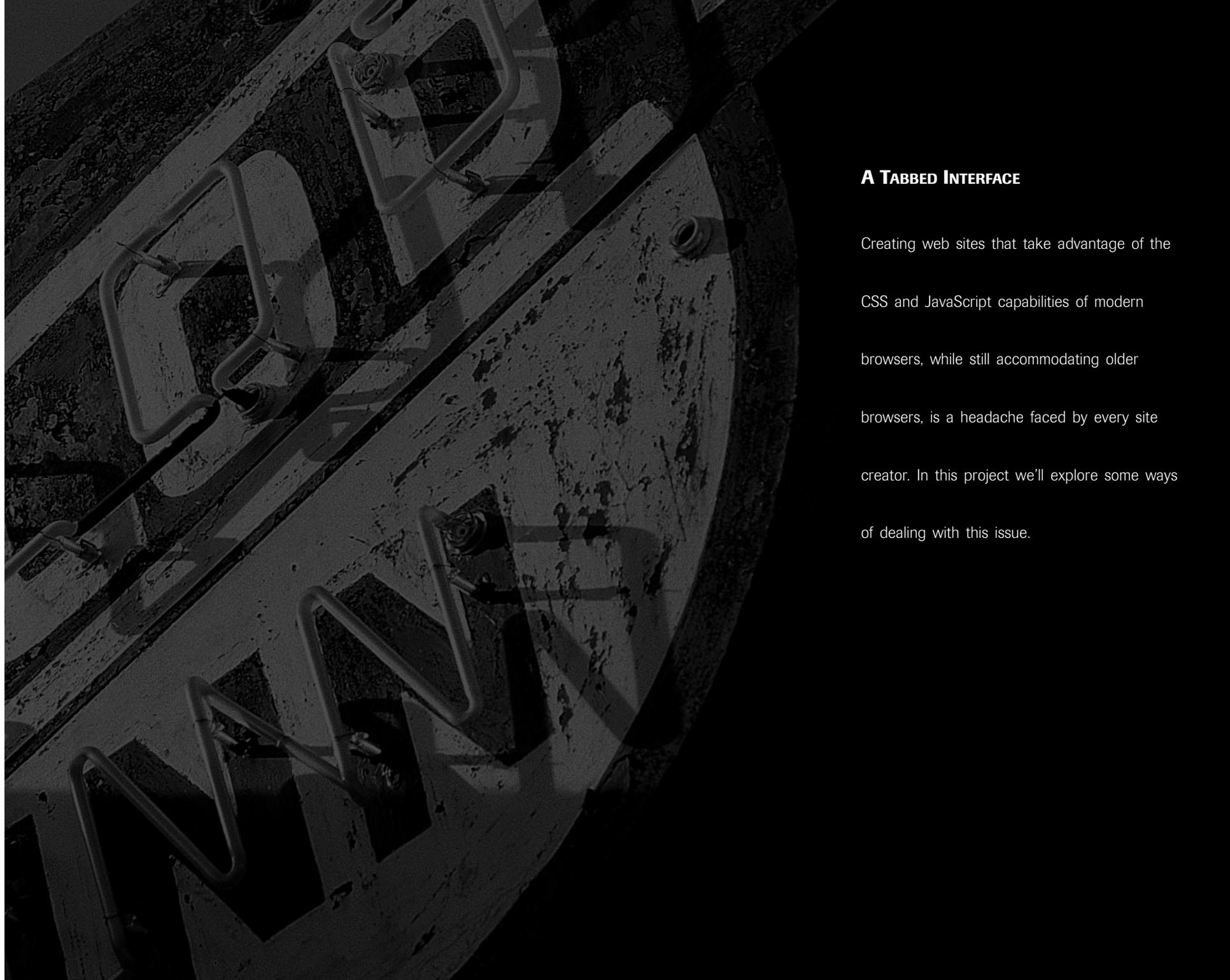




TRANSITIONAL CSS AND JAVASCRIPT STRATEGIES

*“In life, always do right. This
will gratify some people and
astonish the rest.”*

— MARK TWAIN

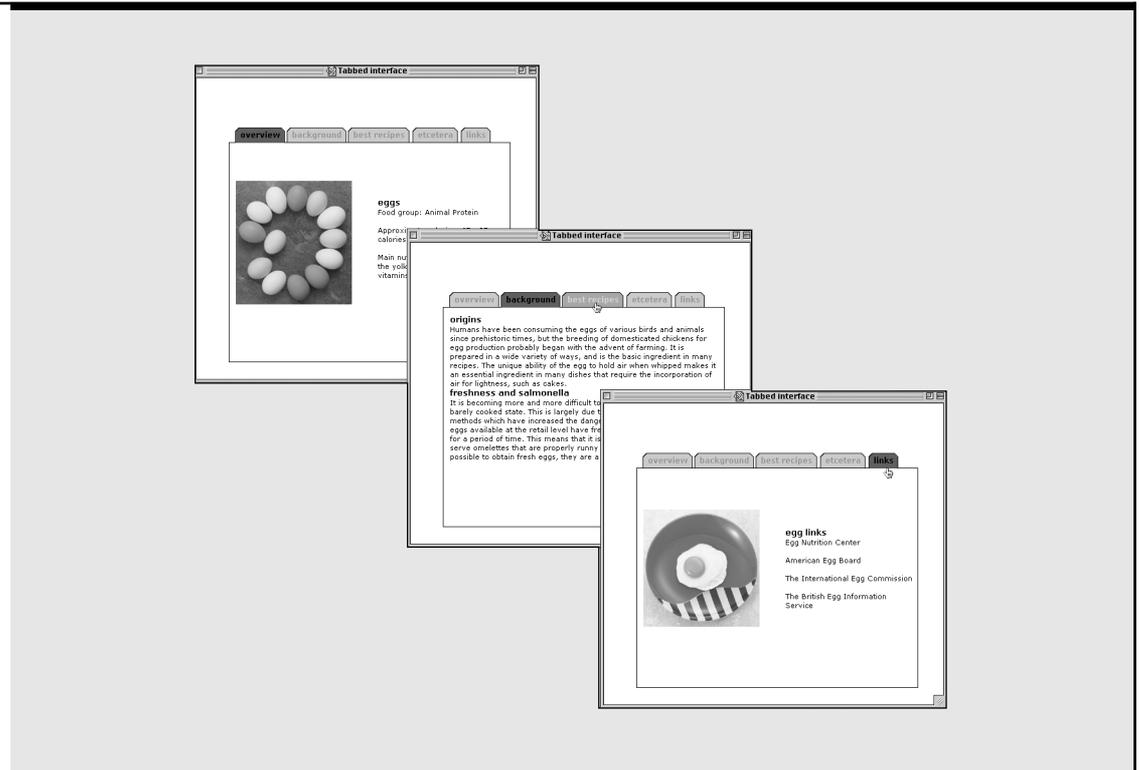


A TABBED INTERFACE

Creating web sites that take advantage of the CSS and JavaScript capabilities of modern browsers, while still accommodating older browsers, is a headache faced by every site creator. In this project we'll explore some ways of dealing with this issue.

Project 12

Transitional CSS and JavaScript Strategies



GETTING STARTED

Web site navigation interfaces that use a tab metaphor are very common. (Perhaps the best-known site that uses a tab-navigation interface is Amazon.com.) Yet in most cases, the navigation tabs are merely graphical links that load other pages from the server.

In this project, we show you how to create a tabbed “index-card” type of interface using a combination of CSS and JavaScript. All of the “pages” are actually contained in the one page; once the whole page is loaded into the browser, the user can very quickly “flip” through each “page” without having to reload content from the server.

Because we are using CSS and JavaScript, which are only recognized by the newer browsers, we’ve also taken steps to ensure that the page is still readable in older browsers.

Here are the main points we’ll cover in this project:

- Creating “gracefully degrading” pages using a combination of CSS and JavaScript
- Modularizing the code
- Using arrays to streamline the script
- Manipulating the stacking order, or z-index, of elements

GRACEFULLY DEGRADING PAGES

The latest advanced graphical browsers offer a wealth of JavaScript and CSS features that were previously unavailable or too buggy to be practical. What's more, these modern browsers finally (in large part) follow a common set of standards as specified by the World Wide Web Consortium. This is a great development for those of us on the creative side of the World Wide Web.

There's still one thorny problem to face, however: How do you take advantage of these new methods and features but still accommodate users of version 4 and older browsers? Statistics for browser usage are in a constant state of flux, but as of this writing, a small yet still significant number of web surfers are using version 4 browsers.

Up until now, you have probably been accommodating users of older or alternate browsers using "browser sniffing" methods

to redirect them to other pages, writing alternate content onto the page, and so on. However, this can greatly increase the time and cost needed to create and maintain a site.

In extreme cases, you may even have closed off users of older and alternate browsers by telling them that your site can only be accessed with certain browsers. This only invites resentment from your site visitors, who may never return to your site.

The method introduced in this project is called *graceful degradation*. Essentially, this means that you while you may design your page so that it's optimized for modern browsers, at the same time you ensure that the content of your site is viewable in older or non-graphical browsers. The goal is to make the same pages viewable and useable by the widest audience. See the "How It Works" sidebars at the end of this project to see how this is accomplished.

CREATING THE BASE GRAPHICS FOR THE TABS AND ASSEMBLING THE TAB TABLE

Each navigational tab is made up of three tiny graphics and the label text, assembled into a small table.

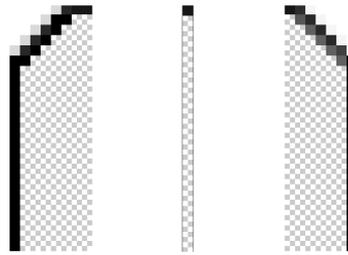
- 1 Create the basic outline shape of the tab. Make the inside of the tab itself transparent and make the outside corners the same color as the background color of your page.
- 2 Slice the image. You only want to obtain the two side segments. Save each slice as a GIF with transparency.

Note that the corners outside of the tab have a white background. This background color should match the background color you intend to use on your page.



The tab showing the transparent area (the checkered area) in Photoshop.

- 3 Select a 1-pixel wide segment of the middle section.
Save this as a GIF with transparency.



The three graphics: the tab sides, and the one-pixel-wide center graphic. The checkered area is the transparent area.

- 4 Pre-assemble the tab elements into an HTML/XHTML table.

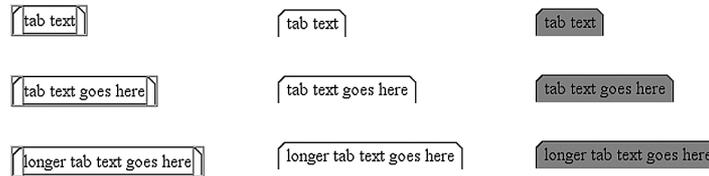
The tab consists of a small table with three table cells. Each table cell has a different background image. The left and right side table cells have a fixed width, but the middle table “stretches” to accommodate the text that is placed inside it.

The graphic `dot.gif` is a 1×1 pixel transparent GIF that is used to hold the side `td`s open.

```
<!-- The tab "table" --->
<table cellpadding="0" cellspacing="0" border="0">
<tr>
<td background="images/tableft.gif"></td>
<td id="tabmiddle4" background="images/middle.gif">tab text</td>
<td background="images/tabright.gif"></td>
</tr></table>
```

Note: Why not simply use the tab graphics as `img src` files instead of as background images for the table cells? This is possible, of course, but we’re going to use a bit of CSS trickery to make the graphics invisible in older browsers, as you’ll see later in this project.

Note: Normally, when you create a GIF with transparent areas, you make the areas surrounding the actual image transparent. However, if you make GIFs with the main areas set as transparent and the “background” areas in the same color as the background of your page, the “background” areas essentially act as masks. Then you can use CSS to “color” the main area of the element. This is a great way to create graphical elements that you can reuse again and again.



The tab “table” shown with the table borders visible (left), borders off (center), and with a background color set. The background color “shows through” the transparent areas of the GIFs.

PLANNING THE PROJECT AND DIVIDING IT INTO LOGICAL MODULES

This project is divided into modules. Instead of putting everything into one document, each specific task is put into its own document, and the result is assembled at the end. The CSS is divided into two separate stylesheets: one containing rules that are compatible with CSS-aware (version 4 and higher) browsers and the other containing rules that are recognized only in browsers that recognize more advanced CSS rules (version 5.0 and higher of Netscape 6.x, Mozilla, and Internet Explorer). The JavaScript is also divided into two: one that contains two document object constructor functions that can be used over and over again for many projects and another that contains the functions specific to this project. The markup itself is XHTML, which follows current W3C recommendations.

- 1** Create a new text file to contain the XHTML markup. Save this in the root directory of your site as an HTML/XHTML file. Here we've named it `tabpage.html`.
- 2** Create a new directory under the root and name it **inc**.
- 3** In the **inc** directory, create a new text file that will contain the basic style rules that are recognized by most CSS-aware browsers. Save this file with the name `common.css`.
- 4** In the same directory, create a new text file that will contain more advanced CSS style rules that are only recognized by the latest graphical browsers. Save this file with the name `advanced.css`.
- 5** In the same directory, create a new text file that will contain the base document object constructor JavaScript. Save it with the filename `docobj.js`.

In the same directory, create a new text file that will contain the tab navigation Javascript. Save it with the filename `tabs.js`.

CREATING THE XHTML BASE

We've used XHTML instead of HTML for the markup for this project. (If you're unsure about XHTML markup conventions, refer back to Project 4, "Creating a Complex Layout with CSS-P.")

- 1 Create the **DOCTYPE** declaration for XHTML 1.0 Transitional in your markup document `tabpage.html`.
- 2 In the **head** section of the document, `tabpage.html`, insert the tags necessary to link the external CSS files.

Note that we created two external stylesheets in the previous step: `common.css` and `advanced.css`. Because `advanced.css` contains rules that will only apply to later browsers, including some rules that may cause problems in version 4 browsers, we'll use the **@import** method of linking an external stylesheet. For `common.css` we will use the **<link>** tag.

Note the order: **@import** comes before the **<link>**. This is because any **@import** rule must precede all other style rules specified for a document.

- 3 In the **head** section of the markup document, `tabpage.html`, insert the **<script></script>** tags to link the external JavaScript files.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/2000/REC-xhtml1-
20000126/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;
  ↪charset=iso-8859-1" />
  <title>Project 13: Tabbed interface demo</title>

  <style type="text/css" title="supplement">
  <!--
  @import "inc/advanced.css";
  -->
  </style>
  <link rel="Stylesheet" rev="Stylesheet" href="inc/common.css"
  />
```

Note: As you saw in Project 4, older browsers, such as Netscape 4.x, do not recognize the **@import** rule. We're using this seeming deficiency to our advantage here, to make sure that any CSS style rules that aren't interpreted properly are isolated from Netscape 4.x. All these rules are put in the `advanced.css` stylesheet.

For more information about the **@import** rule, see the W3C specifications at <http://www.w3.org/TR/REC-CSS2/cascade.html#at-import>.

```
<script src="inc/docobj.js" type="text/javascript"></script>
<script src="inc/tabs.js" type="text/javascript"></script>

</head>
```

- 4 In the **body** section of the document, create the markup for the contents.

Arrange the content logically so that it can be easily read even if the browser in use does not recognize CSS or JavaScript. The text inside the tabs becomes the header for each section.

Each of the three table cells that make up a tab have a **class** or **id** designation: the left side table cells have a **class** attribute value of **tableft**; the right side table cells have a **class** value of **tabright**; and the middle section table cells have the id values **tabmiddle1**, **tabmiddle2**, and so on.

Each **div** has a unique **id**.

```

<body>

<div id="tab1">
<table cellpadding="0" cellspacing="0" border="0">
<tr>
<td width="8" height="24" class="tableft"></td>
<td id="tabmiddle1">overview</td>
<td class="tabright"></td>
</tr></table>
</div>

<div id="box1">
<!-- content of the first text box. Omitted for clarity. -->
</div>

<div id="tab2">
<table cellpadding="0" cellspacing="0" border="0">
<tr>
<td width="8" height="24" class="tableft"></td>
<td id="tabmiddle2">background</td>
<td class="tabright"></td>
</tr></table>
</div>

<div id="box2">
<!-- content of the second text box. Omitted for clarity. -->
</div>

<div id="tab3">
<table cellpadding="0" cellspacing="0" border="0">
<tr>
<td width="8" height="24" class="tableft"></td>
<td id="tabmiddle3">best recipes</td>
<td class="tabright"></td>
</tr></table>
</div>

<div id="box3">
<!-- content of the third text box. Omitted for clarity. -->
</div>

<div id="tab4">
<tr>
<td width="8" height="24" class="tableft"></td>
<td id="tabmiddle4">etcetera</td>
<td class="tabright"></td>
</tr></table>
</div>

```

CREATING THE MAIN CSS STYLESHEET (common.css)

The stylesheet, `common.css`, holds all the CSS style rules that are recognized by older browsers—as well as the newer browsers.

- 1 Create the style rules that control the text appearance rules for the basic elements on the page, such as **a** (link) elements, **p** (paragraph) elements, and **h1** (header) elements.

Group the **h1** and **p** selectors together to specify the style rules that they share in common and then specify the unique style rules for each separately. Grouping together selectors to specify any common style rules together is a good way of streamlining your stylesheets—and also makes it easier to change the styles for these elements at once if necessary.

Group the ID selectors, **#tabmiddle1**, **#tabmiddle2**, **#tabmiddle3**, **#tabmiddle4**, and **#tabmiddle5**, together and specify the styles rules for them all at once.

Note: Because the **div** elements that act as text boxes on this page and the tabs are precisely sized in pixels, specify the **font** and **line-height** sizes in pixels also rather than in relative units such as `em` or percent.

```
a {
    outline: none;
}

a:link {
    text-decoration: none;
    color: #000000;
}

a:visited {
    text-decoration: none;
    color: #000000;
}

a:hover {
    text-decoration: underline;
}

h1, p {
    background: transparent;
    color: #000000;
    font-family: verdana, helvetica, sans-serif;
    margin-left: 3em;
    margin-right: 3em;
}

h1 {
    font-size: 13px;
    margin-bottom: 0px;
}

p {
    font-size: 11px;
    line-height: 14px;
}

#tabmiddle1, #tabmiddle2, #tabmiddle3, #tabmiddle4, #tabmiddle5 {
    font-family: verdana, helvetica, sans-serif;
    font-size: 16px;
    font-weight: bold;
}
```

CREATING THE SUPPLEMENTAL CSS STYLE RULES (advanced.css)

The stylesheet, `advanced.css`, contains the style rules that are recognized only by browsers that recognize `@import`. It contains all of the positioning rules for the page, as well as some rules that override those in `common.css`.

- 1 Create new **margin: property** style rules for the **p** and **h1** elements.

In the `common.css` stylesheet, we specified `3em` **margin** values for these, but on our “real” layout we want `0px` margins. Therefore, we’ll override the rules previously specified in `common.css` by creating new rules that are declared **!important**.

```
h1, p {
  margin: 0px !important;
}
```

- 2 Create the style rules for the **.tableft** and **.tabright** class selectors that specify the **background-image** property rules for each.

```
.tableft {
  background-image: url(../images/tableft.gif);
}

.tabright {
  background-image: url(../images/tabright.gif);
}
```

- 3 Create the style rules for the **#tabmiddle1**, **#tabmiddle2**, **#tabmiddle3**, **#tabmiddle4**, and **#tabmiddle5** ID selectors.

These are the middle table cells of each tab table. Note that the **font-size: 12px** rule is specified as **!important**. This overrides the **font-size: 18px** rule set in `common.css`.

```
#tabmiddle1, #tabmiddle2, #tabmiddle3, #tabmiddle4,
#tabmiddle5 {
  background-image: url(../images/middle.gif);
  text-align: center;
  font-size: 12px !important;
  font-weight: bold;
  color: #999999;
  cursor: arrow;
}
```

- 4 Create the style rules for the **#tab1**, **#tab2**, **#tab3**, **#tab4**, and **#tab5** **div** elements.

These are the **div** elements that hold the tab tables. Note that all the **div** elements are specified as **position:absolute**. The initial **z-index** is set to 0.

```
#tab1, #tab2, #tab3, #tab4, #tab5 {
    position:absolute;
    height: 24px;
    top: 77px;
    z-index: 0;
    background-color: #cccccc;
    color: #999999;
}

#tab1 {
    left: 60px;
}

#tab2 {
    left: 140px;
    z-index: 0;
}

#tab3 {
    left: 235px;
}

#tab4 {
    left: 335px;
}

#tab5 {
    left: 410px;
}
```

- 5 Create the style rules for the ID selectors **#box1**, **#box2**, **#box3**, **#box4** and **#box5**.

These are the **div** elements that contain the main content. The content boxes all have a one-pixel wide black **border** and are absolutely positioned like the tab boxes. The **height** and **width** are specified in absolute (pixel) values.

```
#box1, #box2, #box3, #box4, #box5 {
    position: absolute;
    top: 100px;
    left: 50px;
    width: 415px;
    height: 320px;
    z-index: 0;
    padding: 10px;
    background-color: #ffffff;
    border: 1px solid #000000;
}
```

CREATING THE BASIC DOCUMENT OBJECT CREATION SCRIPT (docobj.js)

In this section we'll create a document object construction script, which is a slight variation on the one we have used in the previous JavaScript projects. This script is more modular given that the document object constructor is defined as two functions. These functions can then be called by subsequent functions to create the necessary document object strings.

- 1 Open the file, `docobj.js`.
- 2 Create the function, `getDocObj()`. This function returns the text string that points out the correct document object of an element, depending on which browser is in use. Two arguments can be passed to the function: `elem`, or the `id` of the element in question, and `parent`, the `id` of the parent element if the element is nested.

If the browser in use is Netscape 4.x and it recognizes the `document.layers` property, the script then looks to see if the argument, `parent`, is specified. If it is, then the string of `"document."+parent+".document."+elem` is returned. If `parent` is not specified, then the string of `"document." + elem` is returned.

Otherwise, if the browser in use is Internet Explorer 4.x and up and it recognizes the `document.all` method, the string of `"document.all." + elem` is returned.

Otherwise, if the browser in use is W3C-DOM compliant (and is not Internet Explorer), the string of `"document.getElementById('+elem+')"` is returned.

- 3 Create the function, `getStyleObj()`. This function returns the string that points to the `style` property of an element, depending on which browser is in use. This function is the same as `getDocObj()` except that it adds the `".style"` string to the base document object string—in the case of Internet Explorer 4.x and up and W3C-DOM-compliant browsers. Note that the string for Netscape 4.x is identical in both functions.

```
function getDocObj(elem,parent) {
    if (document.layers) {
        if (parent) {
            return "document."+parent+".document."+elem;
        } else {
            return "document."+elem;
        }
    } else if (document.all) {
        return "document.all."+ elem;
    } else if (document.getElementById) {
        return "document.getElementById('+elem+')";
    }
}
```

```
function getStyleObj(elem,parent) {
    if (document.layers) {
        if (parent) {
            return "document."+parent+".document."+elem;
        } else {
            return "document."+elem + ".style";
        }
    }
}
```

continues

continued

```
    }  
  } else if (document.all) {  
    return "document.all."+elem + ".style";  
  } else if (document.getElementById) {  
    return "document.getElementById('"+elem+"').style";  
  }  
}
```

THE CROSS-BROWSER DOCUMENT OBJECT CONSTRUCTOR REVISITED

To use the `getDocObj()` and `getStyleObj()` functions in other functions, use the `eval()` method to evaluate the results. This will obtain the actual document object. For example:

```
eval(getDocObj(elementid));
```

will return the document object of the element with the `id` specified, while

```
getDocObj(elementid);
```

will only return the text string.

The advantage of putting the document object constructors into functions is that it modularizes your code and reduces the typing you have to do when defining other functions. Experienced programmers in any object-oriented language like to create a library of custom reusable objects and functions like this.

Let's take an example from the previous project—the `placeIt()` function.

This is the old code. We've used this method so far so that you can see exactly how a custom document object is assembled:

```
function placeIt(elem,leftPos,topPos) {  
  docObj = eval(doc + elem + sty);  
  if (n4 || n6) {  
    docObj.left = leftPos;  
    docObj.top= topPos;  
  } else if (ie) {  
    docObj.pixelLeft = leftPos;
```

```
    docObj.pixelTop = topPos;  
  }  
}
```

Now this is the new code that uses the `getStyleObj()` function (the changed code is highlighted):

```
function placeIt(elem,leftPos,topPos) {  
  docObj = eval(getStyleObj(elem));  
  if (n4 || n6) {  
    docObj.left = leftPos;  
    docObj.top= topPos;  
  } else if (ie) {  
    docObj.pixelLeft = leftPos;  
    docObj.pixelTop = topPos;  
  }  
}
```

As your understanding of JavaScript increases, try to build up your own library of reusable objects and functions too.

The document object text string for Netscape 4.x is not really needed for this project because none of the JavaScript functions that follow are intended to work in that browser. The `getDocObj()` function also is not used in the subsequent functions. However, I have presented the complete `getDocObj()` and `getStyleObj()` functions here because they are very useful when you have to create cross-browser scripts.

CREATING THE TAB NAVIGATION SCRIPT (tabs.js)

The file, tabs.js, contains the JavaScript that makes the tab navigation work.

The tab navigation script basically does two things: it changes the stacking order or **z-index** of the elements and changes the background color and textcolor of the tab.

This is done by simply changing the style property values of elements.

- 1 Open the file tabs.js.
- 2 Define the arrays, **tabArray** and **boxArray**. These arrays hold the style object text strings that are obtained with the **getStyleObj()** function, which will be used in the tab navigation functions.

The **tabArray()** array holds the style property strings for the **tab1** through **tab5 div** elements. The **tabTextArray()** array holds the style property strings for the **tabmiddle1** through **tabmiddle5 td** elements. The **boxArray()** array holds the style property text strings for the **box1** through **box5 div** elements.

It's important to put the array values in the proper order because the functions we'll define later will point to the **index** in each array.

Note: The data in array position 0 for each array has been set to **null**. This is just a matter of convenience—all arrays start at index number 0, but I always tend to forget this because I like to start counting at 1. But remember that the total number of items in the array is 6.

Defining text strings as array values can make writing scripts that require repeating or looping statements much more efficient.

```

/* The tab navigation script. Always use in conjunction with
docobj.js! */

// Set up array of tab element style property strings

var tabArray = new Array(6);
tabArray[0] = null;
tabArray[1] = getStyleObj('tab1');
tabArray[2] = getStyleObj('tab2');
tabArray[3] = getStyleObj('tab3');
tabArray[4] = getStyleObj('tab4');
tabArray[5] = getStyleObj('tab5');

var tabTextArray = new Array(6);
tabTextArray[0] = null;
tabTextArray[1] = getStyleObj('tabmiddle1');
tabTextArray[2] = getStyleObj('tabmiddle2');
tabTextArray[3] = getStyleObj('tabmiddle3');
tabTextArray[4] = getStyleObj('tabmiddle4');
tabTextArray[5] = getStyleObj('tabmiddle5');

// Set up array of text box element style property strings
var boxArray = new Array(6);
boxArray[0] = null;
boxArray[1] = getStyleObj('box1');
boxArray[2] = getStyleObj('box2');
boxArray[3] = getStyleObj('box3');
boxArray[4] = getStyleObj('box4');
boxArray[5] = getStyleObj('box5');

```

- 3 Define some global variables that are used in the functions.

The variable, **active**, is used as a holder to see which element is the one on top.

The other variables hold color values and are used to color the tabs and tab text.

- 4 Start to define the function, **tabcolor()**. This function sets the appearance of the tab.

Three arguments are passed to the function: **tabnum**, or the number of the tab; **color1**, which is the background color of the tab; and **color2**, which is the text color of the tab.

- 5 Continue to define function, **tabcolor()**. Declare the value of the local variable **tab** to be the evaluated result of the value in the **tabArray[tabnum]** array element. For example, if the value of **tabnum** is 3, the value of **tabArray[tabnum]** is the result returned by **getStyleObj('tab3')**.

Declare the value of the local variable, **tabtext**, to be the evaluated result of the value in the **tabTextArray[textnum]** array element. If the value of **tabnum** is 3, the value of **tabTextArray[tabnum]** is the result returned by **getStyleObj('tabmiddle3')**.

- 6 Continue to define the function, **tabcolor()**, by setting the **backgroundColor** property of **tab** to the value of **color1** and the **color** property of **tabtext** to the value of **color2**.
- 7 Finally, define the code that changes the appearance of the cursor. Because the **tabcolor()** function will be used with mouse events associated with a **td** element, the cursor will not change its appearance by default—unlike mouse events that occur when associated with an **a** link element. Therefore, we'll

```
var active = null;
var activebgcolor = "#993399";
var activetextcolor = "#000000";

var inactivebgcolor = "#cccccc";
var inactivetextcolor = "#999999";

var overbgcolor = "#cc99cc";
var overtextcolor = "#ffcc99";
```

```
function tabcolor(tabnum,color1,color2) {
    var tab = eval(tabArray[tabnum]);
    var tabtext = eval(tabTextArray[tabnum]);
    tab.backgroundColor = color1;
    tabtext.color = color2;
```

change the cursor style to the “pointing hand.”

Because the syntax is different for Internet Explorer and Netscape 6/Mozilla, we have a branching statement: the **cursor** style is set to **"hand"** in IE and **"pointer"** in Netscape 6.

8 Define the function, **choosebox()**.

This is the core function that sets the z-index of the content **divs** and tabs. It also calls the **tabcolor()** function.

One argument passed to it is **tabnum**, which contains the appropriate array index number for the three arrays we defined previously.

First, check to see if the browser in use supports **document.all** or **document.getElementById**. If not, the function will not continue.

Next, check to see if the value of the variable **active** is **null**. If it is not, then that means that one of the layers is “active”—or was the last selected one. Therefore, the ‘active’ elements are reset to their ‘inactive’ states.

Assign the local variables **activetablayer**, **activetabtext**, and **activeboxlayer** the evaluated values of the **tabArray[active]**, **tabTextArray[active]**, and **boxArray[active]** array elements, respectively. Then set the **zIndex** property value of **activetablayer** and **activeboxlayer** to **0** and reset the **backgroundColor** and **color** values of the **tabTextArray[active]** element by calling the **tabcolor()** function, passing **active**, **inactivebgcolor**, and **inactivetextcolor** as the arguments.

continued

```

if (document.all) {
    tabtext.cursor = 'hand';
} else {
    tabtext.cursor = 'pointer';
}
}

```

CONVERTING HYPHENATED CSS STYLE PROPERTIES TO JAVASCRIPT

Most CSS style properties can be used as-is in JavaScript, with the exception of style properties with hyphens such as **background-color**, **z-index**, and **font-family**. We can’t use hyphens for variable, function, custom defined property, or method names because the hyphen can be interpreted as a **minus** sign.

Fortunately, the rule for “converting” such CSS property names to JavaScript is simple: Take away the hyphen and capitalize the first letter of the word after the hyphen. So **background-color** becomes **backgroundColor**, **z-index** becomes **zIndex**, and so on. (Remember that while CSS is case-insensitive, JavaScript is case sensitive!)

9 Continue to define the function, **choosebox()**.

This section of the function sets the newly selected elements to the ‘active’ state.

Assign the local variables, **tablayer**, **tabtext**, and **boxlayer**, the evaluated values of the **tabArray[num]**, **tabTextArray[num]**, and **boxArray[num]** array elements, respectively. Then set the **zIndex** property value of **tablayer** and **boxlayer** to 10 and reset the **backgroundColor** and **color** values of the **tabTextArray[tabnum]** element by calling the **tabcolor()** function, passing **tabnum**, **activebgcolor**, and **activetextcolor** as the arguments.

Finally, set the new value of **active** as **tabnum**. This makes the currently selected elements the **active** elements.

```
// the central tab navigation function

function choosebox(tabnum) {
    if (active) {
        var activetablayer = eval(tabArray[active]);
        var activetabtext = eval(tabTextArray[active]);
        var activeboxlayer = eval(boxArray[active]);
        activetablayer.zIndex = 0;
        activeboxlayer.zIndex = 0;
        tabcolor(active,inactivebgcolor, inactivetextcolor);
    }

    tablayer = eval(tabArray[tabnum]);
    tabtext = eval(tabTextArray[tabnum]);
    boxlayer = eval(boxArray[tabnum]);
    tablayer.zIndex = 11;
    boxlayer.zIndex = 10;
    tabcolor(tabnum,activebgcolor, activetextcolor);
    active = tabnum;
}
}
```

10 Define the function, **tabover()**.

The **tabover()** function simply calls the **tabcolor()** function if the value of **tabnum** does not equal the value of **active** (in other words, if the chosen element is not the one currently on top or “active”). The values of the variables, **overbgcolor** and **overtextcolor**, are passed to **tabcolor()** as arguments. The **tabover()** function is invoked with the **mouseover** event.

```
function tabover(tabnum) {
    if (tabnum != active) {
        tabcolor(tabnum,overbgcolor,overtextcolor);
    }
}
}
```

11 Define the function **tabout()**.

The **tabout()** is identical to the **tabover()** function, except that it passes the values of the variables, **inactivebgcolor** and **inactivetextcolor**, as arguments. This function is called with the **mouseout** event.

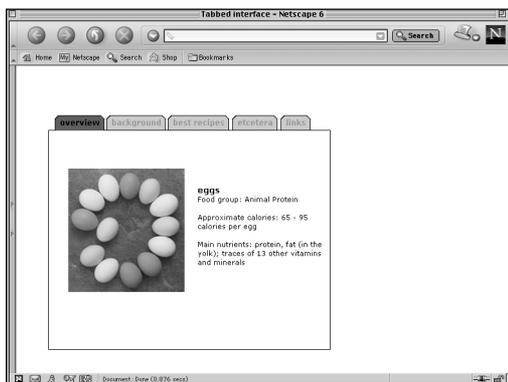
```
function tabout(tabnum) {
    if (tabnum != active) {
        tabcolor(tabnum,inactivebgcolor,inactivetextcolor);
    }
}
}
```

MODIFYING THE XHTML MARKUP TO CALL THE FUNCTIONS

To activate the JavaScript, we will adjust the XHTML markup to call the various functions we've defined.

- 1 Insert an **onload** event call in the **<body>** tag that calls the **choosebox()** function, with **1** passed as the argument.

```
<body onload="choosebox(1)">
```



When the page first loads, the first tab (tab1) is set to "active," and the first content box (box1) is on top.

- 2 Insert the **onmouseover**, **onmouseout**, and **onclick** event calls from the **div** elements **tab1**, **tab2**, **tab3**, **tab4**, and **tab5**. The **tabover()** function is called **onmouseover**; the **tabout()** function is called **onmouseout**; and the **choosebox()** function is called **onclick**. The appropriate value for the tabnum argument is passed to the functions: for **tab1** it's **1**, for **tab2** it's **2**, and so on. The added event calls are highlighted in the code.

```
<div id="tab1" onmouseover="tabover(1)"
onmouseout="tabout(1)"
onclick="choosebox(1)">
<table summary="" cellpadding="0"
cellspacing="0" border="0">
<tr>
<td width="8" height="24" class="tableleft">
</td>
<td id="tabmiddle1">overview</td>
<td class="tabright"></td>
</tr>
</table>
</div>
<div id="tab2" onmouseover="tabover(2)"
onmouseout="tabout(2)"
onclick="choosebox(2)">
<table summary="" cellpadding="0"
cellspacing="0" border="0">
<tr>
<td width="8" height="24" class="tableleft">
</td>
<td id="tabmiddle1">overview</td>
<td class="tabright"></td>
</tr>
</table>
</div>
```

continues

continued

```
</table>

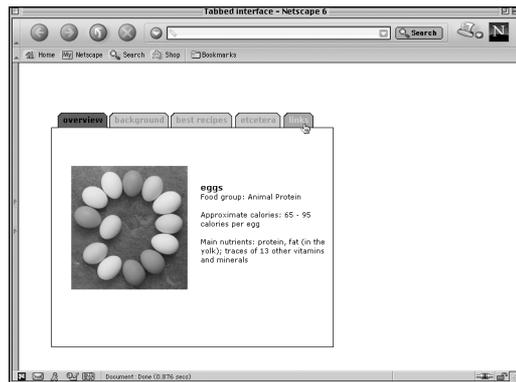
</div>

<div id="tab3" onmouseover="tabover(3)"
➔onmouseout="about(3)" onclick="choosebox(3)">
<table summary="" cellpadding="0" cellspacing="0"
➔border="0">
<tr>
<td width="8" height="24" class="tableleft"></td>
<td id="tabmiddle1">overview</td>
<td class="tabright"></td>
</tr>
</table>

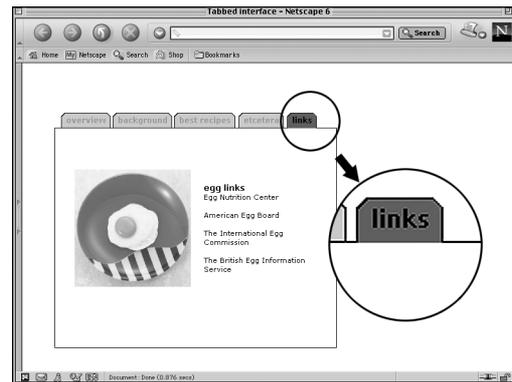
</div>

<div id="tab1" onmouseover="tabover(1)"
➔onmouseout="about(1)" onclick="choosebox(1)">
<table summary="" cellpadding="0" cellspacing="0"
➔border="0">
<tr>
<td width="8" height="24" class="tableleft">
</td>
<td id="tabmiddle1">overview</td>
<td class="tabright"></td>
</tr>
</table>

</div>
```



The user mouses over another tab, and the text and background colors change.



The user clicks the tab, and the new content box appears. Because the whole page is already loaded, the change occurs right away. The tab's appearance changes to the "active" state. Note how the tab is stacked on top of the content box.

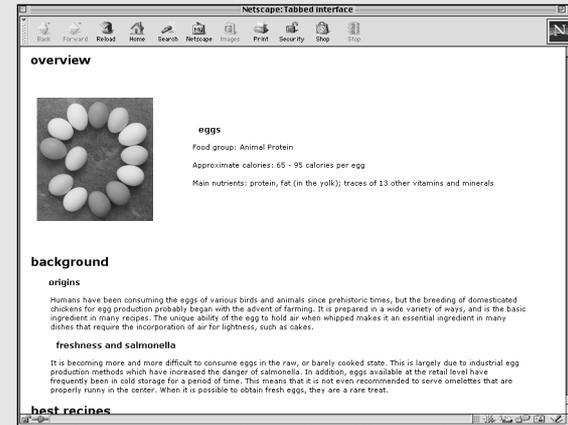
HOW IT WORKS: GRACEFULLY DEGRADING PAGES WITH CSS

In this project, we have dealt with the issue of creating a page that degrades gracefully in older browsers in two ways. The first is with a planned use of the varying support for CSS rules.

To review, our main goal for this project was to create a page that is readable to all and reasonably attractive to users of browsers with partial CSS support—while still avoiding the use of styling tags such as `` within the markup itself.

The CSS for this page has been separated into two documents. The first one contains the rules that will be recognized by Netscape 4.x. This is so that the page looks reasonably attractive in that browser. Generally speaking, these are the style rules for font appearance as well as background and foreground colors. Some basic margin property values are also included.

The second stylesheet, `advanced.css`, contains the rules that are not well supported by Netscape 4.x or any other browser with only rudimentary support for CSS. Because `advanced.css` is associated with the base XHTML document using the `@import` property, the whole stylesheet will be ignored by those browsers that don't support `@import`. Some of the rules in `advanced.css` override the ones in `common.css` by declaring them to be `!important`. It's always important to remember to arrange the markup content in a way that is logical for people who are using non-CSS enabled browsers.



How the page looks in Netscape 4.x, which only recognizes the style rules included in `common.css`. Note how the "header" text for each section is actually the text contained in the tab tables. Here this text is set to a large size (font-size: 18px); this is overridden in the `advanced.css` stylesheet with `!important` (font-size: 12px !important).

HOW IT WORKS: GRACEFULLY DEGRADING PAGES AND JAVASCRIPT

In the previous projects, most of the functions we created had a number of `if...else` statements that specified different code for different browsers. In this project however, there are only two browser-specific `if...else` statements in the whole `tabs.js` script. The first one, in the `tabcolor()` function, deals with the fact that the syntax for cursor styles for Internet Explorer and Netscape 6 differ. The second one, at the beginning of the `choosebox()` function, specifies that the function will only execute if the browser in use recognizes `document.all` or `document.getElementById`.

The `tabover()` and `tabout()` functions, on the other hand, don't have any `if...else` statements that "sniff" the browser out. Instead, the functions are

called from within `div` elements rather than from `a` (link) elements. Modern browsers allow for JavaScript events to be called from any element, while it's only possible to call JavaScript events from a limited set of elements in older browsers. Netscape 4.x, for example, will simply ignore the `onmouseover`, `onmouseout`, and `onclick` event calls in the `div` elements.

When creating a "gracefully degrading" site, the visitor should never get the impression that they are somehow getting a lesser version. Ideally, he should never know what he is missing—unless, of course, he revisits the site with another browser!