

CHAPTER 5

CREATING NAVIGATION

In this chapter

- Introducing web navigation
- Creating links
- Controlling CSS link states
- Mastering the cascade
- Looking at links and accessibility
- Examining a JavaScript alternative to pop-ups
- Creating CSS-based rollovers

Introduction to web navigation

The primary concern of most websites is the provision of information. The ability to enable nonlinear navigation via the use of links is one of the main things that sets the Web apart from other media. But without organized, coherent, and usable navigation, even a site with the most amazing content will fail.

During this chapter, we'll work through how to create various types of navigation. Instead of relying on large numbers of graphics and clunky JavaScript, we'll create rollovers that are composed of nothing more than simple HTML lists and a little CSS. And rather than using pop-up windows to display large graphics when a thumbnail image is clicked, we'll cover how to do everything on a single page.

Navigation types

There are essentially three types of navigation online:

- **Inline navigation:** General links within web page content areas
- **Site navigation:** The primary navigation area of a website, commonly referred to as a **navigation bar**
- **Search-based navigation:** A search box that enables you to search a site via terms you input yourself

Although I've separated navigation into these three distinct categories, lines blur, and not every site includes all the different types of navigation. Also, various designers call each navigation type something different, and there's no official name in each case, so in the following sections, we'll expand a little on each type.

Inline navigation

Inline navigation used to be the primary way of navigating the Web, which, many moons ago, largely consisted of technical documentation. Oddly, inline navigation—links within a web page's body copy—has become increasingly rare. Perhaps this is due to the increasing popularity of visually oriented web design tools, leading designers to concentrate more on visuals than usability. Maybe it's because designers have collectively forgotten that links can be made anywhere and not just in navigation bars. In any case, links—inline links in particular—are the main thing that differentiates the Web from other media, making it unique. For instance, you can make specific words within a document link directly to related content. A great example of this is Wikipedia (www.wikipedia.org), the free encyclopedia.



The screenshot shows the Wikipedia Main Page layout. At the top left is the Wikipedia logo (a globe) and the text "WIKIPEDIA The Free Encyclopedia". To the right of the logo is a navigation bar with links: [Main Page](#), [Recent changes](#), [View source](#), [Page history](#), [Printable version](#), and [Disclaimers](#). Further right is a search box with "Go" and "Search" buttons, and the text "Not logged in", [Log in](#), and [Help](#).

Below the logo is a vertical sidebar with links: [Main Page](#), [Recent changes](#), [Random page](#), [Current events](#), [Community Portal](#), [View source](#), [Discuss this page](#), [Page history](#), [What links here](#), [Related changes](#), [Special pages](#), [Contact us](#), and [Donations](#).

The main content area starts with the heading "Main Page" and the subtext "From Wikipedia, the free encyclopedia." Below this is a welcome message: "Welcome to Wikipedia! We are building an open-content encyclopedia in many languages. We started in January 2001 and are now working on 270334 articles in the English version. Learn how to edit pages, experiment in the sandbox, and visit our Community Portal to find out how you can contribute to Wikipedia." To the right of this message are links: [Browse by topic](#), [Other languages](#), [Sister projects](#), and [Text only](#) / [No tables](#).

The main content area is divided into two columns. The left column is titled "Featured article" and features an image of crash test dummies. The text describes them as full-scale replicas of human beings used to simulate behavior in a vehicle mishap. Below the image is a link: "Recently featured: [Fancy cancel](#) – Buckinghamshire – Comet". At the bottom of this column is a link: [More featured articles...](#)

The right column is titled "In the news" and features a portrait of Chen Shui-bian. Below the portrait is a list of news items:

- [Chen Shui-bian](#) is sworn in to his second term as [President of the Republic of China](#).
- A member of [Fathers 4 Justice](#) hits UK Prime Minister [Tony Blair](#) with a flour-filled [condom](#) in the [House of Commons](#) as a sign of [protest](#).
- [Manmohan Singh](#) is invited to become [Prime Minister of India](#) after [Sonia Gandhi](#) declines the post.

At the bottom of this column is a link: [Recent deaths](#) | [More current events...](#)

Site navigation

Wikipedia showcases navigation types other than inline. To the left, underneath the logo, is a navigation bar that is present on every page of the site, allowing users to quickly access each section. This kind of thing is essential for most websites—long gone are the days when users were happy to keep returning to a homepage to navigate to new content. (Actually, to be fair, they weren't ever too happy about it, but there you go.)

As Wikipedia proves, just because you have a global navigation bar, that doesn't mean you should skimp on inline navigation. In recent times, I've seen a rash of sites that say things like Thank you for visiting our website. If you have any questions, you can contact us by clicking the contact details link on our navigation bar. Quite frankly, this is bizarre. A better solution is to say Thank you for visiting our website. If you have any questions, please contact us, and to turn contact us into a link to the contact details page. This might seem like common sense, but not every web designer thinks in this way.

Search-based navigation

Wikipedia has a search box at the top-right corner of each page. It's said there are two types of web users: those who eschew search boxes and those who head straight for them. The thing is, search boxes are not always needed, despite the claims of middle managers the world over. Indeed, most sites get by with well-structured and coherent navigation.

However, sites sometimes grow very large (typically those that are heavy on information and that have hundreds or thousands of pages, such as technical repositories, review archives, or large online stores). In such cases, it's often not feasible to use standard navigation elements to access information. Attempting to do so leads to users getting lost trying to navigate a huge navigation "tree."

Unlike other types of navigation, search boxes aren't entirely straightforward to set up yourself, and they require server-side scripting for their functionality. A quick trawl through a search engine provides many options. One of my favorites, which provides a simple, free service for sites with fewer than 500 pages (although the free version also automatically places some advertising within the results page), is Atomz Search. If you're one of the many web designers who aren't particularly technically minded, you'll be interested to hear that Atomz takes only a few minutes to set up. For more details, visit the Atomz website at www.atomz.com.

Anchors (creating links)

With the exception of search boxes, which are forms based and driven by server-side scripting, online navigation relies on **anchor elements**. In its simplest form, an anchor tag looks like this:

```
<a href="http://www.friendsofed.com/">A link to the friends of ED website</a>
```

By placing a trailing slash in this type of URL, you make only one call to the server instead of two. Also, some incorrectly configured Apache servers generate a "File not found" error if the trailing slash is omitted.

The href attribute value is the URL of the destination document, which is often another HTML file, but can in fact be any file type (MP3, PDF, JPEG, and so on). If the browser can display the document type (either directly or via a plug-in), it does so; otherwise, it downloads the file (or brings up some kind of download prompt).

Never omit end tags when working with links. Omitting is not only shoddy and invalid XHTML, but most browsers then make all subsequent content on the page into a link!

There are three ways of linking to a file: absolute links, relative links, and root-relative links. We'll cover these in the sections that follow, and you'll see how to create internal page links, style link states in CSS, and work with links and images. We'll also discuss enhanced link accessibility and usability, and link targeting.

Absolute links

The preceding example shows an **absolute link**, sometimes called a **full URL**, which is typically used when linking to external files (i.e., other websites). This type of link provides the entire path to a destination file, including the file transfer protocol, domain name, any directory names, and the file name itself. A longer example is

```
<a href="http://www.wireviews.com/lyrics/instar.html">Instar lyrics</a>
```

In this case, the file transfer protocol is `http://`, the domain is `wireviews.com`, the directory is `lyrics`, and the file name is `instar.html`.

Depending on how the target site's web server has been set up, you may or may not have to include `www` prior to the domain name when creating this kind of link. Usually it's best to include it, to be on the safe side.

If you're linking to a website's homepage, you can usually leave off the file name, as in the earlier link to the friends of ED site, and the server will automatically pick up the default document (assuming one exists), which can be `index.html`, `default.htm`, `index.php`, `index.asp`, or some other name, depending on the server type. However, adding a trailing slash after the domain is beneficial (such as `http://www.wireviews.com/`).

Relative links

A relative link is one that locates a file in relation to the current document. Taking the Wireviews example, if we were on the `instar.html` page, located inside the `lyrics` directory, and we wanted to link back to the homepage via a relative link, we would write

```
<a href="../index.html">Wireviews homepage</a>
```

The `index.html` file name is preceded by `../`, which tells the web browser to move up one directory prior to looking for `index.html`. Moving in the other direction is done in the same way as with absolute links: by preceding the file name with the path. Therefore, to get from the homepage back to the `instar.html` page we write

```
<a href="lyrics/instar.html">Instar lyrics</a>
```

In some cases, you need to combine both methods. For instance, this website has HTML documents in both the `lyrics` and `reviews` folders. To get from the `instar.html` lyrics page to a review, you have to go up one level, and then down into the relevant directory to locate the file:

```
<a href="../../reviews/alloy.html">Alloy review</a>
```

Root-relative links

Root-relative links work in a similar way to absolute links, but from the root of the website. They ensure you point to the relevant document without your having to type an absolute link or mess around with relative links. For instance, regardless of how many directories deep you are in the Wireviews website, a root-relative link to the homepage always looks like this:

```
<a href="/index.html">Homepage</a>
```

And a link to the `instar.html` page within the `lyrics` directory always looks like this:

```
<a href="/lyrics/instar.html">Instar lyrics</a>
```

The initial forward slash tells the browser to start the path to the file from the root of the current website.

All paths in href attributes must contain forward slashes only. Some software—notably some by Microsoft—both creates and permits backward slashes (i.e., `lyrics\wire\154.html`), but this is nonstandard and does not work in non-Microsoft web browsers.

Internal page links

Along with linking to other documents, it's possible to link to another point in the same web page. This is handy for things like a FAQ (frequently asked questions) list, enabling the visitor to jump directly to an answer and then back to the list of questions.

Creating such a system is simple. For a list of questions, we can have something like this:

```
<ul id="questions">
  <li><a href="#answer1">Question one</a></li>
  <li><a href="#answer2">Question two</a></li>
  <li><a href="#answer3">Question three</a></li>
</ul>
```

Later on in the document, the first two answers might look like this:

```
<p id="answer1">The answer to question 1!</p>
<p><a href="#questions">Back to questions</a></p>

<p id="answer2">The answer to question 2!</p>
<p><a href="#questions">Back to questions</a></p>
```

In each case, the link href value is prefixed by a hash sign (#). When the link is clicked, the web page jumps to the element with the relevant id value. Therefore, clicking the Question one link, which has an href value of `#answer1`, jumps to the paragraph with the id value of

answer1. Clicking the Back to questions link, which has an id value of #questions, jumps back to the list, because the unordered list element has an id of questions.

It's worth bearing in mind that the page only jumps directly to the linked element if there's enough room underneath it. If the target element is at the bottom of the web page, you'll see it plus a browser window height of content above.

Backward compatibility/fragment identifiers

Note that obsolete browsers such as Netscape 4 often don't understand this system when working solely with the id attribute. Instead, you have to use a **fragment identifier**, which is an anchor tag with a name attribute, but no href attribute. For instance, a fragment identifier for the first answer is

```
<p><a id="answer1" name="answer1">Answer 1!</a></p>
```

The reason for “doubling up,” using both the name and id attributes, is because the former is on borrowed time in web specifications, and it should therefore only be used for backward compatibility.

Top of page links

Internal page links are sometimes used to create a Back to top link, which is useful for navigating lengthy pages and returning to the top of the document, which usually houses the navigation. The problem here is that the most common internal linking method actually fails in the majority of web browsers.

```
<a href="#top">Back to top</a>
```

You've likely seen the previous sort of link countless times, but unless you're using Internet Explorer for Windows, it's as dead as a dodo. There are various workarounds, though, one of which is to include a fragment identifier at the top of the document. At the foot of the web page we have the Back to top link shown previously, and at the top of the web page we place the fragment identifier:

```
<a id="top" name="top"></a>
```

This technique isn't without its problems, though. Some browsers totally ignore empty elements such as this (some web designers therefore populate the element with a single space); it's tricky to get the element right at the top of the page and not to interfere with subsequent content; and, if you're working with XHTML Strict, it's not valid to have an inline element on its own, outside of a block element, such as p or div. A potential solution is to nest the fragment identifier within a block element, and then style the block element to sit at the top-left of the web page.

HTML:

```
<div id="topOfPageAnchor">
  <a id="top" name="top"> </a></div>
```

CSS:

```
div#topOfPageAnchor {
  position: absolute;
  top: 0;
  left: 0;
  height: 0;
}
```

Setting the div's height to 0 means it takes up no space and is therefore not displayed; setting its positioning to `absolute` means it's outside the normal flow of the document, so it doesn't affect subsequent page content. You can test this by setting the background color of a following element to something vivid—it should sit tight to the edge of the browser window edges.

Another common solution to the Top of page link problem is to use JavaScript. You should be wary of using JavaScript for essential web page elements, though, because not everyone surfs the Web with JavaScript on (according to estimates, between one in ten and one in twenty turn it off). However, by combining two methods, we can create a catchall (well, "catch nearly all") solution:

```
<a href="#top" onclick="javascript: scrollTo(0,0);">Top of page</a>
```

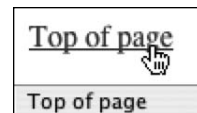
Included is the standard link to `#top`, which works fine in Internet Explorer for Windows. Also included is some very simple JavaScript:

```
onclick="javascript: scrollTo(0,0);"
```

This is pretty self-explanatory: upon an `onclick` event (i.e., when you click the link), the page scrolls to location 0,0, which is the top-left corner.

JavaScript is case sensitive, so ensure that you are careful when writing it. And when working in XHTML, JavaScript event handlers must be all lowercase (so `onclick`, not `onClick`).

As a slight extension to this, it's possible to add some JavaScript that upon a `mouseover` event (the cursor hovering over the link) updates the status text, and upon the `mouseout` event (the cursor moving off the link) returns the status to a blank state.



```
<a href="#top" onclick="javascript: scrollTo(0,0);"
  ⤴onmouseover="window.status='Return to the top of the page'; return true;"
  ⤴onmouseout="window.status='';">Top</a>
```


Again, the JavaScript is pretty obvious and can be repurposed for any link. Note that you need to be consistent with regard to quote marks:

```
onmouseover="window.status='Return to the top of the page'; return true;"
```

Here, the `onmouseover` value is enclosed in double quotes and the nested value in single quotes. Using double quotes on both breaks the script, terminating it early. (It's possible to swap the use of double and single quotes. As mentioned, just be consistent.) Likewise, the `onmouseout` `window.status` value is simply nothing within two single quotes. It is *not* a single double quote!

Link states

By default, links are displayed underlined and in blue when viewed in a web browser. However, links have four states, and their visual appearance varies depending on the current state of the link. The four states are

- **link**: The link's standard state, before any action has taken place
- **visited**: The link's state after having been clicked
- **hover**: The link's state while the mouse cursor is over it
- **active**: The link's state while being clicked

`visited` and `active` also have a default appearance. The former is displayed in purple and the latter in red. Both are underlined.

If every site adhered to this default scheme, it would be easier to find where you've been and where you haven't on the Web. However, most designers prefer to dictate their own color schemes rather than having blue and purple links peppering their designs. In my view, this is fine. Despite what some usability gurus claim, most web users these days probably don't even know what the default link colors are, and so hardly miss them.

In HTML, you can set custom colors for the link, active, and visited states via the `link`, `alink`, and `vlink` attributes of the `body` element. These attributes are deprecated, though, and should be avoided. This is a good thing, because you need to define them in the `body` element of every page of your site, which is a tiresome process—even more so if they later need changing—and, as you might have guessed, it's easier to define link states in CSS.

Defining link states with CSS

CSS has advantages over the obsolete HTML method of defining link states. You gain control over the `hover` state and can do far more than just edit the state colors—although that's what we're going to do first.

The default link state is defined by using a tag selector:

```
a {
  color: #3366cc;
}
```

In this example, all links are turned to a medium blue. Individual states can be defined by using **pseudo-class selectors** (so called because they have the same effect as applying a class, even though no class is applied to the element):

```
a:link {
  color: #3366cc;
}
a:visited {
  color: #666699;
}
a:hover {
  color: #0066ff;
}
a:active {
  color: #cc00ff;
}
```

The difference between “a” and “a:link”

Many designers don’t realize the subtle difference between the selectors `a` and `a:link` in CSS. Essentially, the `a` selector styles all anchors, but `a:link` styles only those that are clickable links (i.e., those that include an `href` attribute) that have not yet been visited. This means that, should you have a site with a number of fragment identifiers, you can use the `a:link` selector to style clickable links only, avoiding styling fragment identifiers, too. (This avoids the problem of fragment identifiers taking on underlines.) However, if you define `a:link` and not `a`, you then need to define the `visited`, `hover`, and `active` states, too, otherwise they will appear in their default appearances.

Correctly ordering link states

The various states have been defined in a very specific order in previous examples: `link`, `visited`, `hover`, `active`. This is because certain states override others, and those “closest” to the link on the web page take precedence.

It makes sense for the link to be a certain color when you hover over it, and then a different color on the active state (when clicked). However, if you put the `hover` and `active` states in the other order (`active`, `hover`), you may not see the active one when the link is clicked. This is because you’re still hovering over the link even when you click it.

A simple way of remembering the state order is to think of the words “love, hate”: link, visited, hover, active.

Editing link styles using CSS

Along with changing link colors, CSS enables you to style links just like any other piece of text. You can define specific fonts; edit padding, margins, and borders; change the font weight and style; and also amend the standard link underline, removing it entirely if you wish (by setting the `text-decoration` property to `none`).

```

a:link {
  color: #3366cc;
  font-weight: bold;
  text-decoration: none;
}

```

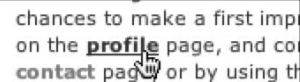
Removing the standard underline is still controversial, even in these enlightened times, and causes endless (and rather tedious) arguments among web designers. My view is that it can be okay to do so, but with some caveats.

If you remove the standard underline, ensure your links stand out from the surrounding copy in some other way. Having your links in the same style and color as other words *and* not underlined is, quite frankly, stupid (that is, unless you don't want people to find the links and click them, or you want them to guess—perhaps for a children's game or educational site).

Just using a different color may not be enough—after all, a significant proportion of the population has some form of color blindness. A commonly quoted figure for color blindness in Western countries is 8%, with the largest affected group being white males (the worldwide figure is lower, at approximately 4%). Therefore, a change of color (to something fairly obvious) *and* a change of font weight to bold often does the trick.

Whatever your choice, be consistent—don't have links change style on different pages of the site. Also, it's useful to reinforce the fact that links are links by bringing back the underline on the hover state.

The example shown to the right is from the Snub Communications website, and it works in the way outlined previously. Links are bold and orange, making them stand out from surrounding text. On the hover state, the link darkens to gray and the standard underline returns. This is achieved by setting `text-decoration` to `underline` in the `a:hover` declaration. Note that even when presented in grayscale, such as in this book, these two states can be distinguished from surrounding text.



Multiple link states: The cascade

A common problem web designers come up against is multiple link styles within a document. I know, I know—I just told you to be consistent. But there are very specific occasions when it's okay to have different link styles. One of these times is for site navigation. Web users are quite happy with navigation bar links differing from standard inline links. Other occasions that spring to mind are for a web page's footer, where links are often displayed in a smaller font than that of the other web page copy, and for areas where background colors are different and the standard link color wouldn't stand out (although in such situations it would perhaps be best to amend either the background or your default link colors).

Some designers apply a class to every link they want to have a style that's not the default, but that method is little better than mucking around with font tags—you end up with loads of inline junk that can't be easily amended at a later date. Instead, by the careful use

of divs (with unique ids) on the web page and contextual selectors in CSS, we can rapidly style links for each section of the web page.

For instance, the following is a basic page content structure. There are three divs: navigation, content, and footer. The first houses an unordered list that forms the basis of the navigation bar. The second is the content area, which has an inline link within a paragraph. The third is the footer, which is commonly used to repeat the navigation bar, albeit in a simpler manner.

```
<div id="navigation">
  <ul>
    <li><a href="index.html">Homepage</a></li>
    <li><a href="products.html">Products</a></li>
    <li><a href="contact_details.html">Contact details</a></li>
  </ul>
</div>

<div id="content">
  <p>Hello there. Our new product is a <a href="banjo.html">fantastic
  ↪banjo</a>!</p>
</div>

<div id="footer">
  <a href="index.html">Homepage</a> | <a href="products.html">Products</a> |
  ↪<a href="contact_details.html">Contact details</a>
</div>
```

This isn't the most feature-packed web page in the world, but it's perfect for this example. Using contextual selectors, we style the link and hover states for links within the navigation area. As mentioned earlier, the hover state returns the default underline that's turned off in the link state:

```
#navigation a:link {
  font-weight: bold;
  font-size: 200%;
  text-decoration: none;
  color: #666666;
}

#navigation a:hover {
  text-decoration: underline;
}
```

We then set the default link style:

```
a:link {
  font-weight: bold;
  color: #aaaaaa;
}
```

And finally, we use a contextual selector to style the footer links (which are commonly the same as those within the general page body, but smaller, hence only defining the font-size property):

```
#footer a {
  font-size: 80%;
}
```

And there we have it: three different link styles on the same page, without messing around with classes.



However, as you can see from the screenshot, if we don't explicitly state a value for a property, the ones from the standard link styles (`a:link`, `a:visited`, `a:hover`, and `a:active`) are used. This explains the color and font weight of the footer links (gray and bold, respectively), despite us not explicitly defining them in the rules created earlier.

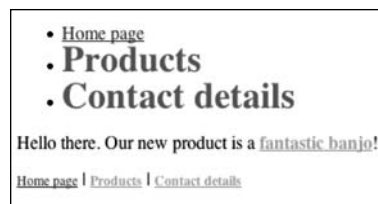
This sort of thing trips up a lot of designers, so take care when working with numerous link styles. For a second example, if you set the active link state background color to red (as in the following CSS), all links on the page—including those in the navigation and footer divs—have a red background when clicked.

```
a:active {
  background-color: red;
}
```

This can be overridden by explicitly setting the background color in the relevant CSS rule(s), such as

```
#navigation a:active {
  background-color: transparent;
}
```

Also remember that if you define `a:link`, but not `a` (as just shown), there is no default for undefined states. Clicking one of the navigation links invokes the `visited` state, which we haven't defined. This means the links appear in their default state: purple, underlined, and in the default size and font. This is most obvious in the visited navigation link, which is now significantly smaller than the other navigation links (see right).



Therefore, in addition to what we've done so far, we must define all states in each context or the `a` selector for each style of link we intend to have on the page (i.e., `#navigation a`, `a`, and so on).

Links and images

Although links are primarily text-based, it's possible to wrap anchor tags around an image, thereby turning it into a link:

```
<a href="a_link.html"></a>
```

Most Windows-based browsers border linked images with whatever link colors have been stated in CSS (or the default colors, if no custom ones have been defined), which looks nasty and can displace other layout elements. There are two ways around this. The most common is to include the `border` attribute in your images and set it to 0, something that many web design applications tend to do by default.

```
<a href="a_link.html"></a>
```

However, this is deprecated, so it's best to use a CSS contextual selector to define images within links as having no border:

```
a img {
border: 0;
}
```

Additionally, you can set a border in CSS for the preceding rule and then use a pseudo-class selector to set a hover state that changes the border color (as we did earlier for links). This can be used (when appropriate) to help users to differentiate those images that are links from those that are not.

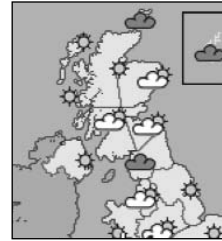
In any case, you must always have usability and accessibility at the back of your mind when working with image-based links. With regard to usability, is the image's function obvious? Plenty of websites use icons instead of straightforward text-based navigation, resulting in frustrated users if the function of each image isn't obvious. People don't want to learn what each icon is for, and they'll soon move on to competing sites. With regard to accessibility, remember that images cannot be increased in size (unless you're using Opera), so if an image-based link has text within it, ensure it's big enough to read easily. Wherever possible, offer a text-based alternative to image-based links, and never omit `alt` attributes. Therefore, the example from earlier becomes

```
<a href="a_link.html"></a>
```

Here, I've placed a subtle hint for you regarding `alt` text content: when linking images, `alt` text should provide an idea of the image's *function*, and not merely what the image actually depicts. For instance, if using a logo to link to your homepage, don't set the `alt` text as logo; instead, set it as Back to homepage.

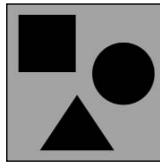
Image maps

Image maps enable you to define multiple links within a single image. There are both server-side and client-side versions of image maps. Server-side image maps are obsolete, and even client-side ones should generally be avoided, because they can cause accessibility problems. However, they occasionally prove useful. For instance, the Met Office website at www.met-office.gov.uk uses image maps to enable users to click a map of the UK and find the weather forecast for their region (see right).



Crown Copyright Met Office

Regardless of the complexity of the image and the defined regions, the method of creating an image map remains the same. Here's the image we're going to turn into an image map:



The image is added to the web page in the usual way, but with the addition of a `usemap` attribute, whose value must be preceded by a hash sign (#).

```

```

The value of the `usemap` attribute must correlate with the name and id values of the associated map element. Note that the name attribute is only required for backward compatibility, whereas the id attribute is mandatory.

```
<map id="shapes" name="shapes">
</map>
```

Because the map element isn't displayed, it can be placed anywhere in the HTML document. On the rare occasions I use image maps, I place the map element under the relevant image (or its containing block element), but other designers group map elements at the end of the document, after all other content.

The map element acts as a container for specifications regarding the map's active areas, which are added as area elements.

```
<map name="shapes">
  <area shape="rect" coords="29,27,173,171" href="square.html"
  alt="Squares page" />
  <area shape="circle" coords="295,175,81" href="circle.html"
  alt="Circles page" />
  <area shape="poly" coords="177,231,269,369,84,369" href="triangle.html"
  alt="Triangles page" />
</map>
```

Each of the preceding area elements has a shape attribute that corresponds to the intended active link area. The first, `rect`, defines a rectangular area, and the `coords` (coordinates) attribute contains two pairs that define the top-left and bottom-right corners in terms of pixel values (which you either take from your original image or guess, should you have amazing pixel-perfect vision). The `circle` value is used to define a circular area. Of the three values within the `coords` attribute, the first two define the horizontal and vertical position of the circle's center, and the third defines the radius. Finally, the `poly` value enables you to define as many coordinate pairs as you wish, enabling you to define active areas for complex and irregular shapes. In this example, three pairs define each corner of the triangle. In the Met Office's map of the UK, the county shapes are complex, thereby necessitating many pairs of values.

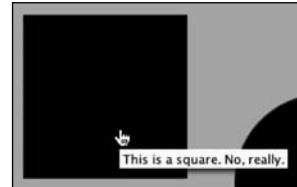
Creating image maps is a notoriously tedious process, and it's one of the few occasions when I strongly advise using a visual web design tool, if you have one handy. However, take care not to overlap defined regions—this is easy to do, and it can cause problems with regard to each link's active area.

Enhanced link accessibility and usability

We've already looked at accessibility and usability concerns during this chapter, so we'll now briefly run through a few attributes that can be used with anchors (and some with area elements) to enhance your web page links.

The title attribute

Regular users of Internet Explorer for Windows may be familiar with its annoying (or helpful, depending on your opinion) habit of popping up alt text as a tooltip. This has encouraged web designers to wrongly fill alt text with explanatory copy for those links that require an explanation, rather than using the alt text for a succinct overview of the image's content or functional purpose. Should you require a pop-up, add a title attribute to your anchor element. The majority of web browsers display its value when the link is hovered over for a couple of seconds (see right), although some older browsers, such as Netscape 4, don't provide this functionality.



```
<area title="This is a square. No, really." shape="rect"
➔coords="29,27,173,171" href="square.html" alt="Squares page" />
```

Behavior varies slightly between browsers. Although Firefox and Opera display the title attribute value when it's added to area elements, Internet Explorer for Windows doesn't (although the Mac version does). Opera goes further, displaying the link location.

Of course, this is kind of a moot point, anyway—if you really need to explain a link (or an area within an image map), you're not doing your job as a designer properly. For reinforcement or clarification, title attribute text can be of use, but it should never be the sole means of relaying information.

Using `accesskey` and `tabindex`

I've bundled these two attributes because they have similar functions—that is, enabling keyboard access to various areas of the web page. Most browsers enable you to use the Tab key to cycle through links, although if you end up on a web page with dozens of links, this can be a soul-destroying experience. (And before you say “So what?”, throw your mouse out of the window and then try using the Web. Many web users cannot use a mouse. You don't have to be severely disabled or elderly to be in such a position either—something as common as repetitive strain injury affects plenty of people, both young and old.)

The `accesskey` attribute can be added to anchor and area elements. It assigns an access key to the link. In tandem with your platform's assigned modifier key (*ALT* for Windows and *CTRL* for Mac), you press the key to highlight or activate the link, depending on how the browser you're using works.

```
<a href="contact_details.html" accesskey="c">Contact us [c]</a>
```

The attribute's value must be a single character, and it's best to stick to standard key characters. Generally, providing at least the main navigation of the site with an access key per link is a good idea. If possible, ensure things remain intuitive by making the access key the first letter of the link's name and, should you use anything more complex, place a full list of access keys used on an accessibility page.

The `tabindex` attribute works in a similar way. Simply define the attribute's value as anything from 0 (which excludes the element from the tabbing order, which can be useful) to 32767 and its place in the tab order is defined, although if you have 32,767 tabbable elements on your web page, you really do need to go back and reread the earlier advice on information architecture (see Chapter 3). Note that tab orders needn't be consecutive, so it's wise to use `tabindex` in steps of ten, so you can later insert extra ones without renumbering everything.

Not all browsers enable tabbing to links, and others require that you amend some preferences to activate this function. It's also worth noting that `tabindex` comes in handy when working with forms, as we'll see in Chapter 10.

Link targeting

From accessibility enhancements to something of an accessibility snafu, **link targeting** is extremely popular, but I don't recommend it. The `target` attribute can be added to anchor and area elements. Its value specifies the name of the window (or frame; see Chapter 9) that the link should be displayed in. A popular value is `_blank`, which opens the target document in a new, blank web browser window.

If you're scratching your head and thinking “So what?” I am well aware that this is common practice. Some argue that opening external links in a new window is beneficial, because it enables users to look at external content and then return to your site. However, what it actually does is take control of the browser *away* from users (after all, if they want to open content in a new window, they can do so using keyboard commands and/or contextual menus). More important, opening documents in new windows breaks

the history path. For many, this might not be a huge issue, but for those navigating the Web via a screen reader, pop-ups are a menace. New content opens up, is deemed to not be of interest, and the Back function is invoked. But this is a new window, with its own *blank* history. Gnashing of teeth ensues.

There are exceptions to this rule—notably when using frames, as you'll see in Chapter 9—but in general it's best to avoid `target`. The W3C agrees: although the `target` attribute is valid when working with XHTML Transitional, it's not when using XHTML Strict.

Links and JavaScript

Although we've used a little JavaScript during this chapter, we're going to work through some slightly more advanced scripts in this section, to show some methods of integrating JavaScript and links to provide web pages with enhanced interactivity and functionality. As mentioned earlier, though, always provide a non-JavaScript backup to essential content for those who choose to surf the Web with JavaScript disabled. In all cases, JavaScript can be added either to external JavaScript files attached to your HTML documents (which is the preferred method; see the section “Attaching favicons and JavaScript” in Chapter 2) or in a script element within the head of the HTML page:

```
<script type="text/javascript">
// 

(script goes here)

// ]]&gt;
&lt;/script&gt;</pre>
</div>
<div data-bbox="205 577 860 611" data-label="Text">
<p>Specifically, we'll look at pop-up windows, swapping images using JavaScript, and toggling div visibility with JavaScript.</p>
</div>
<div data-bbox="179 639 380 664" data-label="Section-Header">
<h2>Pop-up windows</h2>
</div>
<div data-bbox="205 675 860 777" data-label="Text">
<p>I imagine at least one person just threw this book out of the window upon seeing pop-up windows as a heading, but they really should have read on a bit. I'm not going to sit here advocating pop-up windows. They're mostly a serious pain in the backside, especially when automated, and they have the same problems as with the new, blank browser windows that I was moaning about earlier (except many JavaScript pop-ups also remove things like browser controls).</p>
</div>
<div data-bbox="205 791 860 877" data-label="Text">
<p>However, there are some occasions when pop-up windows can be useful. For instance, if you want to provide a user with brief access to terms and conditions without interrupting the checkout process, you might open the terms in a new window. Many portfolio sites also use pop-up windows to display larger versions of images (although we'll later see a much better method of creating an online gallery).</p>
</div>
<div data-bbox="52 938 93 959" data-label="Page-Footer">146</div>
```

Should you require a pop-up window of your very own, the JavaScript is simple:

```
function newWindow()
{
  window.open("location.html");
}
```

And this HTML calls the script using the `onclick` attribute:

```
<a href="#" onclick="newWindow()">Open a new window!</a>
```

Creating a system to open windows with varied URLs requires only slight changes to both script and HTML. The script changes to this:

```
function newWindow(webURL)
{
  window.open(webURL);
}
```

and the HTML changes to this:

```
<a href="#" onclick="newWindow('location_one.html');">Open location one in a new
➤window!</a>
<a href="#" onclick="newWindow('location_two.html');">Open location two in a new
➤window!</a>
```

Note how the target location is now within the single quotes of the `onclick` value. This could be any file name, and the link type can be absolute, relative, or root-relative.

Controlling pop-up windows

By using the script so far, the pop-up windows open in whatever state the browser currently has set (usually that of the most recently closed window), but you may want to control the settings of a pop-up window, along with its name, so it can be targeted. To do so, the script needs to be amended, as follows:

```
function newWindow(webURL)
{
  var newWin = window.open(webURL, "new_window", "toolbar,location,directories,
➤status,menubar,scrollbars,resizable,copyhistory,width=300,height=300");
  newWin.focus();
}
```

The values within the set of quotes that begin `"toolbar, location..."` enable you to set the pop-up window's dimensions and appearance. There must be no white space in the features list, and it must all be on one line. Most of the items are self-explanatory, but

some that may not be are `location`, which defines whether the browser's address bar is visible, and `directories`, which defines whether secondary toolbars such as the links bar are visible. Note that if you specify one or more of these, any you don't specify will be turned off—therefore, you must specify *all* the features you want in the pop-up window.

Now, a word of warning: as alluded to earlier, having control of the web browser wrenched away from them makes some users want to kick a puppy. Therefore, don't use JavaScript to pop up windows without the user knowing that's going to happen. Don't create a site that automatically pops up a window and removes the window controls. And don't start any spurious arguments regarding aesthetics, because there are no real reasons for using pop-up windows in the aforementioned manner, but there are plenty of counterarguments, such as taking control from the user, the general annoyance factor, and so on. Finally, there are methods for forcing pop-up windows to full screen, but that also makes users want to pull teeth. Not only does this take control away from users, but it means the new pop-up window covers up everything else that's onscreen—irritating for the many people who don't surf the Web at full screen, and who work with several browser windows visible at once.

Swapping images using JavaScript

I mentioned earlier that we would explore a better way of creating an online gallery. Instead of using pop-up windows when thumbnails are clicked, we're going to use JavaScript to swap out an image that's on the web page, replacing it with another. Before we begin, we require three full-size images (which have the same dimensions, because it makes things a whole lot easier and the page looks better) and three thumbnail versions.

Exercise: Creating an online gallery

- 1. Add the script.** The JavaScript that drives the swappable images is straightforward:

```
function swapPhoto(photoSRC) {
  document.images.imgPhoto.src = "assets/" + photoSRC;
}
```

As stated earlier, JavaScript should be added to an external JavaScript document linked to your HTML (see the section in Chapter 2 titled “Attaching favicons and JavaScript”) or to a `script` element within the head of your HTML. Be aware of the case-sensitive nature of JavaScript and also the path to the images, which is set here as `assets/`.

- 2. Add the main image.** This requires an `id` attribute (and a `name` attribute, if you're maintaining backward compatibility) that correlates with the one provided in step 1 (`imgPhoto`). Leave off the `height` and/or `width` attributes if your images have varied dimensions.

```

```

3. Add thumbnails. In each case, the `swapPhoto` value is the file name of the image to be loaded. Remember that the path to the images was defined in step 1, so it's not needed here.

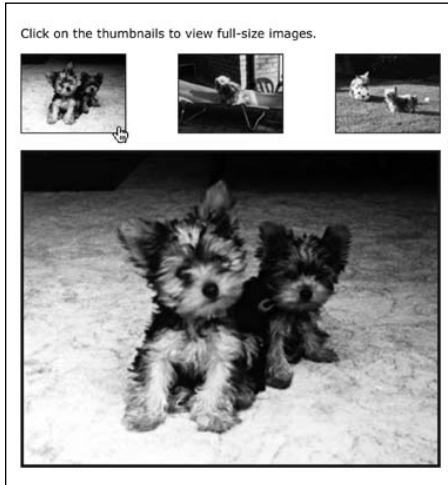
```
<a href="javascript:swapPhoto('dogs_1.jpg')"></a>
```

```
<a href="javascript:swapPhoto('dogs_2.jpg')"></a>
```

```
<a href="javascript:swapPhoto('dogs_3.jpg')"></a>
```

And that's all there is to it. The solution is elegant and doesn't require pop-up windows. Instead, users can see thumbnails on the same page as the main image, making navigation through the portfolio that much easier. Of course, users must have JavaScript turned on for this functionality, so if the images are essential, you should provide an alternate means of accessing them, too. This can be done by using `return false` to prevent the page from being called in JavaScript-enabled browsers. For example, the first link in step 3 can therefore be amended to the following:

```
<a href="dogs_1.html" onclick="swapPhoto('dogs_1.jpg'); return false">
```



Toggling div visibility with JavaScript

The DOM enables you to access and dynamically control various aspects of a web page, and this allows you to use a nifty little trick to toggle the visibility of divs. This trick has numerous uses, from providing a method of hiding “spoiler” content unless someone wants to see it, to various navigation-oriented uses.

Exercise: Setting up a div toggler

- 1. Add the script.** The script in this case can be copied verbatim and requires no editing. Again, if you want to use the script on multiple pages, it's best to add it to your external JavaScript document rather than the head of any web pages. Unlike previous JavaScript examples in this book, this method is not backward compatible; therefore, it doesn't work in obsolete web browsers.

```
function swap(targetId){
    if (document.getElementById)
    {
        target = document.getElementById(targetId);

        if (target.style.display == "none")
        {
            target.style.display = "block";
        }

        else
        {
            target.style.display = "none";
        }
    }
}
```

- 2. Add a link.** A toggle link looks like this:

```
<div><a href="#" title="Toggle section" onclick="swap('hiddenDiv');return
false;">Toggle div!</a>
</div>
```

The value within single quotes—hiddenDiv—is the id value of the div that this link toggles.

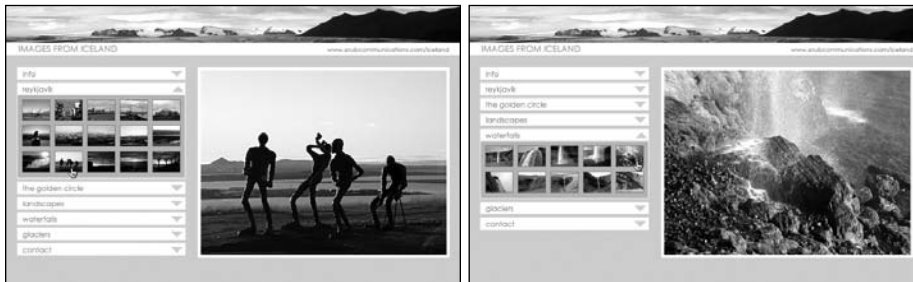
- 3. Add the div to be toggled.** Here is the div mentioned in step 2—note the id value. You can also see that an inline style attribute is included. This is required if the div is initially to be hidden. Don't put this style elsewhere (an external style sheet or the head of the document), otherwise the div toggler won't work.

```
<div id="hiddenDiv" style="display: none;"><p>Hello!</p>
</div>
```

In the screenshots (see right), CSS has been used to color the div's background, to make more obvious what's going on when the link is clicked.



A combination of the previous two exercises can be seen in action at the Images from Iceland website: www.snubcommunications.com/iceland. This site expands on the div toggler by also toggling the arrow images when a section is toggled, and it shows what you can do with some fairly straightforward JavaScript, some decent photographs, and a bit of imagination. With a little lateral thinking, you can apply this knowledge to the list-based navigation discussed in the next part of the chapter, which will enable you to toggle various parts of a navigation bar.



Creating navigation bars

The chapter has so far concentrated on inline navigation, so we'll now turn our attention to navigation bars. Before getting immersed in the technology, you need to decide what names you're going to use. When designing the basic structure of the site, content should have been grouped into categories, and this is often defined by what the user can do with it. It therefore follows that navigation bar links tend to one of the following:

- Action based (buy now, contact us, read our history)
- Site audience (end users, resellers, employees)
- Topic based (news, services, contact details)

Wherever possible, keep to one of the preceding categories rather than mixing topics and actions. This sits easier with readers. Navigation links should also be succinct, to the point, and appropriate to the brand and tone of the website.

In this section, we'll cover using lists for navigation, styling list-based navigation bars, working with inline lists, and creating graphical navigation bars with rollover graphics.

Lists for navigation

Think back to what we've covered to this point about semantic markup. Of the HTML elements that exist, which is the most appropriate for a navigation bar? If you said "A table," go to the back of the class. Using tables for navigation bars might be a rapid way of getting them up and running, but it's not structurally sound. Instead, we see navigation bars as a list of links to various pages on the website. It therefore follows that HTML lists are a logical choice to mark up navigation bars.

When creating the initial pass of the website, just create the list as it is, along with all the associated pages, and let people play around with the bare-bones site. This enables users to get a feel for its structure, without getting distracted by content, colors, and design. However, sooner or later, you're going to want to make that list look a little fancier.

Styling a list-based navigation bar

Much of the remainder of this chapter is concerned with CSS and how it can be used to style lists. From a plain HTML list, you can rapidly create exciting visual designs—and ones that are easy to update, both in terms of content and design. After all, adding another navigation link is usually just a matter of adding another list item.

Exercise: Using CSS to style a list

- 1. Create a list.** By using nested lists, you can provide the navigation bar with a hierarchical structure (and you can style each level in CSS). In this example, the list has two levels. (Refer to Chapter 3 for an overview of correctly formatting lists.) This list is nested within a div with an id value of navigation, which we'll later take advantage of by using contextual selectors.

```
<div id="navigation">
  <ul>
    <li><a href="#">Section one</a>
      <ul>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
      </ul>
    </li>
    <li><a href="#">Section two</a>
      <ul>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
      </ul>
    </li>
  </ul>
</div>
```

- Section one
 - A link to a page
 - A link to a page
 - A link to a page
 - A link to a page
- Section two
 - A link to a page
 - A link to a page
 - A link to a page
 - A link to a page
- Section three
 - A link to a page
 - A link to a page
 - A link to a page
 - A link to a page


```

        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
    </ul>
</li>
<li><a href="#">Section three</a>
    <ul>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
        <li><a href="#">A link to a page</a></li>
    </ul>
</li>
</ul>
</div>

```

In this example, all href attributes have a value of #. This is a quick way of creating a “dummy” link, and it doesn’t distract from what you’re working on by using a load of fake web page file names.

5

2. Style the page body. You now set the background color to gray, add some padding, and set the font size to small:

```

body {
background-color: #aaaaaa;
padding: 20px;
margin: 0;
font-size: small;
}

```

3. Style the list. The unordered list itself needs to be styled, so you’ll remove default items like bullet points, margins, and padding. This is done via the following rule, which also defines a set width, font, and font size:

```

#navigation ul {
list-style-type: none;
padding: 0;
margin: 0;
width: 140px;
font-family: Arial, Helvetica, sans-serif;
font-size: 100%;
}

```

```

Section one
A link to a page
A link to a page
A link to a page
A link to a page
Section two
A link to a page
A link to a page
A link to a page
A link to a page
Section three
A link to a page
A link to a page
A link to a page
A link to a page

```

4. Style list items. For the benefit of Opera, you also set the list item margins to 0:

```

#navigation li {
margin: 0;
}

```

- 5. Style buttons.** You'll use a contextual selector to style links within the navigation div (i.e., the links within this list). These styles initially affect the entire list, but you'll later override them for level two links. Therefore, the styles you're working on now are intended only for level one links (which are for sections or categories).

This first set of property/value pairs turns off the default link underline, sets the list items to uppercase, and defines the font weight as bold.

```
#navigation a {
text-decoration: none;
text-transform: uppercase;
font-weight: bold;
}
```

- 6. Set button display and padding.** Still within the same rule, set the buttons to display as block, thereby making the entire container an active link (rather than just a link text). Add some padding so the links don't hug the edge of the container.

```
display: block;
padding: 3px 12px 3px 8px;
```

- 7. Define colors.** Define the button background and foreground colors, setting the former to gray and the latter to white.

```
background-color: #666666;
color: #ffffff;
```

- 8. Use borders for a 3D effect.** Borders can be styled individually. By setting the left and top borders to a lighter shade than the background, and the right and bottom borders to a darker shade, a 3D effect is achieved. (Don't use black and white, because they are too harsh.)

```
border-top: 1px solid #dddddd;
border-right: 1px solid #333333;
border-bottom: 1px solid #333333;
border-left: 1px solid #dddddd;
```

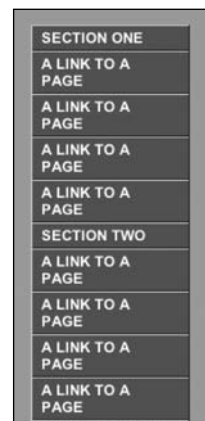
This completes the #navigation a rule.

- 9. Define other states.** The hover state is defined by just changing the background color, making it slightly lighter.

```
#navigation a:hover {
background-color: #777777;
}
```

The active state enables you to build on the 3D effect: the padding settings are changed to move the text up and left by 1 pixel, the background and foreground colors are made slightly darker, and the border colors are reversed.

```
#navigation a:active {
padding: 2px 13px 4px 7px;
background-color: #444444;
```



```

color: #eeeeee;
border-top: 1px solid #333333;
border-right: 1px solid #dddddd;
border-bottom: 1px solid #dddddd;
border-left: 1px solid #333333;
}

```

10. Style page buttons. The selector `#navigation li li a` enables you to style links within a list item that are themselves within a list item (which happen to be in the navigation div). In other words, you can create a declaration for level two links.

These need to be differentiated from the section links, so you'll set them to lowercase and normal font weight (instead of bold). The padding settings indent these links more than the section links, and the background and foreground colors are different, being black on light gray rather than white on a darker gray.

```

#navigation li li a {
text-decoration: none;
text-transform: lowercase;
font-weight: normal;
padding: 3px 3px 3px 17px;
background-color: #999999;
color: #111111;
}

```

11. Style page button hover and active states. This is done in the same way as per the section links, changing colors as appropriate and again reversing the border colors on the active state.

```

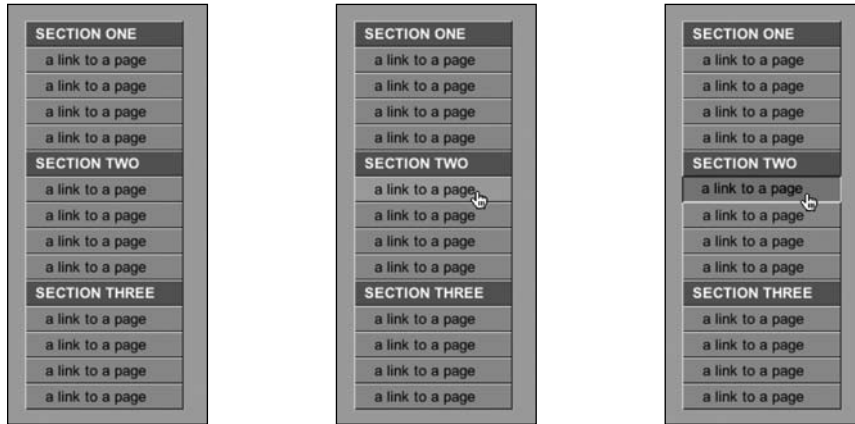
#navigation li li a:hover {
background-color: #aaaaaa;
}

#navigation li li a:active {
padding: 2px 4px 4px 16px;
background-color: #888888;
color: #000000;
border-top: 1px solid #333333;
border-right: 1px solid #dddddd;
border-bottom: 1px solid #dddddd;
border-left: 1px solid #333333;
}

```

The navigation bar is now complete and, as you can see from the following images (which depict, from left to right, the default, hover, and active states), the buttons have a real tactile feel to them. Should this not be to your liking, it's easy to change the look of the navigation bar because everything's styled in CSS. To expand on this design, you could introduce background images for each state, thereby making the navigation bar even

more graphical. However, because you didn't simply chop up a GIF, you can easily add and remove items from the navigation bar, just by amending the list created in step 1.



You could take this exercise further by adding a class to the relevant list item on each page of the site that such a navigation bar is used on, thereby providing users with a visual idea of where they are in the site.

Exercise: Adding a toggler

You could also combine this list with the ideas explored in the “Setting up a toggling div” exercise earlier in the chapter. The JavaScript doesn't need changing, so you're just going to make the amendments required to the list created in the previous exercise.

- 1. Set up toggle links.** For each of the section links, add an onclick event. The swap value must be unique for each link, so the value for the section two link would be `sectionTwoLinks`, and so on.

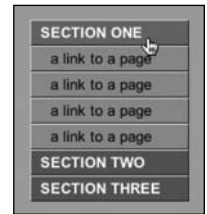
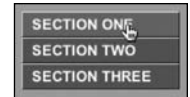
```
<a href="#" onclick="swap('sectionOneLinks');return false;">Section one</a>
```

- 2. Amend the nested lists.** The start tag for each of the nested lists needs two additional attributes. The first is a unique `id` with a value correlating with that set in step 1. The second is an inline `style` attribute, setting `display` to `none` (leave this out if you want the lists to be visible by default).

```
<ul id="sectionOneLinks" style="display: none;">
```

Take care in ensuring section links correlate with the relevant nested list, otherwise section links may toggle the wrong lists!

The result of this exercise is a list-based navigation bar in which the nested lists can be toggled (thereby providing you with a collapsible navigation bar; see right).



Using inline lists

Although most people use lists to set items vertically, it's possible to create horizontal lists. This might strike you as an odd thing to do, but it can be handy. For instance, if you want to stick to the semantic markup ideas we've been talking about, but require a horizontal navigation bar, you can set the list items to display inline. This example uses the following HTML list, again nested within a div with an id of navigation:

```
<div id="navigation">
  <ul>
    <li><a href="#">Latest news</a></li>
    <li><a href="#">Services</a></li>
    <li><a href="#">Customer support</a></li>
    <li><a href="#">Contact details</a></li>
  </ul>
</div>
```

All we need to set the list to display horizontally (inline) is the following CSS:

```
#navigation li {
  display: inline;
}
```

Getting rid of the default bullets, padding, and margins ensures the list displays in the same way across browsers and platforms:

```
#navigation ul {
  list-style-type: none;
  padding: 0;
  margin: 0;
}
```

[Latest news](#) [Services](#) [Customer support](#) [Contact details](#)

As you can see from the preceding screen shot, this is pretty basic, but with some extra CSS definitions it can easily be spiced up:

LATEST NEWS SERVICES CUSTOMER SUPPORT CONTACT DETAILS

Should you set a background color on an inline list and find white gaps between items, delete the white space between the list tags in the HTML document.

Inline lists as breadcrumbs

Inline lists can be handy for creating **breadcrumb** links on complex websites. These are links that show the path you've taken to get to the current document. The markup is the same as in the inline lists example, but with extra padding to the left of each list item, and a nonrepeated background image that creates the familiar arrows:

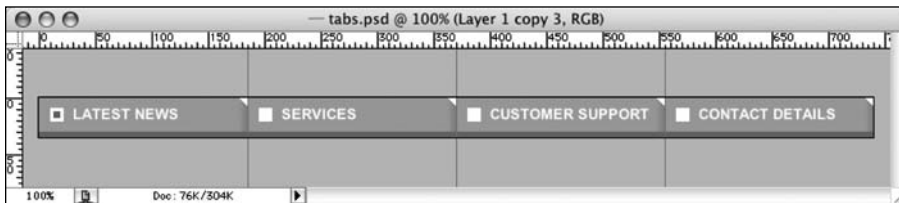
```
#navigation li {
  display: inline;
  font: 12px Arial, sans-serif;
  padding: 0 10px 0 15px;
  background: url(breadcrumb_bullet.gif) left no-repeat;
}

#navigation a {
  color: #000000;
}
```

>> [Home page](#) >> [Reviews](#) >> [Live gigs](#) >> London, 2004

Graphical navigation with rollovers

The final exercise in this chapter concerns navigation with graphical rollovers. The following is a Photoshop mock-up of the navigation bar that we're going to build. (The vertical lines are Photoshop guides, showing the boundary of each button, and are not part of the design.)



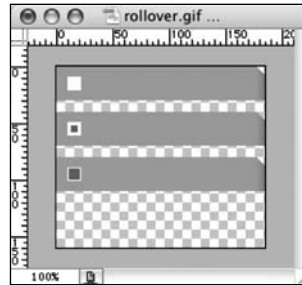
By conventional methods, you'd export eight images from Photoshop—two for each link (twelve if you wanted to incorporate an active state). The rollovers would be applied using JavaScript. However, this process has several problems:

- Updating the rollover graphics requires a lot of work, creating and re-exporting up to a dozen new images.
- By default, there is a short pause on the hover and active states while the rollover graphic downloads.

- The hover state and active state pauses can be eradicated by using a preload script, but such scripts often don't work well with all browsers.
- Not everyone surfs the Web with JavaScript turned on.

Instead of using JavaScript, we'll use a little CSS and (drumroll) just one image, which is depicted to the right.

This image is a *single* transparent GIF that includes all three link states: link (the default), hover, and active. (What's depicted is the final image, which should be exported in one piece; this should remain a single image file, and shouldn't be chopped into three different files.) What we can do is use this as a background image for navigation links, and use CSS to display the relevant portion of the image, depending on the link action taking place.

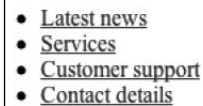


As with the majority of examples in this chapter, the navigation is marked up as an unordered list. No formatting of the links of any kind is required—that's all handled by the CSS. We've again placed the list within a div tag that has an id value of navigation, so we can create CSS rules that apply only to lists within that specific div.

Exercise: Using CSS to create a graphical navigation bar

- 1. Create the list.** As with previous navigation bars created in this chapter, this one consists of a simple unordered list. In a browser, this looks pretty much like you'd expect: just a plain, unordered, vertical list.

```
<div id="navigation">
  <ul>
    <li><a href="#">Latest news</a></li>
    <li><a href="#">Services</a></li>
    <li><a href="#">Customer support</a></li>
    <li><a href="#">Contact details</a></li>
  </ul>
</div>
```



- 2. Style the list.** As done previously, browser defaults are removed.

```
#navigation ul {
  margin: 0px;
  padding: 0px;
  list-style: none;
}
```

- 3. Style list items.** Items within the list are styled to float left and display inline. The background value includes the location of the rollover image, with additional

settings being no-repeat (to stop it from tiling), then left and top, to ensure the relevant portion of the rollover image is seen by default. The margin and padding are set to 0 to override unruly browser defaults.

```
#navigation li {
float: left;
display: inline;
margin: 0px;
padding: 0px;
background: url(assets/shared/rollover.gif) no-repeat left top;
}
```



- 4. Style the links.** Because you haven't set dimensions for the links yet, the backgrounds aren't sitting in the right place. You can fix this by using the following rule, which also deals with the text styles you want to put in place.

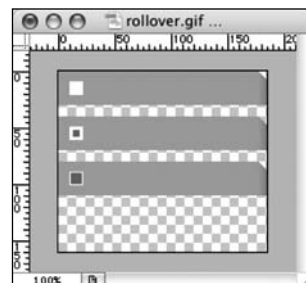
The padding and height/width settings add up to the dimensions of the area of the background image that you want to show at any one time (185px × 30px—the size of one of the link states). This is because padding is added to the element width—it's not a part of it. If you were to set the width of navigation links to 185px and then add 30px of padding to the left (which is used to indent the text), the links would take up 215px of space.

You'll again set display to block, to make the entire container the active link, thereby making this navigation bar work in the normal manner.

```
#navigation a {
font: bold 13px Arial, Helvetica, sans-serif;
text-transform: uppercase;
color: #ffffff;
text-decoration: none;
display: block;
padding: 7px 0px 0px 30px;
height: 23px;
width: 155px;
}
```



- 5. Style other states.** For the hover and active states, you define which portion of the rollover graphic is supposed to be visible. This is done via background-position values. The first of these remains 0px, because you always want to see the image from its far left. The vertical reading depends on where the relevant portion of the image appears in the rollover graphic.



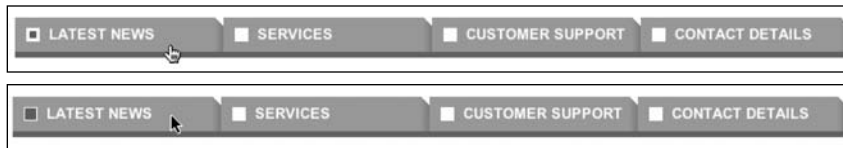
As you can see, the hover state is 40px from the top and the active state is 80px from the top. This means the image needs to be vertically moved -40px and -80px for the hover and active states, respectively.

Therefore, the rules for these states are as follows:

```
#navigation a:hover {
background: url(assets/shared/rollover.gif) 0px -40px;
}

#navigation a:active {
background: url(assets/shared/rollover.gif) 0px -80px;
}
```

The hover and active states are shown in the following images.



- 6. Fix for Internet Explorer 5.5 for Windows.** We get more into browser hacks in Chapter 12, but we'll briefly outline one here. Internet Explorer 5.5 for Windows gets the box model wrong, incorrectly setting padding and borders within defined element dimensions. The navigation bar currently looks like this in that browser:



To deal with Internet Explorer 5.5, the width and height settings need to be the same as the dimensions of the buttons (as opposed to the measurements for compliant browsers, defined in step 4). You can set separate values for Internet Explorer 5.5 by using the box model hack. First, set the width and height for Internet Explorer 5.5 (185px width and 30px height—dimensions identical to those of one state of the rollover graphic). Then add the hack and the correct measurements for compliant browsers (as in step 4).

```
#navigation a {
font: bold 13px Arial, Helvetica, sans-serif;
text-transform: uppercase;
color: #ffffff;
text-decoration: none;
display: block;
padding: 7px 0px 0px 30px;
height: 30px;
width: 185px;
voice-family: "\"}\"";
voice-family:inherit;
height: 23px;
width: 155px;
}
```

The hack is the two voice-family rules. Internet Explorer 5.5 stops reading during the first line (it's fooled into thinking the rule terminates due to the quoted curly bracket); compliant browsers “recover” in the second voice-family line and continue to the end of the rule, with the second set of height and width values overriding the first. Everyone goes home happy—well, almost. Internet Explorer 5.5 sometimes also screws up the case of the links, ignoring the text-transform value. A workaround is to create a second #navigation a rule and put the text-transform property/value pair there:

```
#navigation a {
text-transform: uppercase;
}
```



As you can see, this fixes the navigation bar in Internet Explorer 5.5. Sadly, Internet Explorer 5 for Windows cannot deal with text-transform at all, but it manages to display everything else correctly. Still, it's only a minor thing, and that browser's market share is rapidly reducing anyway. As for obsolete browsers (Netscape 4 et al.), hide the CSS and they'll get a perfectly navigable basic list. Again, everyone goes home happy (and I mean it this time!).

It's worth noting that many browsers permit text resizing. In such cases, users with extreme setups (text size increased a couple of settings above the default) may end up “losing” the second word of multiple-word links in this example. This will affect a minority of users, but even so, take care to ensure your navigation makes sense even if the final word is missing. For instance, contact details becoming contact still makes sense, as does help desk becoming help; however, customer support becoming customer could be problematic. Despite this issue, this method is still superior to using graphics for each link: it's easier to update, more accessible, and degrades more gracefully in alternate browsers.

Updating a graphical navigation bar

This is an extremely flexible system. Even though it has fixed-width links, it's easy to update (just create a single new rollover graphic and rework the CSS rules—such as the width settings—to accommodate the required number of links). Because there's no JavaScript, it works well with all current browsers, and because the navigation consists of semantic markup, it's logical even in browsers that don't support CSS.

Should you want to have a different image for each link (such as a unique color per tab), use CSS classes. Despite the extra work involved, the system remains beneficial compared to older methods, because there's no JavaScript, there are no preloaders, and you control everything from an external document.

Drop-down menus

A method of navigation expansion—and one that web designers often crave—is drop-down menus. It's a good idea to avoid those created by web design applications, because they tend to be composed of obsolete and invalid code. However, we're not going to create one here, because a perfectly good one exists online. Check out the excellent drop-down menu system from [gazingus.org](http://www.gazingus.org/html/Using_Lists_for_DHTML_Menus.html) (available from www.gazingus.org/html/Using_Lists_for_DHTML_Menus.html). It's entirely CSS-based, and although it requires JavaScript to fully function, those who surf without JavaScript are still able to access the top level of each section.

If you do decide to create drop-down menu-based navigation, avoid aping an operating system's menu style, because this may confuse visitors using that operating system and irritate visitors using a rival system. The exception to this rule is if you're creating a site that centers around nostalgia for the days where operating systems used to come on floppy disks. One such site—a Mac OS System 7 look-alike—can be found at <http://myoldmac.net/index-e.htm>.

Dos and don'ts when designing web navigation

So, that's the end of our navigation chapter. Before we move on to working with layout, here are a few succinct tips regarding designing web navigation.

Do

- Use appropriate types of navigation.
- Provide alternate means of accessing information.
- Ensure links stand out.
- Take advantage of link states to provide feedback for users.
- Get the link state order right (link, visited, hover, active).
- Use `accesskey` and `tabindex` attributes.
- Use styled lists for navigation.
- Use CSS for rollovers.

Don't

- Add search boxes just for the sake of it.
- Use deprecated body attributes.
- Style navigation links like normal body copy.
- Use image maps unless absolutely necessary.
- Open new windows from links or use pop-ups.
- Use clunky JavaScript for rollovers.