code>frameborder="1"</code>) or not (<code>frameborder="0"</code>).	
	<code>longdesc=URL</code>	Defines a URL for a long description of the frame contents for non-frames-compatible browsers.	
	<code>marginheight=number</code>	The vertical space between the frame edges and its contents (measured in pixels).	
	<code>marginwidth=number</code>	The horizontal space between the frame edges and its contents (measured in pixels).	
	<code>name=name</code> <i>(deprecated)</i>	Defines a name for the frame.	
	<code>noresize=noresize</code>	Stops the user from resizing the frame. The only available value is <code>noresize</code> .	
	<code>scrolling=auto/no/yes</code>	Specifies whether scroll bars appear when the frame contents are too large for the visible area. The <code>yes</code> value mean permanent scroll bars are shown; <code>no</code> means scroll bars don't appear, even if the content is too large for the frame; and <code>auto</code> means scroll bars appear when the content is too large for the frame.	
	<code>src=URL</code>	Defines the location of the frame's default HTML document.	

A

continues

Element	Attribute	Description	Standard attributes
<frameset>		<p>Defines a frameset. Must have either a cols or a rows attribute. <i>This element and its attributes must only be used with the XHTML Frameset DTD, and not with XHTML Strict or XHTML Transitional.</i></p> <pre><frameset cols="150,* "> <frame src= ➤"frame-one.html" /> <frame src= ➤"frame-two.html" /> </frameset></pre> <p>See also Chapter 7, “Working with frames.”</p>	Core attributes onload, onunload
	<i>cols=percentage/ number"*</i>	<p>Defines the number and sizes of columns (vertical frames). When setting the value to *, the frame it's applied to takes up all remaining browser window space for that dimension. If more than one value is *, the remaining space is split between those frames the * value is assigned to.</p>	
	<i>rows=percentage/ number"*</i>	<p>Defines the number and sizes of rows (horizontal frames). See the preceding entry for an explanation of how the * value works.</p>	

Element	Attribute	Description	Standard attributes
<h <i>n</i> >		Defines enclosed contents as a heading. Available levels are 1 to 6. Note that although h4 through h6 tend to be displayed smaller than body copy by default, they are not a means to create small text; rather, they are a way to enable you to structure your document. This is essential, because headings help with assistive technology, enabling the visually impaired to efficiently surf the Web. See also Chapter 3, “Paragraphs and headings.”	Core attributes, language attributes Core events
<head> (<i>required</i>)	profile= <i>URL</i>	Defines the header of the HTML file. Houses information-based elements, such as base, link, meta, script, style, and title. This is a <i>required</i> element for XHTML web pages. (It’s optional for HTML, but implied when absent. However, it’s good practice to always include a head element in web pages.) The location of a metadata profile for this document. Not commonly used.	Language attributes
<hr />		Inserts a horizontal rule.	Core attributes, language attributes Core events

continues

Element	Attribute	Description	Standard attributes
<code><html></code> <i>(required)</i>	<code>xmlns=namespace</code>	<p>Defines the start and end of the HTML document. This is a <i>required</i> element for XHTML web pages. (It's optional for HTML, but implied when absent. However, it's good practice to always include a head element in web pages.) No HTML content should be placed before the html start tag or after the html end tag.</p> <p>Defines the XML namespace (e.g., <code>http://www.w3.org/1999/xhtml</code>).</p> <p>See also Chapter 2, "Document defaults."</p>	Language attributes
<code><i></code>		<p>Renders text as italic. This element is a physical style, which defines what the content looks like (presentation only), rather than a logical style, which defines what the content is (which is beneficial for technologies like screen readers). It's generally preferable to use the logical element <code></code> in place of <code><i></i></code>. See the preceding <code></code> entry.</p> <p>See also Chapter 3, "Styles for emphasis (bold and italic)."</p>	Core attributes, language attributes Core events
<code><iframe></code>		<p>Defines an inline frame. Content within the element is displayed only in browsers that cannot display the iframe. <i>This element and its attributes cannot be used in XHTML Strict.</i></p> <p>See also Chapter 7, "Working with internal frames (iframes)."</p>	

Element	Attribute	Description	Standard attributes
	<code>frameborder=0 1</code>	Defines whether a frame border is present (<code>frameborder="1"</code>) or not (<code>frameborder="0"</code>).	
	<code>height=percentage/number</code>	Defines the iframe's height.	
	<code>longdesc=URL</code>	Defines a URL for a long description of the iframe's contents for non-frames-compatible browsers.	
	<code>marginheight=number</code>	The vertical space (in pixels) between the iframe's edges and its contents.	
	<code>marginwidth=number</code>	The horizontal space (in pixels) between the iframe's edges and its contents.	
	<code>name=name (deprecated)</code>	Defines a name for the iframe.	
	<code>scrolling=auto no yes</code>	Specifies whether scroll bars appear when the iframe's contents are too large for the visible area. The <code>yes</code> value means permanent scroll bars are shown; <code>no</code> means scroll bars don't appear, even if the content is too large for the frame; and <code>auto</code> means scroll bars appear when the content is too large for the frame.	
	<code>src=URL</code>	Defines the location of the iframe's default HTML document.	
	<code>width=percentage/number</code>	Defines the iframe's width.	

A

continues

Element	Attribute	Description	Standard attributes
		<p>Inserts an image. Both the <code>src</code> and <code>alt</code> attributes are required; although many web designers omit the <code>alt</code> attribute, it's essential for screen readers. The <code>height</code> and <code>width</code> values are recommended, too, in order to assist the browser in rapidly laying out the page. The <code>border</code> value, despite common usage, is deprecated and should be avoided. Use CSS to determine whether images have borders.</p> <p>See also Chapter 4, "Working with images in XHTML."</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
	<code>alt=text</code> (required)	Provides alternate text for nonvisual browsers. Should provide an indication of an image's content or, if it's a link, its function. When an image has no visual semantic significance, include it via CSS. If that's not possible, use <code>alt=""</code> . This attribute is <i>required</i> .	
	<code>border=number</code> (deprecated)	Defines a border. Despite its common usage, this attribute is deprecated and cannot be used in XHTML Strict. Instead, use CSS to set borders on images.	
	<code>height=number</code>	Defines the image's height in pixels.	
	<code>ismap=URL</code>	Defines the image as a server-side image map. The image must be contained within an anchor tag. Server-side image maps require specialized setup and are rarely used. Do not confuse this attribute with <code>usemap</code> (see the upcoming <code>usemap</code> entry).	

Element	Attribute	Description	Standard attributes
	longdesc= <i>URL</i>	Provides the location of a document containing a long description of the image.	
	src= <i>URL</i> (<i>required</i>)	The URL of the image to be displayed. This attribute is <i>required</i> .	
	usemap= <i>URL</i>	Defines the image as a client-side image map. See also Chapter 5, "Image maps."	
	width= <i>number</i>	Defines the image's width in pixels.	
<input />		Defines a form input field. See also Chapter 8, "Adding controls."	Core attributes, keyboard attributes, language attributes Core events, onblur, onchange, onfocus, onselect
	accept= <i>list</i>	A list of MIME types that can be accepted by this element. <i>Only used with type="file".</i>	
	alt= <i>text</i>	Provides alternate text for nonvisual browsers. <i>Only used with type="image".</i>	
	checked= <i>checked</i>	Sets input element's default state to checked. The only value for this attribute is checked. <i>Only used with type="checkbox" and type="radio".</i>	
	disabled= <i>disabled</i>	Disables the input element. The only value for this attribute is disabled. <i>Cannot be used with type="hidden".</i>	
	maxlength= <i>number</i>	Defines the maximum number of characters allowed. <i>Only used with type="text".</i>	

continues

Element	Attribute	Description	Standard attributes
	<code>name=name</code> (required*)	Defines a name for the input element. <i>* Required for the following types: button, checkbox, file, hidden, image, password, text, and radio.</i>	
	<code>readonly=readonly</code>	Indicates the input element is read-only and cannot be modified. The only value for this attribute is <code>readonly</code> . <i>Only used with type="text" and type="password".</i>	
	<code>size=number</code>	Defines in characters (<i>not</i> pixels) the width of the input element. (For pixel-defined widths, use CSS.) <i>Cannot be used with type="hidden".</i>	
	<code>src=URL</code>	Defines the URL of the image to be displayed. <i>Only used with type="image".</i>	
	<code>type=button checkbox file hidden image password radio reset submit text</code>	Defines the input element type. Defaults to <i>text</i> .	
	<code>value=string</code> (required when type=checkbox and type=radio)	When type="button", type="reset", or type="submit", it defines button text. When type="checkbox" or type="radio", it defines the result of the input element; the result being sent when the form is submitted. When type="hidden", type="password", or type="text", it defines the element's default value. When type="image", it defines the result of the field passed to the script. <i>Cannot be used with type="file".</i>	

Element	Attribute	Description	Standard attributes
<ins>		Defines inserted text. Usually appears in underline format, which can be confusing because links are also underlined. It's therefore recommended that you use CSS to change the underline color. ins { text-decoration: none; border-bottom: 1px solid red; }	Core attributes, language attributes Core events
	cite=URL	Defines the URL of a document that explains why the text was inserted.	
	datetime=date	Defines the date and time that the text was amended. Various formats are possible, including YYYY-MM-DD and YYYY-MM-DDThh:mm:ssTZD (where TZD is the time zone designator). See www.w3.org/TR/1998/NOTE-datetime-19980827 for more date and time formatting information.	
<kbd>		Defines “keyboard” text (text inputted by the user). Usually rendered in a monospace font. See also Chapter 3, “Logical styles for programming-oriented content.”	Core attributes, language attributes Core events

A

continues

Element	Attribute	Description	Standard attributes
<label>		<p>Assigns a label to a form control, enabling you to define relationships between text labels and form controls. For example:</p> <pre><p><label for= ➡ "realname">Name</label> ➡
 <input type="text" ➡ name="realname" ➡ id="realname" size="30" ➡ /></p></pre> <p>See also Chapter 8, “The label, fieldset, and legend elements.”</p>	<p>Core attributes, language attributes</p> <p>Core events, onblur, onfocus</p>
	<code>accesskey=character</code>	Defines a keyboard shortcut to access an element.	
	<code>for=text</code>	Defines the form element that the label is for. Value must be the same as the associated control element's id attribute value.	
<legend>		<p>Defines a caption for a fieldset. Must be nested within a fieldset element. For example:</p> <pre><fieldset> <legend>Caption for this fieldset</legend> [form labels/controls] </fieldset></pre> <p>See also Chapter 8, “The label, fieldset, and legend elements.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
	<code>accesskey=character</code>	Defines a keyboard shortcut to access an element.	
		<p>Defines a list item. Must be nested within or elements (see the separate and entries).</p> <p>See also Chapter 3, “Working with lists.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>

Element	Attribute	Description	Standard attributes
	<code>type=format</code> <i>(deprecated)</i>	Specifies the list type for the list item. (See the <code></code> and <code></code> entries for possible values.) <i>Cannot be used in XHTML Strict.</i>	
	<code>value=number</code> <i>(deprecated)</i>	Defines the number of the item in an ordered list. <i>Cannot be used in XHTML Strict.</i>	
<code><link /></code>		<p>Defines the relationship between two linked documents. Must be placed in the head section of a document. Mainly used for attaching external style sheets and favicons to a document. Also, modern blogging systems use <code>link</code> elements to define relationships between the current document and others, such as XML feeds, next and previous pages, and archives. When used fully, <code>link</code> elements can have considerable accessibility and usability benefits; for example, some modern browsers use the data to provide extra navigation toolbars/options.</p> <p>See also Chapter 2, “Attaching external CSS files: The <code>link</code> method,” and “Attaching favicons and JavaScript.”</p>	Core attributes, language attributes Core events
	<code>charset=charset</code>	Defines the character set of the target document.	
	<code>href=URL</code>	The URL of the target.	
	<code>hreflang=language code</code>	Defines the language of the linked document.	

A

continues

Element	Attribute	Description	Standard attributes
	<code>media=media type list</code>	Defines the target medium for the linked document (all, aural, braille, handheld, print, projection, screen, tty, or tv). More than one medium can be combined in a comma-delimited list.	
	<code>rel=relationship</code>	Specifies the relationship from the current document to the target document (alternate, appendix, bookmark, chapter, contents, copyright, glossary, help, index, next, prev, section, start, stylesheet, or subsection). More than one relationship can be combined in a space-separated list.	
	<code>rev=relationship</code>	Specifies the relationship from the target document to the current document (see the preceding entry for values).	
	<code>target=_blank _parent _self _top [name]</code> (<i>deprecated</i>)	Defines where the target URL opens. <i>Cannot be used in XHTML Strict.</i>	
	<code>type=MIME type</code>	Specifies the target's MIME type, such as text/css or text/javascript.	
<map>		Contains client-side image map specifications. Contains one or more area elements (see preceding <area /> entry). See also Chapter 5, "Image maps."	Core attributes, keyboard attributes, language attributes Core events, onblur, onfocus
	<code>id=name</code> (<i>required</i>)	Defines a unique name for the map. This attribute is <i>required</i> .	
	<code>name=name</code> (<i>deprecated</i>)	Defines a unique name for the map. (Superseded by id, but can be used for backward compatibility.)	

Element	Attribute	Description	Standard attributes
<meta />		Provides meta information about the document. Must be placed inside the HTML page's head section. Each meta element requires a content attribute and also an http-equiv or a name attribute. Most commonly used to define the character set, and to set keywords and descriptions for search engines (increasingly ineffective, as search engines now pay more attention to page content and links than to meta tags). See also Chapter 2, "meta tags and search engines," and "What about the XML declaration?"	Language attributes
	content= <i>string</i> (required)	Defines the value of the meta tag property.	
	http-equiv= <i>string</i>	Specifies the http equivalent name for the meta information. Examples are content-type, expires, refresh, and set-cookie.	
	name= <i>string</i>	Specifies a name for the meta information. Examples are author, description, generator, and keywords.	
	scheme= <i>string</i>	Specifies the metadata profile scheme.	
<noembed> (nonstandard)		Nested within embed elements and displayed only when the browser cannot display the embedded object. Nonstandard and not supported by any XHTML DOCTYPE. If this is included in a web page, the page will not validate.	

A

continues

Element	Attribute	Description	Standard attributes
<noframes>		Defines content to be displayed in non-frames-compatible browsers. Should be placed inside a frameset element. <i>Intended for use with XHTML Frameset DOCTYPE only.</i>	Core attributes, language attributes
<noscript>		Defines content to be displayed in browsers that don't support scripting. This is considered a "block-level" element, so it cannot be nested in an element that accepts only inline content, such as a paragraph, heading, or preformatted text. Can be used inside a div, form, or list item.	Core attributes, language attributes
<object>		Defines an embedded object. See also Chapter 7, "Scrollable content areas with CSS."	Core attributes, keyboard attributes, language attributes Core events
	archive= <i>URL</i>	Defines a list of URLs to resources used by the object.	
	border= <i>number</i> (<i>deprecated</i>)	Sets the object's border width. <i>Cannot be used in XHTML Strict.</i>	
	classid= <i>URL</i>	Defines the URL of the object.	
	codebase= <i>URL</i>	Defines the base URL of the object.	
	codetype= <i>MIME type</i>	Defines the object's MIME type.	
	data= <i>URL</i>	Defines the URL of the object's data.	
	declare= <i>declare</i>	Declares an object but does not download it until the object is used. The only value for this attribute is declare.	
	height= <i>number</i>	Defines the object's height in pixels.	

Element	Attribute	Description	Standard attributes
	<code>name=name</code>	Sets a unique name for this instance of the object, which can be used in scripts.	
	<code>standby=text</code>	Defines text to display while the object is downloading.	
	<code>type=MIME type</code>	Defines the object data's MIME type.	
	<code>usemap=URL</code>	Specifies the client-side image map to use with the object.	
	<code>width=number</code>	Defines the object's width in pixels.	
<code></code>		Defines the start and end of an ordered list. Contains one or more <code>li</code> elements. (see preceding <code></code> entry). See also Chapter 3, "Ordered lists."	Core attributes, language attributes Core events
	<code>start=number</code> <i>(deprecated)</i>	Starts the list numbering at the defined value instead of 1. <i>Cannot be used in XHTML Strict.</i>	
	<code>type=1 A a I i</code> <i>(deprecated)</i>	Specifies the list numbering system (1=default numerals, A=uppercase letters, a=lowercase letters, I=uppercase Roman numerals, and i=lowercase Roman numerals). <i>Cannot be used in XHTML Strict.</i>	

continues

Element	Attribute	Description	Standard attributes
<optgroup>		<p>Defines a form option group, enabling you to group related options in a select element. Beware: display output varies between browsers. Some italicize optgroup label values to highlight them, while others highlight them by inverting the optgroup label value. Others display them as per option values.</p> <pre> <select name="> <optgroup label="fruits"> <option value="Apple"> ➤ Apple</option> <option value="Pear"> ➤ Pear</option> </optgroup> <optgroup label="vegetables"> <option value="Carrot"> ➤ Carrot</option> <option value="Turnip"> ➤ Turnip</option> </optgroup> </select> </pre> <p>See also Chapter 8, “Adding controls.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
	<code>disabled=disabled</code>	Disables the option group. The only value for this attribute is disabled.	
	<code>label=string (required)</code>	Defines a label for the optgroup. This attribute is <i>required</i> .	
	<code>tabindex=number</code>	Defines the tab order of an element.	
<option>		<p>Defines an option within a drop-down list. Nested within a select element and can be placed within optgroup elements. (See separate <select> and <optgroup> entries.)</p> <p>See also Chapter 8, “Adding controls.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
	<code>disabled=disabled</code>	Disables the option. The only value for this attribute is disabled.	

Element	Attribute	Description	Standard attributes
	<i>label=string</i>	Defines a label for this option.	
	<i>selected=selected</i>	Sets the option as the default. The only value for this attribute is <i>selected</i> .	
	<i>value=string</i>	Defines the value of the option to be sent when the form is submitted.	
<p>		Defines a paragraph. See also Chapter 3, “Paragraphs and headings.”	Core attributes, language attributes Core events
<param>		Supplies parameters for applets and objects. Must be enclosed within an applet or object element, and must come at the start of the content of the enclosing element.	
	<i>id=name</i>	Defines a unique reference ID for the element.	
	<i>name=name</i>	Defines a unique name for the element.	
	<i>type=MIME type</i>	Specifies the MIME type for the element.	
	<i>value=string</i>	Defines the element's value.	
	<i>valuetype=data object ref</i>	Specifies the MIME type of the value as data, ref (the value of a URL pointing to the data), or object (the value of an object within the document).	
<pre>		Defines enclosed contents as preformatted text, thereby preserving the formatting from the HTML document. Usually displayed in a monospace font. Cannot contain images, objects, or any of the following tags: big, small, sub, and sup.	Core attributes, language attributes Core events

A

continues

Element	Attribute	Description	Standard attributes
	<code>width=number</code> (<i>deprecated</i>)	Defines the maximum number of characters per line. This attribute is deprecated; use CSS to define the element width instead. <i>Cannot be used in XHTML Strict.</i>	
<q>		Defines enclosed content as a short quotation. Some browsers automatically insert quote marks. See also Chapter 3, “Block quotes, quote citations, and definitions.”	Core attributes, language attributes Core events
	<code>cite=URL</code>	Defines the location of quoted online material.	
<s> (<i>deprecated</i>)		Defines strikethrough text. This element is deprecated and cannot be used in XHTML Strict. It’s recommended to use the element (see separate entry) instead.	Core attributes, language attributes Core events
<samp>		Defines enclosed content as a computer code sample. Usually rendered in a monospace font. See also Chapter 3, “Logical styles for programming-oriented content.”	Core attributes, language attributes Core events
<script>		Inserts a script into the document. See also Chapter 2, “Attaching favicons and JavaScript.”	
	<code>charset=charset</code>	Defines the script’s character set.	
	<code>defer=defer</code>	Indicates the script doesn’t generate document content. This attribute’s only value is defer. This allows the browser to delay parsing the script until after the page has loaded. Although this may speed up loading, it will generate script errors if user interaction results in a call to a script that still hasn’t been parsed. Use with care.	

Element	Attribute	Description	Standard attributes
	<code>language=encoding</code> <i>(deprecated)</i>	Specifies the scripting language. Superseded by the <code>type</code> attribute, and no longer required. <i>Cannot be used in XHTML Strict.</i>	
	<code>src=URL</code>	Provides the URL of an external script.	
	<code>type=MIME type</code> <i>(required)</i>	Defines the MIME type of the scripting language, such as <code>text/javascript</code> or <code>text/vbscript</code> . This attribute is <i>required</i> .	
<code><select></code>		Creates a drop-down menu or scrolling list (depending on whether <code>multiple</code> has been set). This element is a container for <code>option</code> and optional <code>optgroup</code> elements. (see separate <code><option></code> and <code><optgroup></code> entries). See also Chapter 8, “Adding controls.”	Core attributes, keyboard attributes, language attributes Core events, <code>onblur</code> , <code>onchange</code> , <code>onfocus</code>
	<code>disabled=disabled</code>	Disables the element. The only value for this attribute is <code>disabled</code> .	
	<code>multiple=multiple</code>	Specifies that multiple items can be selected. If absent, only single options can be selected. If included, the <code>select</code> element displays as a scrolling list rather than a drop-down menu. The only value for this attribute is <code>multiple</code> .	
	<code>name=name</code>	Defines a name for the element.	
	<code>size=number</code>	Sets the element to a pop-up menu when the value is 1, or a scrolling list when the value is greater than 1.	

A

continues

Element	Attribute	Description	Standard attributes
<code><small></code>		<p>Reduces text size as compared to the surrounding text. Because the browser determines the size differential, precise text size changes are better achieved via span elements and CSS.</p> <p>See also Chapter 3, “The big and small elements.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<code></code>		<p>Identifies a span of inline elements for applying styles to. For example:</p> <pre><p>Use span elements to create ➤ styled ➤ inline text.</p></pre>	<p>Core attributes, language attributes</p> <p>Core events</p>
<code><strike></code> (<i>deprecated</i>)		<p>Defines strikethrough text. This element is deprecated and cannot be used in XHTML Strict. It's recommended to use the <code></code> element (see separate <code></code> entry) instead.</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<code></code>		<p>Defines enclosed content as strongly emphasized. Generally renders as bold text in browsers and is preferred over <code></code>. (see separate <code></code> entry).</p> <p>See also Chapter 3, “Logical and physical styles.”</p>	<p>Core attributes, language attributes</p> <p>Core events</p>
<code><style></code>		<p>Used to embed CSS rules in the head of a web page or to import CSS files.</p> <pre><style type="text/css" ➤ media="all"> @import url(stylesheet.css); .thisPageOnly { color: #de3de3; } </style></pre> <p>See also Chapter 2, “Attaching CSS files: The @import method.”</p>	<p>Language attributes</p>

Element	Attribute	Description	Standard attributes
	<i>media=list (required)</i>	Defines target media on which this style can be rendered. Possible values are all, aural, braille, handheld, print, projection, screen, tty, and tv.	
	<i>title=string</i>	Specifies the element's title.	
	<i>type=MIME type (required)</i>	Defines the MIME type of the style's contents. The only currently viable value is text/css, although this may change in the future. The value text/javascript is also allowed.	
<sub>		Defines contents as subscript text. See also Chapter 3, "Teletype, subscript, and superscript."	Core attributes, language attributes Core events
<sup>		Defines contents as superscript text. See also Chapter 3, "Teletype, subscript, and superscript."	Core attributes, language attributes Core events
<table>		Defines the start and end of a table. See also Chapter 6, "How tables work."	Core attributes, language attributes Core events
	<i>border=number</i>	Defines the table border width.	
	<i>cellpadding=percentage number</i>	Defines the space between cell edges and contents.	
	<i>cellspacing=percentage number</i>	Defines the space between table cells.	
	<i>summary=string</i>	Provides a summary of the table contents for nonvisual browsers.	
	<i>width=percentage number</i>	Defines the table's width in pixels or as a percentage of the available space within its parent element.	
<tbody>		Defines the table body. See also Chapter 6, "Row groups" and "Building a table."	Core attributes, language attributes Core events

A

continues

Element	Attribute	Description	Standard attributes
	<code>align=left right justify center</code> <i>(deprecated)</i>	Defines the horizontal alignment of table cell content. It's recommended that you use the CSS <code>text-align</code> property instead (see its entry in the CSS reference) to do this.	
	<code>valign=top middle bottom baseline</code> <i>(deprecated)</i>	Specifies the vertical alignment of table cell content. It's recommended that you instead use the CSS <code>vertical-align</code> property (see its entry in the CSS reference) to do this.	
<code><td></code>		Defines a table cell. See also Chapter 6, "How tables work" and "Building a table."	Core attributes, language attributes Core events
	<code>align=left right justify center</code> <i>(deprecated)</i>	Defines the horizontal alignment of table cell content. It's recommended that you use the CSS <code>text-align</code> property instead (see its entry in the CSS reference) to do this.	
	<code>axis=name</code>	Provides a name for a related group of cells. Not commonly used. (Note: any <code>td</code> cells containing the <code>axis</code> attribute are/should be treated as table header cells by the user agent.)	
	<code>colspan=number</code>	Defines how many columns the cell spans. See also Chapter 6, "Spanning rows and cells."	
	<code>headers=id list</code>	A list of cell IDs that provide header information for this cell, thereby enabling nonvisual browsers to associate header information with the cell. If more than one value is used, values are space separated. Example: <pre><th id="theTitle"> ➤ scope="col">The title</th> <th id="price"> ➤ scope="col">Price</th> <td headers="theTitle">A new ➤ book</td> <td headers="price">\$29.99</td></pre>	

Element	Attribute	Description	Standard attributes
	height= <i>number</i> (<i>deprecated</i>)	Defines the height of a cell in pixels. This attribute is deprecated—use CSS to define cell dimensions. <i>Cannot be used in XHTML Strict.</i>	
	nowrap= <i>nowrap</i> (<i>deprecated</i>)	Disables text wrapping. The only value for this attribute is nowrap. <i>Cannot be used in XHTML Strict. Use the CSS white-space property (see its entry in the CSS reference) instead.</i>	
	rowspan= <i>number</i>	Defines how many rows the cell spans. See also Chapter 6, “Spanning rows and cells.”	
	valign= <i>top middle bottom baseline</i> (<i>deprecated</i>)	Specifies the vertical alignment of table cell content. It’s recommended that you instead use the CSS vertical-align property (see its entry in the CSS reference) to do this.	
	width= <i>number</i> (<i>deprecated</i>)	Defines the width of a cell in pixels. This attribute is deprecated—use CSS to define cell dimensions. <i>Cannot be used in XHTML Strict.</i>	
<textarea>		Defines a text area within a form. Any element content is displayed as the textarea’s default value, and that includes spaces. Therefore, if you want a blank textarea, avoid having any spaces between the start and end tags. Although the cols and rows attributes are required, you can override these settings by using CSS. See also Chapter 8, “Adding controls.”	Core attributes, language attributes Core events, onBlur, onChange, onFocus
	cols= <i>number</i> (<i>required</i>)	Specifies the visible width in characters of the textarea. This attribute is <i>required</i> .	
	disabled= <i>disabled</i>	Disables the element. The only value for this attribute is disabled.	
	name= <i>name</i>		

continues

Element	Attribute	Description	Standard attributes
	<code>readonly=readonly</code>	Indicates the textarea is read-only and cannot be modified. The only value for this attribute is <code>readonly</code> .	
	<code>rows=number</code> (required)	Specifies the visible height (expressed as a number of rows) of the textarea. This attribute is <i>required</i> .	
<tfoot>		Defines a table footer. See also Chapter 6, “Row groups” and “Building a table.”	Core attributes, language attributes Core events
	<code>align=left right justify center</code> (deprecated)	Defines the horizontal alignment of table cell content. It’s recommended that you use the CSS <code>text-align</code> property instead (see its entry in the CSS reference) to do this.	
	<code>valign=top middle bottom baseline</code> (deprecated)	Specifies the vertical alignment of table cell content. It’s recommended that you instead use the CSS <code>vertical-align</code> property (see its entry in the CSS reference) to do this.	
<th>		Defines a table header cell. See also Chapter 6, “How tables work” and “Building a table.”	Core attributes, language attributes Core events
	<code>abbr=string</code>	Provides an abbreviation of the cell’s contents. Browsers can then choose to use this if they are short on space or to aid accessibility. Not commonly used, but particularly potentially useful for screen readers.	Core attributes, language attributes Core events
	<code>align=left right justify center</code> (deprecated)	Defines the horizontal alignment of table cell content. It’s recommended that you instead use the CSS <code>text-align</code> property (see its entry in the CSS reference) to do this.	
	<code>axis=name</code>	Provides a name for a related group of cells. Not commonly used.	

Element	Attribute	Description	Standard attributes
	<code>colspan=number</code>	Defines how many columns the cell spans. See also Chapter 6, “Spanning rows and cells.”	
	<code>headers=id list</code>	A list of cell IDs that provide header information for this cell, thereby enabling nonvisual browsers to associate header information with the cell. If more than one value is used, values are space separated. Example: <pre><th id="theTitle" ➡ scope="col">The title</th> <th id="price" ➡ scope="col">Price</th> <td headers="theTitle">A new ➡ book</td> <td headers="price">\$29.99</td></pre>	
	<code>height=number</code> <i>(deprecated)</i>	Defines the height of a cell in pixels. This attribute is deprecated—use CSS to define cell dimensions. <i>Cannot be used in XHTML Strict.</i>	
	<code>nowrap=nowrap</code> <i>(deprecated)</i>	Disables text wrapping. The only value for this attribute is nowrap. <i>Cannot be used in XHTML Strict. (Use CSS whitespace instead.)</i>	
	<code>rowspan=number</code>	Defines how many rows the cell spans. See also Chapter 7, “Spanning rows and cells.”	
	<code>scope=col </code> <code>colgroup row </code> <code>rowgroup</code>	States whether the cell provides header information for the rest of the row, column, rowgroup, or colgroup that contains it. (See the headers description.)	
	<code>valign=top middle </code> <code>bottom baseline</code> <i>(deprecated)</i>	Specifies the vertical alignment of table cell content. It’s recommended that you instead use the CSS vertical-align property (see its entry in the CSS reference) to do this.	

A

continues

Element	Attribute	Description	Standard attributes
	<code>width=number</code> (<i>deprecated</i>)	Defines the width of a cell in pixels. This attribute is deprecated—use CSS to define cell dimensions. <i>Cannot be used in XHTML Strict.</i>	
<code><thead></code>		Defines a table header. See also Chapter 6, “Row groups” and “Building a table.”	Core attributes, language attributes Core events
	<code>align=left right justify center</code> (<i>deprecated</i>)	Defines the horizontal alignment of table cell content. It’s recommended that you use the CSS <code>text-align</code> property instead (see its entry in the CSS reference) to do this.	
	<code>valign=top middle bottom baseline</code> (<i>deprecated</i>)	Specifies the vertical alignment of table cell content. It’s recommended that you instead use the CSS <code>vertical-align</code> property (see its entry in the CSS reference) to do this.	
<code><title></code> (<i>required</i>)		Defines the title of a document. This is a <i>required</i> element for web pages. See also Chapter 2, “Page titles.”	Core attributes, language attributes
<code><tr></code>		Defines a table row. See also Chapter 6, “How tables work” and “Building a table.”	Core attributes, language attributes Core events
	<code>align=left right justify center</code> (<i>deprecated</i>)	Defines the horizontal alignment of table cell content. It’s recommended that you instead use the CSS <code>text-align</code> property (see its entry in the CSS reference) to do this.	
	<code>valign=top middle bottom baseline</code> (<i>deprecated</i>)	Specifies the vertical alignment of table cell content. It’s recommended that you instead use the CSS <code>vertical-align</code> property (see its entry in the CSS reference) to do this.	
<code><tt></code>		Renders as teletype (monospaced) text. See also Chapter 3, “Teletype, subscript, and superscript.”	Core attributes, language attributes Core events

Element	Attribute	Description	Standard attributes
		Defines the start and end of an unordered list. Contains one or more li elements (see separate entry). See also Chapter 3, “Unordered lists.”	Core attributes, language attributes Core events
<var>		Defines contents as a variable name. Usually rendered in italics. See also Chapter 3, “Logical styles for programming-oriented content.”	Core attributes, language attribute Core events

B WEB COLOR REFERENCE

This section of the reference guides provides an overview of how to write color values for the Web, as well as a full list of supported color names. See the “Color theory” section in Chapter 4 for a discussion of color theory.

Color values

On the Web, colors are displayed by mixing red, green, and blue (RGB) light. Values range from 0 to 255 and can be written as such (e.g., `rgb(5,233,70)`), but they are more commonly written in hexadecimal. Colors written in hex consist of a hash sign (#) followed by six digits. The six digits are made up of pairs, representing the red, green, and blue color values, respectively.

- #XXxxxx: Red color value
- #xxXXxx: Green color value
- #xxxxXX: Blue color value

Hexadecimal notation is a numbering system that has 16, rather than 10, as its base. Digits range from 0 to f, with 0 to 9 representing the same value as ordinary numbers, and the letters a to f representing 10 to 15. The letters can be either uppercase or lowercase. If you set the first two digits to their highest value (ff) and the others to null, you get #ff0000, which is the hex color value for red. If you write #00ff00, you get green, and if you write #0000ff, you get blue. If all are set to full, you get white (#ffffff), and if all are null values, you get black (#000000).

Hexadecimal can also be written in shorthand if the six-digit value is composed of pairs in which both numbers are the same. For instance, #ff6600 (orange) can be written as #f60, and #ffffff (white) can be written as #fff. All three pairs must consist of equal numbers. For instance, you cannot use shorthand for #ffff01. Also, although hexadecimal can be written in shorthand, many designers choose not to do so, because when all color values are written in full, it tends to be easier to scan CSS files for specific values.

Web-safe colors

The 216-color web-safe palette uses hex combinations of the following hex value pairs only: 00, 33, 66, 99, cc, and ff—for example, #cc6699, #33ff66, and #ff0000.

Using these pairs provides you with 216 colors that are said to not dither on Macs and Windows PCs that have 8-bit monitors (256 colors). Because the vast majority of monitors sold since 2000 are able to display thousands or millions of colors, this palette is now rarely used and is generally considered archaic and obsolete.

Color names

Although a significant number of HTML color names are supported by major browsers, the CSS standard only recognizes the following 17.

Color name	Color hex value	Shorthand hex	RGB
Aqua	#00ffff	#0ff	0,255,255
Black	#000000	#000	0,0,0
Blue	#0000ff	#00f	0,0,255
Fuchsia	#ff00ff	#f0f	255,0,255
Gray (or Grey)	#808080	n/a	128,128,128
Green	#008000	n/a	0,128,0
Lime	#00ff00	#0f0	0,255,0
Maroon	#800000	n/a	128,0,0
Navy	#000080	n/a	0,0,128
Olive	#808000	n/a	128,128,0
Orange	#ffa500	n/a	255,165,0
Purple	#800080	n/a	128,0,128
Red	#ff0000	#f00	255,0,0
Silver	#c0c0c0	n/a	192,192,192
Teal	#008080	n/a	0,128,128
White	#ffffff	#fff	255,255,255
Yellow	#ffff00	#ff0	255,255,0

Although each color name in the preceding table begins with a capital letter (for book style purposes), color names are case insensitive, and lowercase is most commonly used. However, most designers ignore color names entirely, using hex all the time for consistency's sake—a practice that the W3C recommends.

C ENTITIES REFERENCE

Generally speaking, characters not found in the normal alphanumeric set must be added to a web page by way of **character entities**. These take the form `&#n;`, with n being a two- to four-digit number. Many entities also have a name, which tends to be more convenient and memorable; these are also listed. However, entities are case sensitive, so take care when adding them to your web pages.

Although most browsers display nonalphanumeric characters when the relevant encoding is specified, it's sometimes necessary to use entities to ensure your page displays as intended across a large range of machines.

Most reference guides tend to list entities in numerical order , but I find it more useful to browse by grouped items, so I list entities alphabetically within sections such as “Common punctuation and symbols” and “Characters for European languages.” (The exception is for Greek characters, which I’ve listed in the order of the Greek alphabet , rather than in alphabetical order from an English language perspective.)

Characters used in XHTML

The less-than and ampersand characters are used in XHTML markup, and to avoid invalid and broken pages, these should be added to your web pages as entities. It's also common (although not required) to add greater-than and quotation marks as entities.

The ampersand character is commonly used in URL query strings (particularly when working with server-side languages), and in such cases, the & must be replaced by the entity name or number (it will still be correctly interpreted by the browser).

Character	Description	Entity name	Entity number
"	Quotation mark (straight)	";	";
&	Ampersand	&;	&;
<	Less-than sign	<;	<;
>	Greater-than sign	>;	>;

Punctuation characters and symbols

Although many web designers tend to get around punctuation character limitations by using double hyphens (--) in place of em dashes (—), triple periods (. . .) in place of an ellipsis (...), and straight quotation marks (") instead of “smart” quotes (”), XHTML supports many punctuation characters as character entities. Likewise, plenty of symbols are supported in XHTML, so you needn't write (c) when the copyright symbol is available.

This section lists all such characters and is split into four subsections: quotation marks, spacing and nonprinting characters, punctuation characters, and symbols.

Quotation marks

Character	Description	Entity name	Entity number
‘	Left single	‘;	‘;
’	Right single	’;	’;
“	Left double	“;	“;
”	Right double	”;	”;
◁	Single left angle	&lsho;;	‹;

Character	Description	Entity name	Entity number
›	Single right angle	›	›
«	Double left angle	«	«
»	Double right angle	»	»
,	Single low-9	‚	‚
„	Double low-9	„	„

Spacing and nonprinting characters

On Windows, zero-width joiner and zero-width nonjoiner may be displayed by default as a vertical bar with an x on top and a vertical bar, respectively. To display these as nonprinting characters, you may need to install the Arabic language pack.

Character	Description	Entity name	Entity number
	Em space	 	 
	En space	 	 
Nonprinting	Left-to-right mark	‎	‎
	Nonbreaking space	 	
	Overline	‾	‾
Nonprinting	Right-to-left mark	‏	‏
	Thin space	 	 
Nonprinting	Zero-width joiner	‍	‍
Nonprinting	Zero-width nonjoiner	‌	‌

Punctuation characters

Character	Description	Entity name	Entity number
	Broken vertical bar	¦	¦
•	Bullet point	•	•
†	Dagger	†	†
‡	Double dagger	‡	‡
"	Double prime, seconds, inches	″	″
...	Ellipsis	…	…
—	Em dash	—	—
–	En dash	–	–
/	Fraction slash	⁄	⁄
¡	Inverted exclamation mark	¡	¡
¿	Inverted question mark	¿	¿
'	Prime, minutes, feet	′	′
-	Soft hyphen	­	­

Symbols

Character	Description	Entity name	Entity number
ℑ	Blackletter capital I, imaginary part	ℑ	ℑ
ℜ	Blackletter capital R, real part	ℜ	ℜ
©	Copyright symbol	©	©
^a	Feminine ordinal	ª	ª

Character	Description	Entity name	Entity number
°	Masculine ordinal	º	º
¬	Not sign	¬	¬
¶	Paragraph sign	¶	¶
‰	Per mille symbol	‰	‰
®	Registered trademark symbol	®	®
§	Section sign	§	§
™	Trademark symbol	™	™
ℙ	Script capital P, power set	℘	℘

Characters for European languages

For any characters that have accents, circumflexes, or other additions, entities are available. However, many of these entities have their roots in the days when ASCII was the only available encoding method. These days, as long as you use the appropriate input method, and the page is correctly encoded, you may not need to use these entities. They are still listed here, though, for times when you just want to be on the safe side.

Take care when adding these, because case is important. In most cases, capitalizing the first letter of the entity name results in an uppercase character, but this isn't always so (notably the Icelandic characters “æth” and “thorn,” the uppercase versions of which require the entire entity name to be in uppercase).

Character	Description	Entity name	Entity number
´	Acute accent (no letter)	´	´
¸	Cedilla (no letter)	¸	¸
^	Circumflex spacing modifier	ˆ	ˆ
—	Macron accent	¯	¯

continues

Character	Description	Entity name	Entity number
·	Middle dot	·	·
~	Tilde	˜	˜
¨	Umlaut	&uml	¨
Á	Uppercase A, acute accent	Á	Á
á	Lowercase a, acute accent	á	á
Â	Uppercase a, circumflex accent	Â	Â
â	Lowercase a, circumflex accent	â	â
À	Uppercase A, grave accent	À	À
à	Lowercase a, grave accent	à	à
Å	Uppercase A, ring	Å	Å
å	Lowercase a, ring	å	å
Ã	Uppercase A, tilde	Ã	Ã
ã	Lowercase a, tilde	ã	ã
Ä	Uppercase A, umlaut	Ä	Ä
ä	Lowercase a, umlaut	ä	ä
Æ	Uppercase AE ligature	Æ	Æ
æ	Lowercase ae ligature	æ	æ
Ç	Uppercase C, cedilla	Ç	Ç
ç	Lowercase c, cedilla	ç	ç
É	Uppercase E, acute accent	É	É

Character	Description	Entity name	Entity number
é	Lowercase e, acute accent	é	é
Ê	Uppercase E, circumflex accent	Ê	Ê
ê	Lowercase e, circumflex accent	ê	ê
È	Uppercase E, grave accent	È	È
è	Lowercase e, grave accent	è	è
Ë	Uppercase E, umlaut	Ë	Ë
ë	Lowercase e, umlaut	ë	ë
Ð	Uppercase eth	Ð	Ð
ð	Lowercase eth	ð	ð
Í	Uppercase I, acute accent	Í	Í
í	Lowercase i, acute accent	í	í
Î	Uppercase I, circumflex accent	Î	Î
î	Lowercase i, circumflex accent	î	î
Ì	Uppercase I, grave accent	Ì	Ì
ì	Lowercase i, grave accent	ì	ì
Ï	Uppercase I, umlaut	Ï	Ï
ï	Lowercase i, umlaut	ï	ï

continues

Character	Description	Entity name	Entity number
Ñ	Uppercase N, tilde	Ñ	Ñ
ñ	Lowercase n, tilde	ñ	ñ
Ó	Uppercase O, acute accent	Ó	Ó
ó	Lowercase o, acute accent	ó	ó
Ô	Uppercase O, circumflex accent	Ô	Ô
ô	Lowercase o, circumflex accent	ô	ô
Ò	Uppercase O, grave accent	Ò	Ò
ò	Lowercase o, grave accent	ò	ò
Ø	Uppercase O, slash	Ø	Ø
ø	Lowercase o, slash	ø	ø
Õ	Uppercase O, tilde	Õ	Õ
õ	Lowercase o, tilde	õ	õ
Ö	Uppercase O, umlaut	Ö	Ö
ö	Lowercase o, umlaut	ö	ö
Œ	Uppercase OE ligature	Œ	Œ
œ	Lowercase oe ligature	œ	œ
Š	Uppercase S, caron	Š	Š
š	Lowercase s, caron	š	š
ß	Lowercase sz ligature	ß	ß

Character	Description	Entity name	Entity number
Þ	Uppercase thorn	Þ	Þ
þ	Lowercase thorn	þ	þ
Ú	Uppercase U, acute accent	Ú	Ú
ú	Lowercase u, acute accent	ú	ú
Û	Uppercase U, circumflex accent	Û	Û
û	Lowercase u, circumflex accent	û	û
Ù	Uppercase U, grave accent	Ù	Ù
ù	Lowercase u, grave accent	ù	ù
Ü	Uppercase U, umlaut	Ü	Ü
ü	Lowercase u, umlaut	ü	ü
Ý	Uppercase Y, acute accent	Ý	Ý
ý	Lowercase y, acute accent	ý	ý
Ÿ	Uppercase Y, umlaut	Ÿ	Ÿ
ÿ	Lowercase y, umlaut	ÿ	ÿ

Currency signs

Although the dollar sign is supported in XHTML, other common currency symbols are not. However, several can be added by way of entities, as shown in the following table.

Character	Description	Entity name	Entity number
¢	Cent	¢	¢
¤	General currency sign	¤	¤
€	Euro	€	€
£	Pound	£	£
¥	Yen	¥	¥

Mathematical, technical, and Greek characters

This set of entities combines mathematical and technical symbols and the Greek alphabet (which is commonly used in scientific work). For ease of use, this section is divided into three subsections: common mathematical characters (fractions and the most commonly used mathematical symbols), advanced mathematical and technical characters (characters of interest to those marking up technical documents or anything other than basic mathematical text), and Greek characters.

Common mathematical characters

Character	Description	Entity name	Entity number
°	Degree sign	°	°
÷	Division sign	÷	÷
½	Fraction—one half	½	½
¼	Fraction—one quarter	¼	¼
¾	Fraction—three quarters	¾	¾
>	Greater-than sign	>	>

Character	Description	Entity name	Entity number
\geq	Greater-than or equal to sign	≥	≥
$<$	Less-than sign	<	<
\leq	Less-than or equal to sign	≤	≤
$-$	Minus sign	−	−
\times	Multiplication sign	×	×
¹	Superscript one	¹	¹
²	Superscript two	²	²
³	Superscript three	³	³

Advanced mathematical and technical characters

Character	Description	Entity name	Entity number
\aleph	Alef symbol, first transfinite cardinal	ℵ	ℵ
\approx	Almost equal to, asymptotic to	≈	≈
\angle	Angle	∠	∠
\cong	Approximately equal to	≅	≅
$*$	Asterisk operator	∗	∗
\oplus	Circled plus, direct sum	⊕	⊕
\otimes	Circled times, vector product	⊗	⊗
\ni	Contains as member	∋	∋
\cdot	Dot operator	⋅	⋅

continues

Character	Description	Entity name	Entity number
\in	Element of	<code>&isin;</code>	<code>&#8712;</code>
\emptyset	Empty set, null set, diameter	<code>&empty;</code>	<code>&#8709;</code>
\forall	For all	<code>&forall;</code>	<code>&#8704;</code>
f	Function, florin (Latin small f with hook)	<code>&fnof;</code>	<code>&#402;</code>
\equiv	Identical to	<code>&equiv;</code>	<code>&#8801;</code>
∞	Infinity	<code>&infin;</code>	<code>&#8734;</code>
\int	Integral	<code>&int;</code>	<code>&#8747;</code>
\cap	Intersection, cap	<code>&cap;</code>	<code>&#8745;</code>
\lceil	Left ceiling	<code>&lceil;</code>	<code>&#8968;</code>
\lfloor	Left floor	<code>&lfloor;</code>	<code>&#8970;</code>
\wedge	Logical and, wedge	<code>&and;</code>	<code>&#8743;</code>
\vee	Logical or, vee	<code>&or;</code>	<code>&#8744;</code>
μ	Micro sign	<code>&micro;</code>	<code>&#181;</code>
∇	Nabla, backward difference	<code>&nabla;</code>	<code>&#8711;</code>
\prod	N-ary product, product sign	<code>&prod;</code>	<code>&#8719;</code>
Σ	N-ary summation	<code>&sum;</code>	<code>&#8721;</code>
\notin	Not an element of	<code>&notin;</code>	<code>&#8713;</code>
$\not\subset$	Not a subset of	<code>&nsup;</code>	<code>&#8836;</code>
\neq	Not equal to	<code>&ne;</code>	<code>&#8800;</code>
∂	Partial differential	<code>&part;</code>	<code>&#8706;</code>
\pm	Plus-minus sign, plus-or-minus sign	<code>&plusmn;</code>	<code>&#177;</code>

Character	Description	Entity name	Entity number
\propto	Proportional to	∝	∝
\rceil	Right ceiling	⌉	⌉
\rfloor	Right floor	⌋	⌋
$\sqrt{}$	Square root, radical sign	√	√
\subset	Subset of	⊂	⊂
\subseteq	Subset of or equal to	⊆	⊆
\supset	Superset of	⊃	⊃
\supseteq	Superset of or equal to	⊇	⊇
\exists	There exists	∃	∃
\therefore	Therefore	∴	∴
\sim	Tilde operator, varies with, similar to, approximately	∼	∼
\cup	Union, cup	∪	∪
\perp	Up tack, orthogonal to, perpendicular	⊥	⊥

Greek characters

Character	Description	Entity name	Entity number
A	Uppercase alpha	Α	Α
α	Lowercase alpha	α	α
B	Uppercase beta	Β	Β
β	Lowercase beta	β	β
Γ	Uppercase gamma	Γ	Γ

continues

Character	Description	Entity name	Entity number
γ	Lowercase gamma	γ	γ
Δ	Uppercase delta	Δ	Δ
δ	Lowercase delta	δ	δ
Ε	Uppercase epsilon	Ε	Ε
ε	Lowercase epsilon	ε	ε
Ζ	Uppercase zeta	Ζ	Ζ
ζ	Lowercase zeta	ζ	ζ
Η	Uppercase eta	Η	Η
η	Lowercase eta	η	η
Θ	Uppercase theta	Θ	Θ
θ	Lowercase theta	θ	θ
Ι	Uppercase iota	Ι	Ι
ι	Lowercase iota	ι	ι
Κ	Uppercase kappa	Κ	Κ
κ	Lowercase kappa	κ	κ
Λ	Uppercase lambda	Λ	Λ
λ	Lowercase lambda	λ	λ
Μ	Uppercase mu	Μ	Μ
μ	Lowercase mu	μ	μ
Ν	Uppercase nu	Ν	Ν
ν	Lowercase nu	ν	ν
Ξ	Uppercase xi	Ξ	Ξ
ξ	Lowercase xi	ξ	ξ
Ο	Uppercase omicron	Ο	Ο

Character	Description	Entity name	Entity number
ο	Lowercase omicron	ο	ο
Π	Uppercase pi	Π	Π
π	Lowercase pi	π	π
Ρ	Uppercase rho	Ρ	Ρ
ρ	Lowercase rho	ρ	ρ
ς	Lowercase final sigma	ς	ς
Σ	Uppercase sigma	Σ	Σ
σ	Lowercase sigma	σ	σ
Τ	Uppercase tau	Τ	Τ
τ	Lowercase tau	τ	τ
Υ	Uppercase upsilon	Υ	Υ
υ	Lowercase upsilon	υ	υ
Φ	Uppercase phi	Φ	Φ
φ	Lowercase phi	φ	φ
Χ	Uppercase chi	Χ	Χ
χ	Lowercase chi	χ	χ
Ψ	Uppercase psi	Ψ	Ψ
ψ	Lowercase psi	ψ	ψ
Ω	Uppercase omega	Ω	Ω
ω	Lowercase omega	ω	ω
ϑ	Small theta symbol	ϑ	ϑ
Υ	Greek upsilon with hook	ϒ	ϒ
ϖ	Greek pi symbol	ϖ	ϖ

Arrows, lozenge, and card suits

Character	Description	Entity name	Entity number
↵	Carriage return	↵	↵
↓	Down arrow	↓	↓
⇓	Down double arrow	⇓	⇓
←	Left arrow	←	←
⇐	Left double arrow	⇐	⇐
↔	Left-right arrow	↔	↔
⇔	Left-right double arrow	⇔	⇔
→	Right arrow	→	→
⇒	Right double arrow	⇒	⇒
↑	Up arrow	↑	↑
⇑	Up double arrow	⇑	⇑
◇	Lozenge	◊	◊
♣	Clubs suit	♣	♣
♦	Diamonds suit	♦	♦
♥	Hearts suit	♥	♥
♠	Spades suit	♠	♠

Converting the nonstandard Microsoft set

The final table in this section lists the nonstandard Microsoft set and modern equivalents. Some older HTML editors, such as Dreamweaver 4, insert nonstandard entity values into web pages, causing them to fail validation. Here, we present the outdated nonstandard value and its corresponding approved alternatives (entity name and entity number, either of which can be used).

Character	Description	Nonstandard value	Entity name	Entity number
,	Single low-9 quote	‚	‚	‚
f	Lowercase Latin f with hook (florin)	ƒ	ƒ	ƒ
„	Double low-9 quote	„	„	„
...	Ellipsis	…	…	…
†	Dagger	†	†	†
‡	Double dagger	‡	‡	‡
^	Circumflex spacing modifier	ˆ	ˆ	ˆ
‰	Per mille symbol	‰	‰	‰
Š	Uppercase S, caron	Š	Š	Š
<	Less-than sign	‹	<	<
Œ	Uppercase OE ligature	Œ	Œ	Œ
‘	Left single quote	‘	‘	‘
’	Right single quote	’	’	’
“	Left double quote	“	“	“
”	Right double quote	”	”	”
•	Bullet point	•	•	•
—	En dash	–	–	–
—	Em dash	—	—	—
~	Tilde	˜	˜	˜
™	Trademark symbol	™	™	™

continues

Character	Description	Nonstandard value	Entity name	Entity number
š	Lowercase s, caron	š	š	š
>	Greater-than sign	›	>	>
œ	Lowercase oe ligature	œ	œ	œ
ÿ	Uppercase Y, umlaut	Ÿ	Ÿ	Ÿ

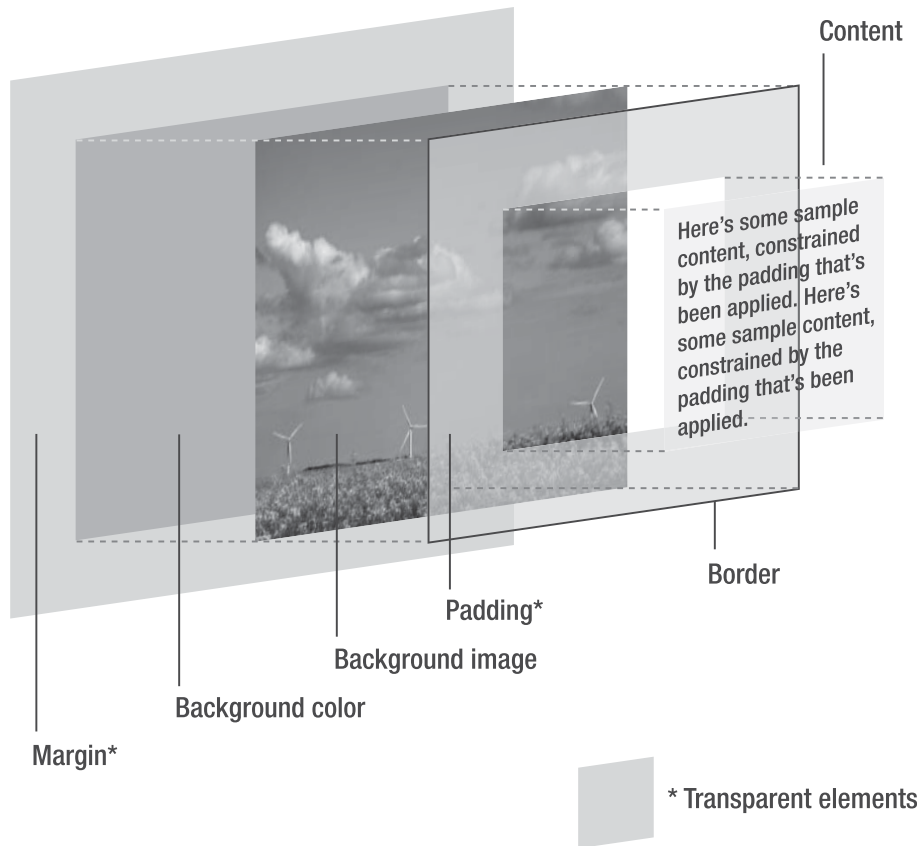
D CSS REFERENCE

This section includes a table listing CSS properties and values. In many cases, properties have specific values, which are listed in full. However, some values are common across many properties. These are outlined in the “Common CSS values” section, and in the CSS properties and values table these values are shown in italics. The end of the section includes information on basic selectors, pseudo-classes, pseudo-elements, CSS boilerplates, and CSS management.

The CSS box model

In CSS, every element is considered to be within its own box, and you can define the dimensions of the content and then add padding, a border, and a margin to each edge as required, as shown in the following image.

THE CSS BOX MODEL HIERARCHY



© Jon Hicks (www.hicksdesign.co.uk)

Padding, borders, and margins are added to the set dimensions of the content so the sum of these elements is the overall space that they take up. For example, a 100-pixel-wide element with 20 pixels of padding will take up an overall width of 140 pixels, not 100 pixels with 20 pixels of padding within.

Note that the top and bottom margins on adjacent elements collapse. For example, if you set the bottom margin to 50px on an element, and set a top margin of 100px on the element below, the margin between the two elements will be 100 pixels, not 150 pixels.

Internet Explorer 5.x for Windows gets the box model wrong, placing padding and borders inside the defined dimensions of an element. The bug is explained in Chapter 9, which also offers workarounds to fix layouts that get broken in aging versions of Microsoft’s browser.

Common CSS values

In addition to the values listed in the following table, a property may have a value of `inherit`, whereupon it takes the same value as its parent . Some properties are inherited by default—see the CSS properties and values table for more information.

D

Value	Formats
<i>color</i>	<p>Color name. See Appendix B (Color Reference) for information on available CSS color names.</p> <p><code>rgb(n,n,n)</code>: Where n is a value from 0 to 255 or a percentage. <code>#rrggbb</code>: Hexadecimal color format (preferred).</p>
<i>length</i>	<p>An optional sign (+ or -), followed by a number and one of the following units (there should be no whitespace between the number and unit):</p> <p>%: A percentage. cm: Centimeters. em: One em is equal to the font size of the parent or current element (see following focus point for elaboration). ex: One ex is, in theory, equal to the font size of the x character of the current element. Most browsers render ex as half an em. in: Inches. mm: Millimeters. pc: Picas. 1pc = 12pt. pt: Points. 1pt = 1/72in. px: Pixels.</p> <p>For zero values, the unit identifier may be omitted. Generally, px, em, and % are the best units for screen design, and pt is best for print fonts.</p>
<i>number</i>	An optional sign (+ or -) followed by a number.
<i>percentage</i>	An optional sign (+ or -) followed by a number, immediately followed by the percentage symbol.
<i>url</i>	The word <code>url</code> immediately followed by parentheses, within which is placed a URL. The URL can optionally be enclosed in single or double quotes.

When setting element dimensions (width, height, margins, etc.), one em is equal to the font size of that element. However, when setting font sizes for an element, one em is equal to the font size of its parent element. In both cases, this is measured relative to the dimensions of the M character.

CSS properties and values

In the tables within this section, default values are listed in bold and shorthand properties are shaded in gray. A number of tables online list browser compatibility with regard to CSS. Some good examples of these and related resources can be found at the following URLs:

- www.westciv.com/style_master/academy/browser_support/index.html: CSS support for most browsers
- www.webdevout.net/browser-support-css: CSS support for Internet Explorer, Firefox, and Opera
- www.quirksmode.org/css/contents.html: Concentrates on quirks
- www.macedition.com/cb/resources/macbrowsercsssupport.html: CSS2 support in old Mac browsers—note: not updated since 2004
- www.macedition.com/cb/resources/abridgedcsssupport.html: CSS2 support chart for old browsers—note: not updated since 2004
- www.css3.info/selectors-test/: Live CSS3 support testing of your browser
- <http://devedge-temp.mozilla.org/toolbox/sidebars/>: Useful sidebar reference tools for Gecko browser users

Remember that such charts are guides only, are sometimes out of date, and should not be considered a replacement for thorough testing in a range of web browsers.

To inherit a parent element's style for a property, use the value `inherit`. To raise a property's weight in the cascade, use `!important`. Important declarations override all others.

```
p {color: red !important;}
```

Add comments to CSS files as follows:

```
/*
This is a comment in CSS
*/

/* This is a single-line comment */
```

Property	Values	Description	Inherited
background		<p>Shorthand for defining background property values in a single declaration. Values can be any of those from background-attachment, background-color, background-image, background-position, and background-repeat, in any order. Example:</p> <pre>background: #ffffff ➡ url(background.gif) fixed left ➡ repeat-y;</pre> <p>See also Chapter 2, “Web page backgrounds in CSS” and “CSS shorthand for web backgrounds.”</p>	No
background-attachment	scroll fixed	<p>Determines whether a background image is fixed or scrolls with the page.</p> <p>See also Chapter 2, “background-attachment.”</p>	No
background-color	transparent <i>color</i>	<p>Defines an element’s background color. See also Chapter 2, “background-color.”</p>	No
background-image	none <i>url</i>	<p>Sets an element’s background image. Example:</p> <pre>background-image: ➡ url(background_image.jpg);</pre> <p>See also Chapter 2, “background-image.”</p>	No
background-position	<i>length</i> <i>percentage</i> top center bottom left right	<p>Defines the initial position of the background image. Defaults to 0,0. Values are usually paired: x,y. Combinations of keyword, length, and percentage are permitted, although combining keywords with either length or percentages is buggy in some browsers. If only one keyword is provided, the other defaults to center. If only one length or percentage is given, it sets the horizontal position, and the vertical position defaults to 50%.</p> <p>See also Chapter 2, “background-position.”</p>	No

Continued

Property	Values	Description	Inherited
background-repeat	repeat repeat-x repeat-y no-repeat	Defines how the background image tiles. See also Chapter 2, “background-repeat.”	No
border		<p>Shorthand for defining border property values in a single declaration. Values can be any of those from border-width, border-style, and border-color. Borders are drawn on top of a box’s background. Example:</p> <p>border: 1px solid #000000;</p> <p>See also Chapter 4, “Applying CSS borders to images,” and Chapter 6, “Styling a table.”</p>	No
border-bottom		Shorthand for defining bottom border property values (see border).	No
border-bottom-color	<i>color</i> transparent	Sets the bottom border color.	No
border-bottom-style	(See border-style.)	Sets the bottom border style.	No
border-bottom-width	(See border-width.)	Sets the bottom border width.	No
border-collapse	collapse separate	<p>Defines a table’s border model. In the separate border model, which is the default, each table cell has its own distinct borders, but in the collapsed border model, adjacent table cells share borders.</p> <p>See also Chapter 6, “Adding borders to tables.”</p>	Yes
border-color	<i>color</i> transparent	Defines the element’s border color. Defaults to the element’s color.	No
border-left		Shorthand for defining left border property values (see border).	No
border-left-color	<i>color</i> transparent	Sets the left border color.	No
border-left-style	(See border-style.)	Sets the left border style.	No
border-left-width	(See border-width.)	Sets the left border width.	No

Property	Values	Description	Inherited
<code>border-right</code>		Shorthand for defining right border property values (see <code>border</code>).	No
<code>border-right-color</code>	<i>color</i> transparent	Sets the right border color.	No
<code>border-right-style</code>	(See <code>border-style</code> .)	Sets the right border style.	No
<code>border-right-width</code>	(See <code>border-width</code> .)	Sets the right border width.	No
<code>border-spacing</code>	<i>length length</i>	Defines the distance between borders or adjacent table cells when using the separated borders model. (See <code>border-collapse</code> .) If a single length is given, it's used for horizontal and vertical values; if two lengths are provided, the first is used for the horizontal spacing and the second for the vertical spacing. Negative values are not permitted.	Yes
<code>border-style</code>	dashed dotted double groove inset none outset ridge solid	<p>Sets the style of an element's borders. Can work as shorthand, with one style per edge, from the top clockwise. Example:</p> <pre>border-style: solid dashed dotted ➡ groove;</pre> <p>Not all styles are supported in all browsers. Notably, Internet Explorer 5 and 6 render dotted as dashed when a border is 1 pixel in width.</p>	No
<code>border-top</code>		Shorthand for defining top border property values (see <code>border</code>).	No
<code>border-top-color</code>	<i>color</i> transparent	Sets the top border color.	No
<code>border-top-style</code>	(See <code>border-style</code> .)	Sets the top border style.	No
<code>border-top-width</code>	(See <code>border-width</code> .)	Sets the top border width.	No
<code>border-width</code>	<i>length</i> medium thick thin	<p>Sets the width of an element's borders. Can work as shorthand:</p> <pre>border-width: 1px 2px 3px 4px;</pre> <p>See also Chapter 4, "Applying CSS borders to images."</p>	No

Continued

Property	Values	Description	Inherited
bottom	auto <i>length</i> <i>percentage</i>	Determines the vertical offset of the element's bottom edge from the bottom edge of its parent element if the parent is positioned; if not, then offset is determined from the first positioned ancestor. Must be used with a position value of relative, absolute, or fixed.	No
caption-side	bottom top	Specifies the position of table caption elements with relation to the table element box.	Yes
clear	both left none right	Moves the element down until its margins are clear of floated elements to its left, right, or both sides. (See the float entry.) See also Chapter 7, "Placing columns within wrappers and clearing floated content."	No
clip	auto (shape)	Creates a clipping area for an absolute positioned element to determine the visible area. As of CSS 2.1, the only available shape is rect. Example: clip: rect(5px, 60px, 15px, 20px); As per the preceding code block, dimensions are stated as a comma-separated list, and percentage lengths are not permitted. The dimensions are, as per typical CSS shorthand, in the following order: top, right, bottom, left. The top and bottom values specify offsets from the top border edge of the box. The left and right measurements specify offsets from the left border edge of the box in left-to-right text and from the right border edge of the box in right-to-left text. The defined region clips out any aspect of the element that falls outside the clipping region. The preceding example creates a window 40 pixels wide and 10 pixels high, through which the content of the clipped element is visible. Everything else is hidden. See also www.w3.org/TR/CSS21/visufx.html#propdef-clip .	No

Property	Values	Description	Inherited
color	<i>color</i>	Sets an element's foreground color (i.e., the color of the text).	Yes
content	normal (<i>string</i>) <i>url</i> counter(<i>name</i>) counter(<i>name</i> , list-style-type) counters(<i>name</i> , <i>string</i>) counters(<i>name</i> , <i>string</i> , list- style-type) open-quote close-quote no- open-quote no- close-quote attr(<i>X</i>)	Generates content to attach before or after a CSS selector, using the :before and :after pseudo-elements. Example: #users h2:before { content: "Username: "; display: inline; } See also Chapter 7, "Placing columns within wrappers and clearing floated content."	No
counter-increment	none <i>identifier number</i>	Increments a counter when the current selector is encountered. The <i>identifier</i> defines the selector, ID, or class that is to be incremented; the optional <i>number</i> defines the increment amount. Used in conjunction with content. Browser support for this property is poor.	No
counter-reset	none <i>identifier number</i>	Defines a new value for the specified counter whenever the current selector is encountered.	No
cursor	auto crosshair default help pointer move progress text wait n-resize ne-resize e-resize se-resize s-resize sw-resize nw-resize w-resize <i>url</i>	Defines the cursor type to be displayed. Can be a comma-separated list. Cursors vary by system, so use this property with care. Also, if using custom cursors via the <i>url</i> value, include a generic cursor at the end of the list, in case of compatibility problems. Note: Internet Explorer 5.x for Windows does not recognize pointer, the correct CSS value for displaying a hand-shaped cursor. Instead, it uses the nonstandard value hand, which can be applied using a style sheet attached via a conditional comment.	Yes

Continued

Property	Values	Description	Inherited
direction	ltr rtl	Sets the direction of text flow. ltr: Left to right. rtl: Right to left.	Yes
display	block inline list-item none run-in inline-block table inline-table table-caption table-cell table-column table-column-group table-footer-group table-header-group table-row table-row-group table-row	States how an element is displayed on the page. The most common values are none, block, and inline, which all happen to be well supported. See several of the exercises in Chapters 5 and 7 for more on this property.	No
empty-cells	hide show	Determines whether empty table cell borders show when using the separated borders model. (See border-collapse.)	Yes
float	left none right	Defines whether an element floats left or right (allowing other content to wrap around it) or displays inline (by using the none value). See also Chapter 7, “The float property.”	No
font		Shorthand for defining font properties in a single declaration. Values can include any or all of the following: font-style, font-variant, font-weight, font-size, line-height, and font-family. Any omitted values revert to default settings, but font-size and font-family are mandatory. If font-style, font-weight, and font-variant values are included, they should appear at the start of the rule, prior to the font-size value.	Yes

Property	Values	Description	Inherited
font (continued)		<p>When using line-height, you must combine it with the font-size property using the syntax font-size/line-height (e.g., 12px/18px). Examples (using selected values):</p> <pre>font: bold 12px/16px Verdana, ➡ sans-serif; font: 85%/1.3em Georgia, serif;</pre> <p>See also Chapter 3, “Styling text using CSS” and “CSS shorthand for font properties.” Additional values for the font property are also available: caption, icon, menu, message-box, small-caption, status-bar. These set the font to system fonts, or the nearest equivalent, and are not available via font-family. However, these values are rarely, if ever, used.</p>	Yes
font-family	<i>(family name)</i> <i>(generic family)</i>	<p>Defines the font family of an element. Takes the form of a prioritized comma-separated list, which should terminate in a generic family name (cursive, fantasy, monospace, serif, or sans-serif).</p> <p>Multiple-word font-family names must be quoted (e.g., "Times New Roman"). Readers used to American typographical conventions should take care not to put commas inside the closing quotes. Example:</p> <pre>font-family: Georgia, "Times New ➡ Roman", serif;</pre> <p>See also Chapter 3, “Defining fonts.”</p>	Yes
font-size	xx-small x-small small medium large x-large xx-large smaller larger <i>length</i> <i>percentage</i>	<p>Sets the size of a font.</p> <p>See also Chapter 3, “Defining font size and line height.”</p>	Yes

Continued

Property	Values	Description	Inherited
font-style	<i>italic</i> normal <i>oblique</i>	Sets the font's style. See also Chapter 3, "Defining font-style, font-weight, and font-variant."	Yes
font-variant	normal <i>small-caps</i>	Sets the font to display in small caps. See also Chapter 3, "Defining font-style, font-weight, and font-variant."	Yes
font-weight	<i>lighter</i> normal bold bolder <i>number</i> *	Sets the font weight. * When using a number, it must be a multiple of 100 between 100 and 900 inclusive. The value 700 is considered equivalent to bold, and 400 is synonymous with normal. In practice, numbers are supported inconsistently and poorly in browsers. See also Chapter 3, "Defining font-style, font-weight, and font-variant."	Yes
height	auto <i>length</i> <i>percentage</i>	Sets the content height of an element.	No
left	auto <i>length</i> <i>percentage</i>	Determines the horizontal offset of the element's left edge from the left edge of its parent element if the parent is positioned; if not, then offset is determined from the first positioned ancestor. Must be used with a position value of relative, absolute, or fixed. See also the Chapter 7 exercise, "Using absolute positioning to center a box onscreen."	No
letter-spacing	<i>length</i> normal	Amends kerning (i.e., the space between characters). Positive and negative values are permitted. Relative values are determined once and then inherited. See also Chapter 3, "Setting letter-spacing and word-spacing."	Yes

Property	Values	Description	Inherited
line-height	normal <i>length</i> <i>number</i> <i>percentage</i>	Controls the element's leading. When the line-height value is larger than the font-size value, the difference (which is the leading) is halved, and this new value is applied to the top and bottom of the element's inline box. See also Chapter 3, "Setting line height."	Yes
list-style		Shorthand for defining list properties in a single declaration. Values can be those from list-style-type, list-style-position, and list-style-image. See also Chapter 3, "Styling lists with CSS" and "List style shorthand."	Yes
list-style-image	none <i>url</i>	Defines an image for list bullet points.	Yes
list-style-position	inside outside	Determines whether the bullet point appears as the first character of the list item content (inside) or in default fashion (outside).	Yes
list-style-type	none disc circle square decimal decimal-leading-zero lower-alpha upper-alpha lower-greek lower-latin upper-latin lower-roman upper-roman armenian georgian	Sets the bullet point style. If a browser doesn't understand an ordered list value, it defaults to decimal. Generally, none, circle, square, decimal, and the alpha and roman values are best supported. The W3C recommends using decimal for ordered lists whenever possible.	Yes

Continued

Property	Values	Description	Inherited
margin		<p>Shorthand for defining margin properties in a single declaration. Examples: margin: 0; (sets all margins to 0) margin: 0 10px 20px 30px; (sets individual margins for each edge)</p> <p>See also Chapter 2, “Content margins and padding in CSS” and “Working with CSS shorthand for boxes.”</p>	No
margin-bottom	auto <i>length</i> <i>percentage</i>	Sets the bottom margin. Defaults to 0. Note that browsers usually override the zero value by applying default margins to most block elements. Set margins explicitly to 0 to cancel the browser’s default. See Chapter 2, “Zeroing margins and padding on all elements.”	No
margin-left	auto <i>length</i> <i>percentage</i>	Sets the left margin. Defaults to 0. Note that browsers usually override the zero value by applying default margins to most block elements. Set margins explicitly to 0 to cancel the browser’s default. See Chapter 2, “Zeroing margins and padding on all elements.”	No
margin-right	auto <i>length</i> <i>percentage</i>	Sets the right margin. Defaults to 0. Note that browsers usually override the zero value by applying default margins to most block elements. Set margins explicitly to 0 to cancel the browser’s default. See Chapter 2, “Zeroing margins and padding on all elements.”	No
margin-top	auto <i>length</i> <i>percentage</i>	Sets the top margin. Defaults to 0. Note that browsers usually override the zero value by applying default margins to most block elements. Set margins explicitly to 0 to cancel the browser’s default. See Chapter 2, “Zeroing margins and padding on all elements.”	No
max-height	none <i>length</i> <i>percentage</i>	Sets the maximum height of an element. Does not apply to table elements.	No

Property	Values	Description	Inherited
max-width	none <i>length</i> <i>percentage</i>	Sets the maximum width of an element. Does not apply to table elements. See also the Chapter 7 exercise, “Creating a maximum-width layout.”	No
min-height	none <i>length</i> <i>percentage</i>	Sets the minimum height of an element. Does not apply to table elements.	No
min-width	none <i>length</i> <i>percentage</i>	Sets the minimum width of an element. Does not apply to table elements.	No
orphans	<i>number</i>	Defines the number of lines of a paragraph that must be left at the bottom of a page when printing. Defaults to 2. Defined number must be an integer. Very poorly supported.	Yes
outline		Shorthand for defining outline properties in a single declaration. Outlines are rendered outside the border edge and do not affect document flow. Example: <pre>.highlight { outline: 1px dotted #ff0000; }</pre> Not supported by Internet Explorer up to and including version 7.	No
outline-color	<i>color</i> invert	Sets the color of an outline. Defaults to invert , which inverts the color of the pixels onscreen, ensuring the outline is visible.	No
outline-style	<i>dashed</i> <i>dotted</i> <i>double</i> <i>groove</i> <i>inset</i> none <i>outset</i> <i>ridge</i> <i>solid</i>	Sets the style of an outline.	No
outline-width	<i>length</i> medium <i>thick</i> <i>thin</i>	Sets the width of an outline.	No

Continued

Property	Values	Description	Inherited
overflow	auto hidden scroll visible	<p>Determines what happens when content is too large for the defined dimensions of the element.</p> <p>auto: If content is clipped, the browser displays a scroll bar. hidden: Content is clipped, and content outside the element's box is not visible. scroll: Content is clipped, but a scroll bar is made available. visible: Content is not clipped and may be rendered outside of the element's containing box.</p> <p>See also Chapter 7, “Scrollable content areas with CSS.”</p>	No
padding		<p>Shorthand to define padding properties in a single declaration. Examples:</p> <p>padding: 0; (sets padding on all sides to 0) padding: 0 10px 20px 30px; (sets individual padding for each edge)</p> <p>See also Chapter 2, “Content margins and padding in CSS” and “Working with CSS shorthand for boxes.”</p>	No
padding-bottom	<i>length</i> <i>percentage</i>	Sets the bottom padding of an element.	No
padding-left	<i>length</i> <i>percentage</i>	Sets the left padding of an element.	No
padding-right	<i>length</i> <i>percentage</i>	Sets the right padding of an element.	No
padding-top	<i>length</i> <i>percentage</i>	Sets the top padding of an element.	No
page-break-after	auto always avoid left right	Determines whether a page break should appear after the element when printing. Poorly supported.	No
page-break-before	auto always avoid left right	Determines whether a page break should appear before the element when printing. Poorly supported.	No
page-break-inside	auto avoid	Determines whether a page break should appear inside the element when printing. Poorly supported.	Yes

Property	Values	Description	Inherited
position	absolute fixed relative static	<p>Determines the positioning method used to render the element's box:</p> <p>absolute: Element is placed in a specific location outside of normal document flow, using the top, right, bottom, and left properties.</p> <p>fixed: As per absolute, but the element remains stationary when the screen scrolls. Poorly supported by some browsers.</p> <p>relative: Offset from the static position by the values set using top, right, bottom, and left properties.</p> <p>static: The default. The top, right, bottom, and left properties do not affect the element if this value is set. The element is not removed from the document's normal flow.</p> <p>Various examples of this property in use are found in Chapters 5 and 7.</p>	No
quotes	none <i>string string</i>	Determines the type of quote marks to be used for embedded quotations. The string contains paired quoted values, which determine each level of quote embedding. The default depends on the user agent (browser).	Yes
right	auto <i>length</i> <i>percentage</i>	Determines the horizontal offset of the element's right edge from the right edge of its parent element if the parent is positioned; if not, then offset is determined from the first positioned ancestor. Must be used with a position value of relative, absolute, or fixed.	No
table-layout	auto fixed	Controls the layout algorithm used to render tables. Using fixed, table columns are based on analysis of the first row and rendered accordingly. This can speed up processing time, but may lead to columns that are too narrow for subsequently downloaded content.	No

Continued

Property	Values	Description	Inherited
text-align	center justify left* right	<p>Sets the text alignment for an element.</p> <p>* The default is left in left-to-right languages and right in right-to-left languages such as Arabic, Hebrew, and Urdu. Should be used instead of the HTML align attribute.</p>	Yes
text-decoration	blink line-through none overline underline	<p>Adds decoration to text. Values may be combined in a space-separated list, and the default depends on the element in question.</p> <p>Note that browsers may ignore blink but still be considered compliant. Examples:</p> <pre>text-decoration: underline; text-decoration: underline ➡ line-through;</pre> <p>See also Chapter 5, “Editing link styles using CSS.”</p>	No
text-indent	<i>length</i> <i>percentage</i>	Sets the horizontal indent of an element’s first line of text. Defaults to 0.	Yes
text-transform	capitalize lowercase none uppercase	<p>Sets the case of an element’s text.</p> <p>See also Chapter 3, “Controlling case with text-transform.”</p>	Yes
top	auto <i>length</i> <i>percentage</i>	<p>Determines the vertical offset of the element’s top edge from the top edge of its parent element if the parent is positioned; if not, then offset is determined from the first positioned ancestor. Must be used with a position value of relative, absolute, or fixed.</p> <p>See also the Chapter 7 exercise, “Using absolute positioning to center a box onscreen.”</p>	No

Property	Values	Description	Inherited
unicode-bidi	bidi-override embed normal	<p>Enables overrides for text direction. The embed value forces text to be displayed with regard to the associated direction property. The bidi-override value also overrides the default Unicode ordering scheme.</p> <p>This is a complex subject concerned with inserting elements of right-to-left text in blocks of left-to-right text (such as embedding Arabic or Hebrew in English, or vice versa). For details about working with bidirectional text, see www.w3.org/International/resource-index.html#bidi.</p>	No
vertical-align	<i>length</i> <i>percentage</i> baseline bottom middle top sub super text-bottom text-top	Determines the vertical alignment of an element. Applies to inline elements and those within table cells. Should be used in place of the HTML valign attribute. If a percentage value is used, that refers to the element's line-height value.	No
visibility	collapse hidden visible	Sets the visibility of an element. When hidden is used, the element box is invisible but still affects page layout (use display: none for an element to not affect document flow). When collapse is used, results are similar to hidden, except for spanned table cells, which may appear clipped.	Yes
white-space	normal nowrap pre pre-wrap pre-line	Determines how whitespace within an element is handled. Browser support for pre-line and pre-wrap is poor.	Yes
widows	<i>number</i>	Defines the number of lines of a paragraph that must be left at the top of a page when printing. Defaults to 2. Defined number must be an integer. Very poorly supported.	Yes

Continued

Property	Values	Description	Inherited
width	auto <i>length</i> <i>percentage</i>	Sets the content width of an element.	No
word-spacing	<i>length</i> normal	Provides space between words in addition to the default settings. See also Chapter 3, “Setting letter-spacing and word-spacing.”	Yes
z-index	auto <i>number</i>	Changes an element’s position in the stack. Higher numbers are “closer” and lower numbers are “further away.” Negative values are permitted, but will result in content not being displayed in some browsers.	No

Basic selectors

This section outlines the most commonly used selectors, along with their syntax. Note that selectors for pseudo-classes and pseudo-elements are covered in the following two sections, rather than being duplicated.

Some selectors are not fully supported in all browsers. Notably, child and adjacent selectors are not supported by versions of Internet Explorer prior to 7. See www.webdevout.net/browser-support-css for an overview of basic selector support.

Selector type	Syntax	Description
Universal	*	Matches any element. Can be used in context to attach a rule to all elements within another element (e.g., #sidebar *).
Type	<i>element</i>	Matches any element of type <i>element</i> . For example: h1.
Class	<i>.value</i>	Matches an element with a class value of <i>value</i> .
ID	<i>#value</i>	Matches an element with an id value of <i>value</i> .

Selector type	Syntax	Description
Descendant	<i>element descendant</i>	Matches a <i>descendant</i> element that is a descendant of the element of type <i>element</i> . For example, <code>div p</code> targets paragraphs that are descendants of <code>div</code> elements.
Child	<i>element>child</i>	Matches an element that is a child of another element. Similar to but more precise than descendant selectors, rules are applied to elements that are direct children of the parent only. For example, <code>div p</code> matches all paragraphs within all <code>div</code> s. <code>div>p</code> only matches paragraphs that are direct children of <code>div</code> s, and so would not match a paragraph within a table within a <code>div</code> .
Adjacent	<i>element1+element2</i>	Matches <i>element2</i> , adjacent to <i>element1</i> . For example, <code>h1+h2</code> matches any <code>h2</code> element that directly follows an <code>h1</code> element within the web page, with no other elements in between.
Attribute	<i>element[attribute]</i>	Matches an element of type <i>element</i> that has an attribute of type <i>attribute</i> . Further clarification can be added via the syntax <i>element[attribute="value"]</i> (targets <i>element</i> with <i>attribute</i> with value equal to <i>value</i>), <i>element[attribute~="value"]</i> (targets <i>element</i> with <i>attribute</i> that has a list of space-separated values, of which one is equal to <i>value</i>), <i>element[lang=value]</i> (targets <i>element</i> with a <code>lang</code> attribute equal to <i>value</i>).

*Note that the word `element` in the preceding table refers to a general element on the web page, rather than a *de facto* HTML element.*

Pseudo-classes

Pseudo-classes initially provided additional styles relating to a selector's state, but now also include those that apply styles to conceptual document components.

Pseudo-class	Description
:active	The state when an element is active (e.g., when a link is being clicked)
:first-child	Affects the first descendant of an element
:focus	The state when an element is focused to accept keyboard input
:hover	The state when the pointer is over an element
:lang	Applies to elements with the specified language (defined using <code>xml:lang</code>)
:link	Applies to an unvisited link
:visited	Applies to a visited link

Pseudo-elements

Pseudo-elements enable generated content that’s not in the document source and the styling of conceptual document components.

Pseudo-element	Description
:after	Used in conjunction with content to generate content after an element. For example: <code>h1:after {content: url(bleep.wav);}</code>
:before	Used in conjunction with content to generate content before an element.
:first-letter	Styles the first letter of an element.
:first-line	Styles the first rendered line of a “block-level” element.

CSS boilerplates and management

By using CSS comments and a monospace font when editing CSS, it’s possible to create clear sections within the style sheet and a table of contents, enabling you to more easily manage rules. A full example is available in the `advanced-boilerplates` folder of the download files. An example of a table of contents is shown following:

```

/*

STYLE SHEET FOR [WEB SITE]
Created by [AUTHOR NAME]
[URL OF AUTHOR]

ToC

    1. defaults
    2. structure
    3. links and navigation
    4. fonts
    5. images
    6. tables
    7. forms

Notes

*/

```

An example of a section of a boilerplate is shown following, with empty rules waiting to be filled. Here, a single tab is represented by eight spaces. Note how the property/value pairs and closing curly quotes are indented equally. This makes it easier to scan the far-left side of the document for selectors.

```

/* ----- 4. fonts ----- */

html {
    font-size: 100%;
}

body {
    font-size: 62.5%;
}

h1, h2, h3, h4, p, ul {
}

h1 {
}

h2 {
}

h3 {
}

h4 {
}

```

```
p {
}

ul {
}
```

The use of the CSS comment to introduce the section, with a string of hyphens before and after the section name, provides a useful visual separator for when directly editing code. Subsections are best added by indenting them the same amount as the property/value pairs; rule-specific comments are best placed after the opening curly quote; pair -specific comments are best placed after the pair. See the following for examples.

Sub-section introduction:

```
/* --- sidebar headings --- */
#sidebar h2 {
}

#sidebar h3 {
}
```

Rule-specific comment:

```
.boxoutProducts { /* used on sales and purchase pages */
}
```

Pair-specific comment:

```
body.advert h2 {
    font-size: 1.5em;
    text-transform: uppercase; /* over-ride for ad pages only */
}
```

Note that the indents in this section are different from those shown elsewhere in this book. This is intentional, in order to provide a close match to the code in the actual style sheet, rather than something that works better on the printed page.

Modular style sheets

From a management perspective, I find it easiest to work with a single style sheet, albeit one that already has a number of elements prewritten. However, you can also work in a modular manner, creating a number of small boilerplate documents (e.g., to reset margins and padding and define font size defaults) and area-specific style sheets (for navigation, layout, forms, etc.), and then importing them into your CSS via an `@import` line. As an example, you could save the `clearFix` rule (shown following, and used in various exercises throughout the book, notably in Chapter 7's “Clearing floated content” exercise) in its own style sheet as `clearfix.css`.

```
.clearFix:after {  
  content: ".";  
  display: block;  
  height: 0;  
  clear: both;  
  visibility: hidden;  
}
```

This could then be imported into your main style sheet as follows:

```
@import url(clearfix.css);
```

You can import as many style sheets as you want, depending on how modular you want to be, and how you want to organize your CSS. For example, at the time of writing, this book's technical editor, David Anderson, imports all of his CSS, using eight `@import` lines to do so, and separating out his CSS into categories such as “generic,” “navigation,” and “forms”. How you decide to work is up to you.

D

APPENDIX B FRAMES, AND HOW TO AVOID THEM



Some things, I'm afraid, are simply unavoidable, and when you write a book purporting to cover *every* (X)HTML tag that there is, you can't just not mention frames because you don't like them—tempting though that may be. So here they are, but discussed in an appendix rather than in a dedicated chapter, as their use usually contravenes every notion of good practice, and more often than not there's a better, more accessible and usable solution.

That said, we're long past the days when frames were in common use anyway. The many arguments against them—browsers struggled with printing framed pages and with book-marking them, browsers lacking support for frames, and so on—have since dwindled due to advances in browser technology. Despite these advances, however, problems do still exist:

- The browser's back button can no longer work intuitively.
- You can't easily reference a specific document within a frameset.
- The browser's reload button may reload the entire frameset, resetting the frame contents to their default sources, rather than reloading the specific framed document.
- Search engines can struggle when navigating through framed documents.
- Search engines can index pages outside of the frameset producing pages lacking in branding or navigation.
- Numerous accessibility issues exist with using frames.

Even discoverability can be a problem if the web designer has set things so that the frames lack borders and scrollbars; you may not even realize you're within a frameset until you try to refresh the page. Some browsers will allow you to right-click a framed page and reload it or bookmark it, but you have to know where it is to be able to manipulate it; it isn't always clear. Frames always were a clunky solution to a problem with serious usability and accessibility issues, and this remains the case today.

Furthermore, the arguments *for* them—mostly related to being able to include persistent and consistent navigation on every page—have also dwindled due to advances in server technology; using server-side includes (SSIs) or PHP server-side include statements (which I'll give an overview of later in this appendix in the section "Alternatives to frames") allows you to include consistent navigation on any page you like while only having to update one file. CSS also allows you to create a frame-like appearance, as you'll see further on in the section "Frame-like behavior with CSS."

So frames in their current incarnation can practically be considered obsolete. For those who want to build their own website, even the cheapest hosting packages these days will at least allow SSI, and the rise of hosted blog services such as LiveJournal, TypePad, and Blogger (where both the website code and markup are all provided alongside the hosting) means that amateur bloggers have much less of a need to build their own system from the ground up.

However, they are still used, albeit rarely—such as in an offline help package distributed with an application. There are also *inline frames*—the `<iframe>`—to consider. This element is used a lot more, mostly to embed third-party ads (for instance, Google's AdSense) on your webpages, but it has been removed from strict XHTML 1.0, so when I get to it I'll give you a look at what the W3C is planning for us to use instead: XFrames.

So, let's begin by taking a quick look at how frames are currently used in (X)HTML, which should be considered more as a historical footnote than recommended production techniques; if you think you need to use frames, please do consider all the alternatives first. If after that you *still* think you need to use frames, please use them with caution. Frame markup remains valid markup (more or less) to this day, but there are many, many good reasons as to why their use is now practically nonexistent.

(X)HTML frames

To use (X)HTML frames, first of all, you need to make sure you're using the correct doctype (you may remember this from Chapter 1). There is a specific doctype that should be used when, and *only* when, you're using frames. For sites written in HTML, that doctype is

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
```

while the XHTML equivalent is similar:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

That doctype is to be used on the root document—the one that will contain the frames. The doctypes you use on the files you're loading into the frames do not have to use the frameset doctype.

From there, the document will contain at least one `<frameset>`, which will define the number of rows and columns within the document, and will in turn contain usually at least two self-closing `<frame>` elements, which will load in the framed documents. So the markup for a simple two-frame document, in which the navigation frame is to be 80 pixels wide and the content frame is to fill the rest of the available space, would look somewhat like this:

```
<frameset cols="80,*">
  <frame src="nav.html" scrolling="no" noresize="noresize" ➡
  name="navFrame" title="Navigation" />
  <frame src="content.html" name="mainFrame" title="Content" />
</frameset>
```

There should technically also be a `<noframes>` section that displays content to those user agents lacking support for frames; those that do support frames would not display the contents. This is of less importance these days than it used to be, as all browsers come with frame support—even text browsers such as Lynx come with some support for navigating framed documents. Historically the `<noframes>` content provided by developers has been limited to messages such as “Your browser doesn’t support frames.” Not especially helpful, and another reason that the use of frames had been criticized in the past.

Note the few frame-specific attributes in the preceding code. The `<frameset>` uses the `cols` attribute, with a comma-separated list of values, to determine the number of columns contained within the frameset, and the sizes of each one (using the `*` character to indicate that the size should expand to fit—you can also use percentage values that allow the frames to scale with the size of the viewport). If your frameset contains horizontal frames instead of vertical frames (or a combination of both), the `rows` attribute works in the same way.

The `src` attribute is used to point to the file to be loaded within the frame; it can be an (X)HTML document, an image, a text file—anything that a browser can display. Because it allows for both relative and absolute URLs, it is possible to load another website's content into your own frameset—a nefarious practice referred to as **framejacking**.

The `noresize` attribute used on the `<frame>` element prevents the frame from being resized—fairly obvious that one. If left out, users can drag the border between the two frames (if it's visible) and change the width of the columns as they please.

There are some presentational attributes— `scrolling` (yes | no | auto) `frameborder`, `marginwidth`, `marginheight`—these can all be better controlled with the CSS `overflow`, `border`, and `margin` properties.

Finally, there exists here a `longdesc` attribute, which you may remember turned up in the discussion of images in Chapter 2. The principle is similar here: the value of a `longdesc` is a URL that points to a page giving a long description of the *purpose* of the frame, rather than its content—as the content of the frame can change, a hard-coded description of that content could cease very quickly to be accurate. The browser support however, is the same as for the `` variety—effectively zero.

Targeting links within frames

A link in one frame will always open within that frame, unless told otherwise; this can lead to another problem from the frames era, where websites could get stuck in somebody else's frameset.

To get a link in one frame to open in another, you need to use the deprecated `target` attribute on the link, and make sure each frame has a unique name. Thus, a link like this:

```
<a href="about.html" target="mainFrame">About</a>
```

will open up the page `about.html` not in its own frame, but in the frame named `mainFrame`. Adding `target` attributes on every link could get quite laborious, so you can save a bit of time by using the `<base>` element. This is a self-closing element that sits in the `<head>`, and it must be placed before any `<style>`, `<script>`, or `<link>` elements that are pointing to external sources. Its purpose is to define the root URL that all relative links should use as their base:

```
<head>
  <base href="http://unfortunatelypaul.com" />
  ...
```


So in the preceding example, when you click the `About` link, the browser would read that URL as `http://unfortunatelypaul.com/about.html`—even if the site domain was different

Why is this relevant here? Because you can also include a `target` attribute within the `<base>`:

```
<head>
  <base target="mainFrame" />
  ...
```

By including this in the markup of your navigation page, each link will gain the `target` attribute value without your needing to type it in on each `<a>`. To solve the problem of new sites opening within a frameset, you can use a target value of `_top` on any external hyperlinks, which will replace the entire frameset.

Inline frames

An inline frame does not require the use of a frameset doctype. It is used to create a fixed-width box¹ into which another document can be loaded, and it can be placed into a web page in exactly the same way as you would place a `<div>` or a `<table>` or any other element. It loads its content via the `src` attribute just like a regular frame, and its `noframes` content is actually stored within the content of the tag itself:

```
<iframe src="frame.html"><a href="frame.html">➡
View the content of this frame</a></iframe>
```

A browser that supported the `<iframe>` element would ignore its content and display the contents of `frame.html` within the frame boundaries; a browser that lacked `<iframe>` support would instead display the link to the document.

There are several presentational attributes that can be used, all of which can be better controlled with CSS; the `scrolling` attribute mentioned earlier that determines whether a scrollbar will ever be visible on the frame can be controlled with the `CSS overflow` property (`overflow: hidden`, `overflow: scroll`, and `overflow: auto`). The `frameborder`, `marginwidth`, and `marginheight` attributes can all be reproduced with `border` and `margin`.

Alternatives to frames

Server-side technologies allow for a lot of the functionality of frames to be reproduced. A classic reason for using frames was to create consistent navigation; by including the navigation menu in a frame, you could ensure it remained the same on every page. These days, unless you're creating a website that is to be run entirely from the client side, you can use

1. Although you can give an `<iframe>` a width or height value in percentages, the frame won't scale when you resize your viewport. For instance, an `<iframe>` with a width of 60% will remain fixed at 60% of whatever the width of the viewport was when the document was first loaded.

server-side includes that piece together your files on the web server before sending the completed pages down the pipe to the user I'm going to take a very brief look at two ways of doing this: with SSI and PHP.

First, SSIs. To use these, your files must usually end with an `.shtml` file extension rather than `.html` or `.htm`, though this can be dependent on how your web server is set up. An SSI looks like this:

```
<!--#include file="nav.html" -->
```

By placing that line in your web page, the server will insert the contents of `nav.html` into the document in its place (assuming that `nav.html` is contained within the same directory as the file you place this include in); then when you need to change your navigation, you just have to change the contents of `nav.html`. You can find detailed instructions on all you can do with SSIs in “Introduction to Server Side Includes” (<http://httpd.apache.org/docs/1.3/howto/ssi.html>).

If PHP is available on your server, you can use that instead. Again, you will most likely need to ensure your files all end with a `.php` extension, but servers can be set up to parse files with an extension of `.html` or `.htm` as PHP, so check with your system administrator. A typical PHP include looks like this:

```
<?php include 'nav.html'; ?>
```

Also available is the `require` function, which will cause the page to stop loading if `nav.html` is not available, and `require_once`, which acts in the same way as `require`, but will not include `nav.html` more than once. PHP errors will be generated and displayed to the user if there's a problem, but these error messages can be suppressed by prefixing the function with an ampersand, like this:

```
<?php @include 'nav.html'; ?>
```

More information on PHP includes can be found in the PHP manual (<http://uk2.php.net/manual/en/function.include.php>).

Frame-like behavior with CSS

If you've previously used frames because of their distinct visual nature—a combination of fixed and scrollable divisions—advances in server technology won't be of much use to you. However, advances in CSS (or, more accurately, advances in CSS support from the browsers) allow you to reproduce these effects to some extent, though sometimes not without a certain amount of cross-browser hackery.

At its most basic, use of the CSS `overflow` property can reproduce a frame-like effect; by applying `overflow: scroll` to an element and constraining its width or height, a scrollbar will appear, as shown in Figure B-1.

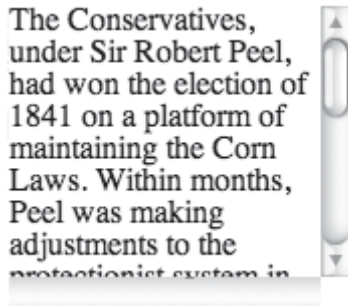


Figure B-1. A paragraph with a fixed width and height and `overflow:scroll` forcing a scrollbar; the default is for the paragraph contents to spill out across the boundaries of the paragraph.

You can be even more specific by using the `overflow-x` and `overflow-y` properties, which control the nature of the overflow in either the horizontal or vertical axis. These properties will cause your CSS to fail validation, however, as they were originally created by Microsoft (though they are now supported in Mozilla browsers, as well as the very latest bleeding-edge builds of Safari, and thus will probably be in a released version by the time this book is in your hands). The two properties are being included as part of the CSS3 box model module (www.w3.org/TR/2002/WD-css3-box-20021024/#the-overflow-x), so they will eventually be considered standard CSS, and the support is already there in the real world.

Also of use is the `position` property with a value of `fixed`, which allows you to position an element within the viewport and force it to remain in place while the rest of the content scrolls. For instance, in Figures B-2 and B-3, the heading (`anch1>`) remains visible while the rest of the page content scrolls behind it.

B

On the repeal of the Corn Laws: 28 May 1846

My Lords, I cannot allow this question for the second reading of this Bill to be put to your Lordships, without addressing to you a few words on the vote you are about to give. I am aware, my Lords, that I address you on this occasion under many disadvantages. I address your Lordships under the disadvantage of appearing here, as a Minister of the Crown, to press this measure upon your adoption, knowing at the same time how disagreeable it is to many of you with whom I have constantly acted in political life, with whom I have long lived in intimacy and friendship with the utmost satisfaction to myself -- on whose good opinion I have ever relied, and, I am happy to say, whose good opinion it has been my fortune hitherto to have enjoyed in no small degree.

Figure B-2. The header will remain in place as the content scrolls underneath it . . .

On the repeal of the Corn Laws: 28 May 1846

under many disadvantages. I address your Lordships under the disadvantage of appearing here, as a Minister of the Crown, to press this measure upon your adoption, knowing at the same time how disagreeable it is to many of you with whom I have constantly acted in political life, with whom I have long lived in intimacy and friendship with the utmost satisfaction to myself -- on whose good opinion I have ever relied, and, I am happy to say, whose good opinion it has been my fortune hitherto to have enjoyed in no small degree.

Figure B-3. . . . like so.

The CSS for this is simple:

```
h1 { position: fixed;}
```

This isn't supported in Internet Explorer 6 and below (support has been introduced in IE7), but there are various hacks and scripts available that can trick the browser and fake the effect I've had good results using a JavaScript solution provided by Doxdesk (www.doxdesk.com/software/js/fixed.html), but you can also try the various CSS solutions around. (There are several. Stu Nicholl's, available at www.cssplay.co.uk/layouts/fixed.html, is one I've used in the past to good effect.) A word of warning about this style of displaying your content though: if your users use their Page Down button to scroll the content, they can end up having to use their cursor keys to go back up a few lines as the content scrolls underneath the fixed element, so use with caution.

Roger Johansson has an exhaustive frame-like effect with support for multiple browsers, detailed in "CSS Frames, v2, full-height" (www.456bereastreet.com/archive/200609/css_frames_v2_fullheight/). This sees the creation of a page with a fixed-height header, a fixed-height footer that stays glued to the bottom of the browser viewport, and a scrollable middle section. Also of interest is Emil Stenström's "Frames or Iframes with CSS" (<http://friendlybit.com/css/frames-or-iframes-with-css/>).

Future frames: XFrames

The forthcoming XHTML 2.0 recommendation doesn't include any frame markup; that functionality will instead be available as an XML application named XFrames (not part of any (X)HTML specification but sharing the syntax and some common elements), which is being created to overcome many of the problems the current (X)HTML implementation of frames suffer from. At the time of writing, no browser has any support for XFrames aside from the X-Smiles browser mentioned in Chapter 7, so all of what's to follow is largely theoretical and also subject to change, as the specification has not yet been finalized.

XFrame markup is a lot like the current (X)HTML frame markup. Replacing the `<frameset>` is the `<frames>` tag, not to be confused with `<frame>`, which is more or less the same as its (X)HTML equivalent. Also the same are the `<head>`, `<title>`, and `<style>` elements—these all act exactly as you would expect them to. Finally there's a `<group>` element, which replaces the notion of nested framesets. Here's an example of a simple XFrames document—this is a trimmed down example from the current XFrames working draft:

```

<frames xmlns="http://www.w3.org/2002/06/xframes/">
  <head>
    <title>Home page</title>
  </head>
  <group compose="vertical">
    <frame xml:id="banner" source="banner.html" />
    <group compose="horizontal">
      <frame xml:id="atoz" source="atoz.html" />
      <frame xml:id="main" source="news.html" />
      <frame xml:id="nav" source="nav.html" />
    </group>
    <frame xml:id="footer" source="copyright.html" />
  </group>
</frames>

```

This describes a three-column layout topped and tailed with a horizontal header and a horizontal footer that both span the widths of the three central columns. You can see that the `src` attribute has been replaced by the `source` attribute—it still does the same thing, though its value can be overridden by values in the URL, discussed in a moment. I'll talk about the `compose` attribute in a moment, but first let's look at all those `xml:id` attributes.

The `xml:id` attribute is the same as the (X)HTML `id` attribute and follows the same rules. It's used for identifying the framed documents, but can also be used to populate the contents of the frames via a reference in the URL. For example, if you had a document with the file name `example.xframes`, and the markup was

```

<group compose="horizontal">
  <group compose="vertical">
    <frame xml:id="one" />
    <frame xml:id="two" />
  </group>
  <frame xml:id="three" source="default.html" />
</group>

```

you could populate all of those frames with a URL like this:

```

http://example.com/example.xframes#frames➡
(one=one.html,two=two.html,three=three.html)

```

It's a bit messy, but this way the URL is always representative of the contents of the frameset, changing in relation to the activities of the user, making it possible to bookmark and reference specific frameset configurations. The value provided in the URL for the frame named `three` will override the value set in the `source` attribute; if not set in the URL, it will display `default.html` instead. In the absence of any values supplied, the frames simply remain empty.

Regarding the presentation of XFrames, there's more flexibility, and they follow the same ethos of separating style from content that's seen in the XHTML 2.0 working draft—where all the markup can do is “suggest” to the rendering device how the markup should be dealt with. This is from the current XFrames working draft (www.w3.org/TR/xframes/):

“An XFrames document is a specification for composing several documents, potentially of different types, together in a view. The frames element forms the container for the composed document. The individual sub-documents (‘frames’) may be composed together in a rectangular space by placing them next to, or above, each other in rows and columns, or they may be displayed as separate movable window-like panes, or as tabbed panes, or in any other suitable manner.”

So there’s more flexibility in terms of displaying the framed content—traditionally , or as movable windows, or as tabbed panes, and so on; this allows for a greater level of device independence, with the user agent selecting a display method suitable to the display device. A “suggestion” as to the preferred way of presenting an XFrame document can be included within the XFrame markup—the `compose` attribute, used on the `<group>` tag. It accepts values of

- **vertical:** Frames within the group should be tiled vertically, one above the other. This would also be the default value if the user agent didn’t understand the supplied value.
- **horizontal:** Frames within the group should be tiled horizontally , all along the same row.
- **single:** Only one frame at a time should be visible, but the user should be able to see that other frames are selectable, so this is similar to a tabbed interface.
- **free:** The frames are available as movable, overlapping windows within the frames container.

So that’s XFrames in a nutshell: more powerful, flexible, and accessible than (X)HTML frames, but probably not usable in a production environment for a good few years yet—for the time being, merely another point of interest.

Summary

That covers frames—a collection of elements that were pretty horrible when first introduced, continued to be horrible during their heyday , and remain horrible in obscurity today. But, as the existence of XFrames demonstrates, they’re here to stay, in one form or another, and when used appropriately and with an awareness of the many usability and accessibility issues they can cause, they *can* be useful. Apart from `<iframe>`, though, it would be a rare situation that leads the modern web designer to needing to use frames, and until XFrames is finalized and has enough browser support to be usable, frames should be considered a historical curiosity more than anything else.

