CHAPTER 6

Common Uses of JavaScript: Images and Windows

If you read the last few chapters, you should be well equipped now with your knowledge of JavaScript and its interaction with CSS and HTML. Now you will learn about some of the most common uses of JavaScript on the Web these days, and we'll go through some examples. In these examples, you'll see how to ensure that they work independently of other scripts on the page, and I'll explain what problems might occur. We'll also touch on what functionality is tempting to use but might not be the safest of options.

Note This chapter consists of a lot of code examples, and you will be asked to open some of them in a browser to test the functionality for yourself, so if you haven't been to `http://www.beginningjavascript.com` yet to download the code examples for this book, it might be a good time to do so now.

Most of the full code examples here use DOM-2 event handling. This makes them a bit more complex than their DOM-1 equivalents, but it also makes them work a lot better with other scripts, and they are much more likely to work in future browsers. Just bear with me, and I promise that by repeatedly using these methods, you will get the hang of them quite quickly.

The examples are also developed with maintenance and flexibility in mind. This means that everything that might be changed at a later stage by non-JavaScript-savvy people is stored in properties and that you can easily have several parts of the same document use the scripts' functionality. This also adds to the complexity of some of the scripts, but it is a real-life deliverable most clients ask for.

Images and JavaScript

Dynamic changing of images was most likely the first "wow" effect JavaScript was used for. When CSS was not supported by browsers yet (and—to be fair—was still in the process of being defined), JavaScript was the only way to change an image when the user moved the mouse over it or clicked it. In recent years, more and more of the image effects traditionally achieved via JavaScript have been replaced by pure CSS solutions that make maintenance a lot easier. We'll discuss these later; for now, let's take a look at the basics of what JavaScript can do to images.

Basics of Image Scripting

In JavaScript you can reach and amend images in two ways: the DOM-2 way via `getElementsByTagName()` and `getElementById()`, or an older way, which involves the `images` collection that is stored in a property of the `document` object. As an example, let's take an HTML document with a list of photos:

```
<ul class="slides">
  <li><img src="pictures/thumbs/cat2.jpg" alt="Lazy Cat" /></li>
  <li><img src="pictures/thumbs/dog10.jpg" alt="Dog using the shade" /></li>
  <li><img src="pictures/thumbs/dog12.jpg" alt="Squinting Dog" /></li>
  <li><img src="pictures/thumbs/dog63.jpg" alt="Dog cooling off in the sand" /></li>
  <li><img src="pictures/thumbs/dog7.jpg"  alt="Very flat dog" /></li>
  <li><img src="pictures/thumbs/donkeycloseup.jpg" alt="Curious Donkey" /></li>
  <li><img src="pictures/thumbs/donkeyeating.jpg" alt="Hay-eating Donkey" /></li>
  <li><img src="pictures/thumbs/kittenflat.jpg" alt="Ginger and White Cat" /></li>
</ul>
```

You can retrieve all these photos in JavaScript to do something to them in both ways:

```
// Old DOM
var photosOldDOM=document.images;
// New DOM
var photos=document.getElementsByTagName('img');
```

Both methods result in an array containing all the images as objects. As with any object, you can read and manipulate their properties. Say, for example, you want to know the alternative text of the third image. All you need to do is to read out the `alt` property of the object:

```
// Old DOM alt property
var photosOldDOM=document.images;
alert(photosOldDOM[2].alt);
// W3C DOM-2 alt attribute
var photos=document.getElementsByTagName('img');
alert(photos[2].getAttribute('alt'));
```

Images have several properties, some of which are obvious, but others you may not have heard about:

- `border`: The value of the `border` attribute in the HTML
- `name`: The `name` attribute of the `img` tag
- `complete`: A property that is `true` if the image has finished loading (read-only—you cannot change this attribute)
- `height`: The height of the image (in pixels—returned as an integer)
- `width`: The width of the image (in pixels—returned as an integer)
- `hspace`: The horizontal space around the image
- `vspace`: The vertical space around the image
- `lowsrc`: The image preview as defined in the attribute of the same name
- `src`: The URL of the image

You can use these properties to access and change images dynamically. If you open the example document `exampleImageProperties.html` in a browser, you can read and write the properties of the demonstration image as Figure 6-1 shows.

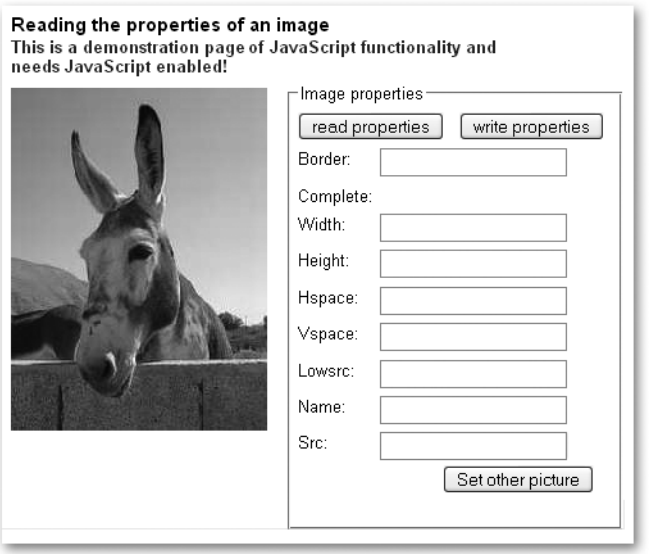

Figure 6-1. *Reading and writing the properties of an image*

■ **Note** Notice that if the dimensions of the image have been defined via the HTML `width` and `height` attributes, and you change its source, you don't automatically change its dimensions. For an example, activate the Set other picture button in the demo. This can result in unsightly distortion of the other image, as browsers don't resize images in a sophisticated way.

Preloading Images

If you use images dynamically in the page for rollover or slide show effects, you'll want to have the images already loaded into the browser's memory cache to give the visitor a smooth experience. You can do this in several ways. One is to create a new image object for each image you want to preload when you initialize the page:

```
kitten = new Image();
kitten.src = 'pictures/kittenflat.jpg';
```

You'll see an example of this soon in the "Rollover Effects" section. Some development tools offer scripts that loop through all the images necessary like Macromedia's simple preloader:

```
function simplePreload() {
  var args = simplePreload.arguments;
  document.imageArray = new Array( args.length );
  for(var i = 0; i < args.length; i++ ) {
    document.imageArray[i] = new Image;
    document.imageArray[i].src = args[i];
  }
}
```

If you call this function with the images you want to preload, it'll create a new array with all the images in it, loading them one after the other, for example:

```
simplePreload( 'pictures/cat2.jpg', 'pictures/dog10.jpg' );
```

A different, scripting-independent, way of preloading images is putting them as 1×1-pixel images in the HTML inside a container element that you hide via CSS. This mixes structure and behavior and has the same issue as any image preloading technique has: You force the visitor to download a lot of images he might not want to see immediately. If you were to use preloaders, it might be a good option to keep them optional and let the user decide if he wants to preload all the images.

We'll keep image preloading brief here, as there is much more to learn about images.

Rollover Effects

Rollover or hover effects were the absolute craze when JavaScript first got supported widely in the most common user agents. Many scripts were written, and a lot of small tools came out that allowed "instant rollover generation without any need to code."

The idea of a rollover effect is pretty easy: you hover with your mouse over an image and the image changes, indicating that this is a clickable image and not just eye candy. Figure 6-2 shows a rollover effect.

Figure 6-2. *A rollover effect means the element changes its look when the mouse hovers over it.*

Rollovers Using Several Images

You can create a rollover effect by changing the `src` property of the image when the mouse hovers over it. Old-school rollover effects were tied to the `name` attribute of the `<img>` tag and used the `images` collection. A construct like this was not uncommon in web pages in the 1990s:

`exampleSimpleRollover.html` *(excerpts)*

HTML:

```
<a href="contact.html"
   onmouseover="rollover('contact','but_contact_on.gif')"
   onmouseout="rollover('contact','but_contact.gif')">
   <img src="but_contact.gif" name="contact"
      width="103" height="28" alt="Contact Us" border="0" />
</a>
```

JavaScript:

```
function rollover( img, url ) {
  document.images[img].src=url;
}
```

The problem with rollovers was (and still is) that the second image might not be loaded yet, which is counterproductive. The fact that this is an interactive element is not immediately obvious—only when the second image is shown—so this would confuse rather than aid the user in this case. This is why the classic rollover functions like the one that came bundled with

Macromedia Dreamweaver use the image object preloading technique explained earlier in conjunction with the name attribute:

examplePreloadingRollover.html *(excerpts)*

```
<a href="contact.html"
   onmouseover="rollover('contact',1)"
   onmouseout="rollover('contact',0)">
    <img src="but_contact.gif" name="contact"
      width="103" height="28" alt="Contact Us" border="0" />
</a>
```

JavaScript:

```
contactoff = new Image();
contactoff.src = 'but_contact.gif';
contacton = new Image();
contacton.src = 'but_contact_on.gif';
function rollover( img, state ) {
  var imgState = state == 1 ? eval(img + 'on.src') :➥
  eval(img + 'off.src');
  document.images[img].src = imgState;
}
```

The script creates two new image objects with the names `contacton` and `contactoff`. The rollover script checks the name of the image and the state of the rollover and uses `eval()` to retrieve the correct object and read its `src` property.

It then sets the `src` property of the image to the `src` property retrieved from the object. You can imagine the amount of code when you want to have 20 rollover images in a page, so you can see it was necessary to come up with a more advanced and generic way of creating rollovers.

Daniel Nolan came up with a very clever solution in 2003 as described at `http://www.dnolan.com/code/js/rollover/`. His solution uses the file name of the image and assumes a suffix of "_o" for the rollover state. All you need to add to the image you want to have a rollover effect for is a class called `imgover`.

You can replicate the same functionality easily using DOM-2 handlers. First you need an HTML document that has images with the correct class assigned to them:

exampleAutomatedRollover.html *(excerpt)*

```
<ul>
  <li>
    <a href="option1.html">
      <img src="but_1.gif" class="roll" alt="option one" />Option 1
    </a>
  </li>
```

```
  <li>
    <a href="option2.html">
      <img src="but_2.gif" class="roll" alt="option two" /> Option 2
    </a>
  </li>
  [... code snipped ...]
</ul>
```

Then you plan your script. The main object of the script will be called `ro` for rollover. As you want to make things as easy as possible for future maintainers, you keep all the bits and bobs that might change in properties of the main object.

In this script, this is the class that defines which image should get a rollover state and the suffix of the mouseover image. In this case, you'll use "roll" and "_on", respectively. You will need two methods, one to initialize the effect and one to do the rollover. Furthermore, you will need an array to store the preloaded images. All of this together makes up the skeleton of the rollover script:

`automatedRollover.js` *(skeleton)*

```
ro = {
  rollClass : 'roll',
  overSrcAddOn : '_on',
  preLoads : [],
  init : function(){},
  roll : function( e ){}
}
DOMhelp.addEvent( window, 'load', ro.init, false );
```

Let's start fleshing out the skeleton. First up are the properties and the `init` method. In it you predefine a variable called `oversrc` and store all the images of the document in an array called `imgs`. You loop through the images and skip those that don't have the right CSS class attached to them.

`automatedRollover.js` *(excerpt)*

```
ro = {
  rollClass : 'roll',
  overSrcAddOn : '_on',
  preLoads : [],
  init : function() {
    var oversrc;
    var imgs = document.images;
    for( var i = 0; i < imgs.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', imgs[i], ro.rollClass ) ) {
        continue;
      }
```

If the image has the right CSS class attached to it, you read its source attribute, replace the full stop in it by the suffix defined in the `overSrcAddOn` property followed by a full stop, and store the result in the `oversrc` variable.

`automatedRollover.js` *(continued)*

```
      oversrc = imgs[i].src.toString().replace( '.',➥
      ro.overSrcAddOn + '.');
```

■ **Note** For example, the first image in the document has the `src` `but_1.gif`. The value of `oversrc` with the suffix property defined here would be `but_1_on.gif`.

You then create a new image object and store it as a new item of the `preLoads` array. Set the `src` attribute of the new image to `oversrc`. Use `addEvent()` from the DOMhelp library to add an event handler for both `mouseover` and `mouseout` that points to the `roll` method.

`automatedRollover.js` *(continued)*

```
      ro.preLoads[i] = new Image();
      ro.preLoads[i].src = oversrc;
      DOMhelp.addEvent( imgs[i], 'mouseover', ro.roll, false );
      DOMhelp.addEvent( imgs[i], 'mouseout', ro.roll, false );
    }
  },
```

The `roll` method retrieves the image the event occurred on via `getTarget(e)` and stores its `src` property in a variable called `s`. You then test which of the events occurred by reading out the event type. If the event type was `mouseover`, you replace the full stop in the file name with the add-on followed by a full stop, and vice versa if the event was `mouseout`. You add an event handler to the window that calls `ro.init()` when the window has finished loading.

`automatedRollover.js` *(continued)*

```
  roll : function( e ) {
    var t = DOMhelp.getTarget( e );
    var s = t.src;
    if( e.type == 'mouseover' ) {
      t.src = s.replace( '.', ro.overSrcAddOn + '.' );
    }
    if( e.type == 'mouseout' ) {
      t.src = s.replace( ro.overSrcAddOn + '.', '.' );
    }
  }
}
DOMhelp.addEvent( window, 'load', ro.init, false );
```

The outcome of the demo page as shown in Figure 6-3 features rollovers with highlight images that are already loaded into the browser's cache when the user hovers over the original ones.

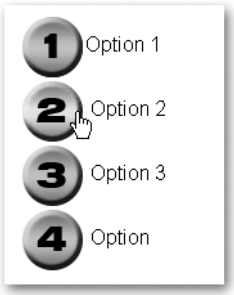

Figure 6-3. *The preloaded and automated rollovers*

As much as you can try to use clever scripting to preload images, it might not always work. The user's browser cache settings or special settings in her connection might make it impossible to sneakily preload something in the back without really adding the image to the document. Therefore, it might be a safer option to use a single image for the rollover effect.

Rollover Effects Using a Single Image

When CSS designers started exploring the `:hover` pseudo-selector to do a bit more than just changing the underline of a link, CSS-only rollovers were born. These basically mean that you assign different background images to the link and the hover state of the link.

The same problems occurred—images had to get loaded before they were displayed, which made the rollover effect flicker or not happen at all. The solution was to take one single image for both states and use the `background-position` property to change the location of the image as shown in Figure 6-4.

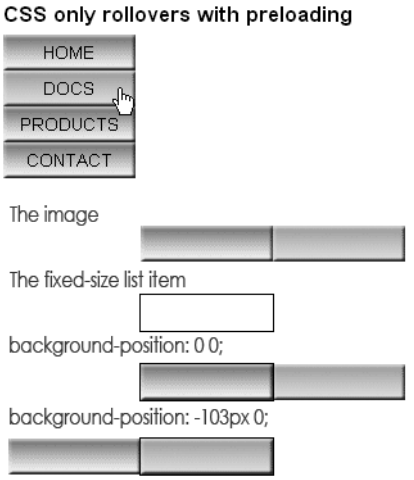

Figure 6-4. *Rollover effects with background position and CSS*

You can see the effect by opening `exampleCSSonlyRollover.html` in a browser. The CSS in question constrains the link to a certain size and achieves the rollover effect by shifting the background image in the hover state to the left via a negative background-position value that is half the width of the image:

`exampleCSSonlyRollover.html` *(excerpt)*

```
#nav a{
  width:103px;
  padding-top:6px;
  height:22px;
  background:url(doublebutton.gif) top left no-repeat #ccc;
}
#nav a:hover{
  background-position:-103px 0;
}
```

You can do the same in JavaScript; however, let's be more creative and do something CSS cannot do.

Rollover Effects on Parent Elements

Let's take an HTML list and turn it into a snazzy navigation bar by adding a nice background image, and then make the links change the background image when you hover over them. The first thing you need is a background image with all the states of the background as shown in Figure 6-5.

Figure 6-5. *The navigation background with all states (resized)*

The HTML for the navigation bar is a list of links. As basic web usability strongly suggests never linking the current page, the current link is replaced with a `<strong>` tag.

`exampleParentRollover.html` *(excerpt)*

```
<ul id="nav">
  <li><a href="index.html">Home</a></li>
  <li><a href="documentation.html">Documentation</a></li>
  <li><strong>Products</strong></li>
  <li><a href="contact.html">Contact Us</a></li>
</ul>
```

However, as this navigation could be the first level in a multilevel navigation menu, it might also be that the highlight is not a STRONG element but a class on the list item instead:

```
<ul id="nav">
  <li><a href="index.html">Home</a></li>
  <li><a href="documentation.html">Documentation</a></li>
  <li class="current"><a href="products.html">Products</a></li>
  <li><a href="contact.html">Contact Us</a></li>
</ul>
```

Both scenarios have to be taken into account. Explaining the CSS in the demo page is not the purpose of this book; it suffices to say that you fix the dimensions of the list with the ID nav, float it to the left, and float all list elements in it.

Instead, let's go straight into planning the script. You'll need to define several properties for the main object (which is called pr for parent rollover): the ID of the navigation list, the height of the navigation (which is also the height of each of the images and necessary for the background position), and the optional class that might have been used to highlight the current section instead of a tag.

parentRollover.js *(excerpt)*

```
pr = {
  navId : 'nav',
  navHeight : 50,
  currentLink : 'current',
```

You start with an initialization method that checks for DOM support, and whether the necessary list with the right ID is available.

parentRollover.js *(continued)*

```
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
    pr.nav = document.getElementById( pr.navId );
    if( !pr.nav ){ return; }
```

The next task is to loop through all the list items contained in this list and check if there is either a STRONG element inside the item or the item has the "current" class. If either is true, the script should store the counter for the loop in the current property of the main object. This property will be used in the rollover method to reset the background to the original state.

parentRollover.js *(continued)*

```
    var lis = document.getElementsByTagName( 'li' );

    for(var i = 0; i < lis.length; i++)
    {
```

```
      if( lis[i].getElementsByTagName( 'strong' ).length > 0 ||
          DOMhelp.cssjs('check', lis[i], pr.currentLink) ) {
        pr.current = i;
      }
```

Each of the list items gets a new property called `index`, which contains its counter value in the whole list array. Using this property is a trick that prevents you from having to loop through all the list items and compare them with the target in the event listener method.

You assign two event handlers pointing to the `roll()` method: one when the mouse is over the list item, and another when the mouse leaves the list item.

`parentRollover.js` *(continued)*

```
      lis[i].index = i;
      DOMhelp.addEvent( lis[i], 'mouseover', pr.roll, false );
      DOMhelp.addEvent( lis[i], 'mouseout', pr.roll, false );
    }
  },
```

The rollover method starts by predefining a variable called `pos` that later on becomes the offset value needed to show the correct image. It then calls `getTarget()` to determine which element was rolled over and compares the node name of the `target` with `LI`. This is a safety measure because —although you assigned the event handler to the `LI`—browsers may actually send the link instead as the event target. The reason might be that a link is an interactive page element, whereas an `LI` isn't, and the browser's rendering engine considers a link more important. You won't know, but you should be aware of the fact that some user agents will see the link instead of the list element as the event target.

`parentRollover.js` *(continued)*

```
  roll : function( e ) {
    var pos;
    var t = DOMhelp.getTarget(e);
    while(t.nodeName.toLowerCase() != 'li'
          && t.nodeName.toLowerCase() != 'body') {
      t = t.parentNode;
    }
```

Then, you define the position needed to show the right background image. This position is either the `index` value of the list item or the stored `current` property multiplied with the height of each image.

Which of the two gets applied depends on whether the user hovers his mouse over the list item or not—something you can find out by comparing the event type with `mouseover`. Set the style of the navigation's background position accordingly, and then call the `init()` method when the page has finished loading.

`parentRollover.js` *(excerpt)*

```
    pos = e.type == 'mouseover' ? t.index : pr.current;
    pos = pos * pr.navHeight;
    pr.nav.style.backgroundPosition = '0 -' + pos + 'px';
  }
}
DOMhelp.addEvent( window, 'load', pr.init, false );
```

When you open `exampleParentRollover.html` in a browser, you can see that rolling over the different links of the navigation shows the different background images as demonstrated in Figure 6-6.

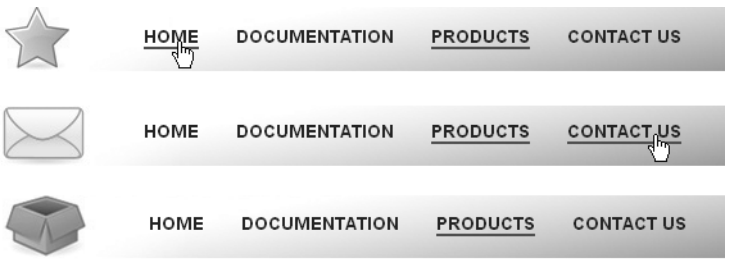

Figure 6-6. *The navigation in different rollover states*

This is a programmatic solution for the problem of rollovers affecting parent elements. However, it has one problem: if the order of the menu items were to change, the maintainer would also have to change the image accordingly. This is not a very flexible solution, which is why you might be better off using dynamically assigned classes to the navigation list to position the background image.

The necessary changes to the script affect the properties and the `roll()` method; the initialization stays the same. In addition to the `currentLink` and the `navId` property, you also need a class name to add to the navigation list. This new property can be called `dynamicLink`.

In the `roll()` method, you check once again whether the event triggering the method was `mouseover`, and add or remove a new dynamic class accordingly. This dynamically assigned

and named class consists of the dynamicLink property value and the current index plus one (as it is easier for humans to have a first class called item1 class instead of item0):

parentCSSrollover.js *as used in* parentCSSrollover.html *(abbreviated)*

```
pr = {
  navId : 'nav',
  currentLink : 'current',
  dynamicLink : 'item',
  init : function() {
   // [... same as in parentRollover.js ...]
  },
  Roll : function( e ) {
    // [... same as in parentRollover.js ...]
    var action = e.type == 'mouseover' ? 'add' : 'remove';
    DOMhelp.cssjs( action, pr.nav, pr.dynamicLink + ( t.index + 1 ) );
  }
}
DOMhelp.addEvent( window, 'load', pr.init, false );
```

This way you allow the CSS designer to define the different states for the rollover navigation as classes in the CSS:

parentCSSrollover.css *as used in* parentCSSrollover.html *(excerpt)*

```
#nav.item1{
  background-position:0 0;
}
#nav.item2{
  background-position:0 -50px;
}
#nav.item3{
  background-position:0 -100px;
}
#nav.item4{
  background-position:0 -150px;
}
```

This also provides the CSS designer with one more hook to design the navigation: the dynamic class can be used to define the current rollover or highlight state of the link itself differently from item to item.

Slide Shows

Slide shows are small images embedded in the page with previous and next buttons or sometimes even automatic changing of images after a certain time. They are used to illustrate text or offer different views of a product.

We can distinguish between two kinds of slide shows: embedded ones that have all the images in the same document and dynamic ones that load the images when they are needed.

Embedded Slide Shows

Probably the easiest way to add a slide show to a page is to add all the images as a list. You can then use JavaScript to turn this list into a slide show by hiding and showing the different list items with the embedded images. The demo document `examplePhotoListInlineSlideShow.html` does exactly that, as Figure 6-7 shows.

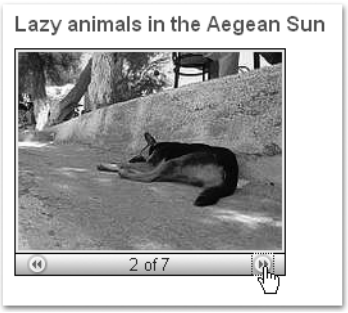

Figure 6-7. *An embedded slide show with JavaScript*

The underlying HTML is an unordered list with all the images as list items. Notice that this also allows you to set a proper alternative text for each of the images.

`examplePhotoListInlineSlideShow.html` *(excerpt)*

```
<ul class="slides">
  <li>
     <img src="pictures/thumbs/cat2.jpg" alt="Lazy Cat" />
  </li>
  <li>
    <img src="pictures/thumbs/dog10.jpg" alt="Dog using the shade" />
  </li>
  <li>
    <img src="pictures/thumbs/dog12.jpg" alt="Squinting Dog" />
  </li>
  <li>
    <img src="pictures/thumbs/dog63.jpg" alt="Dog cooling off in the sand" />
  </li>
  <li>
    <img src="pictures/thumbs/dog7.jpg"  alt="Very flat dog" />
  </li>
  <li>
    <img src="pictures/thumbs/donkeyeating.jpg" alt="Hay-eating Donkey" />
  </li>
  <li>
    <img src="pictures/thumbs/kittenflat.jpg" alt="Ginger and White Cat" />
  </li>
</ul>
```

All future maintainers have to do to change the order of the images or add or delete images is to change the HTML; there is no need to change the JavaScript at all. Provided that you supply an appropriate style sheet, visitors without JavaScript will get all the images displayed as shown in Figure 6-8. Users without style sheets will get a list with image thumbnails.

Figure 6-8. *The embedded slide show without JavaScript*

One effect of embedding all images in the document is that they will all be loaded when the visitor loads the page. This could be a good or a bad thing, depending on the visitor's connection speed. Later on we'll look at an example that loads the larger images only when the user clicks smaller ones.

Let's look at the script that turns this list into a slide show. You will use the DOMhelp library you developed in previous chapters to work around browser issues and shorten the code slightly.

As always, the first thing to do is to plan your script. In this case, you should give the CSS designer and HTML developer several classes as hooks to trigger functionality or define look and feel:

- A class to indicate that the list should be turned into a slide show
- A class to define the look and feel of the dynamic slide show list
- A class to show elements that were previously hidden
- A class to define the look of the image counter (e.g., image 1 of 3)
- A class to hide elements that should not be there at a certain state of play

You should also allow the maintainer to change the look and content of the forward and backward links and the text content of the image counter.

As for methods, all you need (apart from the helper methods contained in DOMhelp) is a global initialization method, a method to initialize each slide show, and one to show a slide. All of this together makes up the skeleton of your script:

photoListInlineSlides.js *(skeleton)*

```
inlineSlides = {

  // CSS classes
  slideClass : 'slides',
  dynamicSlideClass : 'dynslides',
  showClass : 'show',
  slideCounterClass : 'slidecounter',
  hideLinkClass : 'hide',

  // Labels
  // Forward and backward links, you can use any HTML here
  forwardsLabel : '<img src="control_fastforward_blue.png"➥
  alt="next" />',
  backwardsLabel : '<img src="control_rewind_blue.png"➥
  alt="previous" />',
  // Counter text, # will be replaced by the current image count
  // and % by the number of all pictures
  counterLabel : '# of %',

  init : function() {},
  initSlideShow : function( o ) {},
  showSlide : function( e ) {}
}
DOMhelp.addEvent( window, 'load', inlineSlides.init, false );
```

■ **Note** Notice that you offer an HTML-based option in the labels for the forward and backward links. This allows for much more flexible styling of the slide show, as the maintainer can add his own HTML (like images). Furthermore, if you want to allow the maintainer to change dynamic text like the counter, it might be beneficial to use placeholders like # and % and explain what they will be replaced with.

Let's go through the methods in the script step by step. First up is the global initialization method `init()`:

1. Test for DOM support.
2. If the test is successful, loop through all the `UL` elements of the document.
3. For each `UL`, check whether it has the class defining it as a slide show (which is stored in the `slideClass` property), and skip the rest of the steps performed by this function if it doesn't have the class (use `continue` to do this).
4. If the current `UL` is to become a slide show, you replace the class defining it as a slide show with the class defining the dynamic slide show; add a new property called `currentSlide` to the list and call the method `initSlideShow` with the list as a parameter.

`photoListInlineSlides.js` *(excerpt)*

```
  init : function() {
    if( !document.getElementById  || !document.createTextNode ) {
      return;
    }
    var uls = document.getElementsByTagName( 'ul' );
    for( var i = 0; i < uls.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', uls[i],➥
      inlineSlides.slideClass ) ) {
       continue;
      }
      DOMhelp.cssjs( 'swap', uls[i],inlineSlides.slideClass,➥
      inlineSlides.dynamicSlideClass );
      uls[i].currentSlide = 0;
      inlineSlides.initSlideShow( uls[i] );
    }
  },
```

You can spare yourself a lot of looping and checking with these tricks. First of all, replacing the class only when JavaScript is available with another class allows you to hide all the list items in the CSS instead of looping through them in the `initSlideShow()` method:

`photoListInlineSlides.css` *(excerpt)*

```
.dynslides li{
  display:none;
  margin:0;
  padding:5px;
}
```

Other dynamically assigned CSS classes you have to define to make the slide show work are a `hide` class to remove the backward link when the first image is shown or the forward link when the last image is shown and a `show` class to overrule the hiding you achieved with the `.dynslides li` selector. All the other CSS selectors and properties inside the demo are of a purely cosmetic nature.

`photoListInlineSlides.css` *(excerpt)*

```
.dynslides .hide{
  visibility:hidden;
}
.dynslides li.show{
  display:block;
}
```

By storing the current visible image in a property of the list, you don't need to loop through all the images and hide them before you show the current one. Instead, all you need to do is to determine which one will have to be shown, read the property of the parent list element, and hide the previous image stored in this property.

You then reset the property to the new image, and the next time an image gets shown, the cycle starts anew. You could have stored the current image in a property of the main object, but storing it in a property of the list means you allow for several slide shows on the same page.

The `initSlideShow()` method gets each slide show list as a parameter called `o`. First, define the variables you'll use with the `var` keyword to make sure they don't overwrite global variables with the same name. Then create a new paragraph element to host the forward and backward links and the image counter and insert it immediately after the list (using `o.nextSibling`).

`photoListInlineSlides.js` *(continued)*

```
  initSlideShow : function( o ) {
    var p, temp, count;
    p = document.createElement( 'p' );
    DOMhelp.cssjs( 'add', p, inlineSlides.slideCounterClass );
    o.parentNode.insertBefore( p, o.nextSibling );
```

Next, create the backward link by means of DOMhelp's `createLink` method and add the proper label using `innerHTML`. Add an event handler to call the `showSlide` method, hide the link by applying the appropriate CSS class, and add the link to the newly created paragraph. You'll store the link in a property of the list called `rew` to make it easier to reach it later on.

`photoListInlineSlides.js` *(continued)*

```
    o.rew = DOMhelp.createLink( '#', ' ' );
    o.rew.innerHTML = inlineSlides.backwardsLabel;
    DOMhelp.addEvent( o.rew, 'click', inlineSlides.showSlide, false );
    DOMhelp.cssjs( 'add', o.rew,inlineSlides.hideLinkClass );
    p.appendChild( o.rew );
```

A new SPAN element that acts as the image counter is next. Get the counterLabel property of the main object and replace the # character with the current list's currentSlide property value and add 1 to it (because humans start counting at 1 and not at 0 like computers do). Replace the % character with the number of LI elements in the list, and add the resulting string as a new text node to the SPAN before adding it as a new child node to the paragraph.

photoListInlineSlides.js *(continued)*

```
    o.count = document.createElement( 'span' );
    temp = inlineSlides.counterLabel.➥
    replace( /#/, o.currentSlide + 1 );
    temp = temp.replace( /%/, o.getElementsByTagName( 'li' ).length );
    o.count.appendChild( document.createTextNode( temp ) );
    p.appendChild( o.count );
```

■ **Note** Notice that you store the counter SPAN in a property of the list, called count. This is pure laziness, as it saves you having to reach it via getElementsByTagName('span')[0] later on. It also makes the script less likely to be broken by maintainers who might add other spans inside the list items at a later stage.

Adding the forward link works analogously to adding the backward link, except that the forwardsLabel property is used as the content and a new property called fwd as the shortcut.

photoListInlineSlides.js *(continued)*

```
    o.fwd = DOMhelp.createLink( '#', ' ' );
    o.fwd.innerHTML = inlineSlides.forwardsLabel;
    DOMhelp.addEvent( o.fwd, 'click', inlineSlides.showSlide, false );
    p.appendChild( o.fwd );
```

The method concludes with taking the list item that corresponds to the currentSlide property and adding the show class to it. You could have just used o.firstChild instead, but a future maintainer might want to initially show a different photo than the first one.

photoListInlineSlides.js *(continued)*

```
    temp = o.getElementsByTagName( 'li' )[o.currentSlide];
    DOMhelp.cssjs( 'add', temp, inlineSlides.showClass );
  },
```

The showSlide() method defines a variable called action and gets the event target via getTarget(e). Since you don't know if the maintainer used an image in the link labels, you need to find the link by testing whether the nodeName of the target's parentNode is A. This also

counteracts the Safari bug of sending the text contained in a link as the target instead of the link itself. The method then grabs the list the event was triggered in by reading the `closestSibling()` of the target's `parentNode`.

photoListInlineSlides.js *(continued)*

```
  showSlide : function( e ) {
    var action;
    var t = DOMhelp.getTarget( e );
    while( t.nodeName.toLowerCase() != 'a'
        && t.nodeName.toLowerCase() != 'body' ) {
      t=t.parentNode;
    }
    var parentList = DOMhelp.closestSibling( t.parentNode, -1 );
```

Summary The visitor clicks the content of the link to go one image forward or backward. The event target could be an image (as it is in this example) or text—or anything else the maintainer of this script put in the `forwardsLabel` and `backwardsLabel` properties. Therefore—and because Safari sends the text contained in a link as the target rather than the link itself—you need to check for the name of the node and compare it with `A`. Then, you take the parent node of this `A`—which is the paragraph that was newly created—and get its previous sibling, which is the `UL` that contains the images.

Next, you need to find the `currentSlide` property from the list in question and the total number of images by checking the `length` property of the list item array. Hide the previously shown image by removing the `show` class.

photoListInlineSlides.js *(continued)*

```
    var count = parentList.currentSlide;
    var photoCount = parentList.getElementsByTagName( 'li' ).➥
    length - 1;
    var photo = parentList.getElementsByTagName( 'li' )[count];
    DOMhelp.cssjs( 'remove', photo, inlineSlides.showClass );
```

Determine whether the link that was activated was the forward link by comparing the target with the `fwd` property of the list, and then increment or decrement the counter accordingly.

If the counter is larger than `0`, remove the `hide` class from the backward link; otherwise, add this class, effectively hiding or showing the link. The same logic applies for the forward link, although the comparison criteria this time is the counter being less than the total number of list items. This prevents the backward link from showing on the first slide and the forward link from showing on the last slide.

photoListInlineSlides.js *(continued)*

```
    count = ( t == parentList.fwd ) ? count+1 : count-1;
    action = ( count > 0 ) ? 'remove' : 'add' ;
    DOMhelp.cssjs( action, parentList.rew,➥
    inlineSlides.hideLinkClass );
    action = ( count < photoCount ) ? 'remove' : 'add';
    DOMhelp.cssjs( action, parentList.fwd,➥
    inlineSlides.hideLinkClass);
```

That took care of the links; now you need to increment the counter display. Since the counter is stored as a property of the list, it is easy to read the first child node of that property—that is, the text inside the SPAN. You can then use the `replace()` method of the `String` object to replace the first numerical entry (here via a regular expression) with the new image number, which is `count+1`—again, because humans do count from 1 and not from 0. Next, reset the `currentSlide` property, grab the new photo (remember you changed `count`), and show the current photo by adding the `show` class. All that is left to do is start the `init()` method when the window has loaded.

photoListInlineSlides.js *(excerpt)*

```
    photo = parentList.getElementsByTagName( 'li' )[count];
    var counterText = parentList.count.firstChild
    counterText.nodeValue = counterText.nodeValue.➥
    replace( /\d/, count + 1 );
    parentList.currentSlide = count;
    photo = parentList.getElementsByTagName( 'li' )[count];
    DOMhelp.cssjs( 'add', photo, inlineSlides.showClass );
    DOMhelp.cancelClick( e );
  }
}
DOMhelp.addEvent( window, 'load', inlineSlides.init, false );
```

However, you are not quite finished yet. If you try out the slide show in Safari, you will realize that the forward and backward links do get hidden, but they are still clickable and cause errors when you try to reach images that are not there.

■ **Caution** This is a common mistake in dynamic web development—hiding things visibly does not necessarily make them disappear for all users. Think of people who are blind, or users of textual browsers (such as Lynx). And then there are browser bugs and oddities to consider, too.

Preventing this issue is rather easy though: all you need to amend is the `showSlide()` method so as not to do anything when the target that was clicked has the `hide` CSS class assigned to it. And when fixing that you might as well add the Safari fix to cancel the default action of the newly generated links. The demo `examplePhotoListInlineSlideShowSafariFix.html` incorporates these changes:

photoListInlineSlidesSafariFix.js

```
inlineSlides = {

  // CSS classes
  slideClass : 'slides',
  dynamicSlideClass : 'dynslides',
  showClass : 'show',
  slideCounterClass : 'slidecounter',
  hideLinkClass : 'hide',
  // Labels
  // Forward and backward links, you can use any HTML here
  forwardsLabel : '<img src="control_fastforward_blue.png"➥
  alt="next" />',
  backwardsLabel : '<img src="control_rewind_blue.png"➥
  alt="previous" />',
  // Counter text, # will be replaced by the current image count
  // and % by the number of all pictures
  counterLabel : '# of %',

  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
    var uls = document.getElementsByTagName( 'ul' );
    for( var i = 0; i < uls.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', uls[i],➥
      inlineSlides.slideClass ) ) {
       continue;
      }
      DOMhelp.cssjs( 'swap', uls[i], inlineSlides.slideClass,➥
      inlineSlides.dynamicSlideClass );
      uls[i].currentSlide = 0;
      inlineSlides.initSlideShow( uls[i] );
    }
  },
```

```
initSlideShow : function( o ) {
  var p, temp, count;
  p = document.createElement( 'p' );
  DOMhelp.cssjs( 'add', p, inlineSlides.slideCounterClass );
  o.parentNode.insertBefore( p, o.nextSibling );
  o.rew = DOMhelp.createLink( '#', ' ' );
  o.rew.innerHTML = inlineSlides.backwardsLabel;
  DOMhelp.addEvent( o.rew, 'click', inlineSlides.showSlide, false );
  DOMhelp.cssjs( 'add', o.rew, inlineSlides.hideLinkClass );
  p.appendChild( o.rew );
  o.count = document.createElement( 'span' );
  temp = inlineSlides.counterLabel._
  replace( /#/, o.currentSlide + 1 );
  temp = temp.replace( /%/, o.getElementsByTagName( 'li' ).length );
  o.count.appendChild( document.createTextNode( temp ) );
  p.appendChild( o.count );
  o.fwd=DOMhelp.createLink( '#', ' ' );
  o.fwd.innerHTML = inlineSlides.forwardsLabel;
  DOMhelp.addEvent( o.fwd, 'click', inlineSlides.showSlide, false );
  p.appendChild( o.fwd );
  temp = o.getElementsByTagName( 'li' )[o.currentSlide];
  DOMhelp.cssjs( 'add', temp,inlineSlides.showClass );
  o.fwd.onclick = DOMhelp.safariClickFix;
  o.rew.onclick = DOMhelp.safariClickFix;
},
showSlide : function( e ) {
  var action;
  var t = DOMhelp.getTarget( e );
  while( t.nodeName.toLowerCase() != 'a'
      && t.nodeName.toLowerCase() != 'body' ) {
    t = t.parentNode;
  }
  if( DOMhelp.cssjs( 'check', t,_
  inlineSlides.hideLinkClass ) ){
   return;
  }
  var parentList = DOMhelp.closestSibling( t.parentNode, -1 );
  var count = parentList.currentSlide;
  var photoCount = parentList.getElementsByTagName( 'li' ).length-1;
  var photo = parentList.getElementsByTagName( 'li' )[count];
  DOMhelp.cssjs( 'remove', photo, inlineSlides.showClass );
  count = ( t == parentList.fwd ) ? count + 1 : count - 1;
  action = ( count > 0 ) ? 'remove' : 'add' ;
  DOMhelp.cssjs( action, parentList.rew,➥
  inlineSlides.hideLinkClass );
  action = ( count < photoCount ) ? 'remove' : 'add';
```

```
    DOMhelp.cssjs( action, parentList.fwd,➥
    inlineSlides.hideLinkClass );
    photo = parentList.getElementsByTagName( 'li' )[count];
    var counterText = parentList.count.firstChild
    counterText.nodeValue = counterText.nodeValue.➥
    replace( /\d/, count + 1 );
    parentList.currentSlide = count;
    DOMhelp.cssjs( 'add', photo, inlineSlides.showClass );
    DOMhelp.cancelClick( e );
  }
}
DOMhelp.addEvent( window, 'load', inlineSlides.init, false );
```

Turning embedded image lists into slide shows is an effect that degrades nicely on non-JavaScript user agents, although it is not really image manipulation or even dynamic. The real power of JavaScript is to avoid page reloads and show larger images in the same document instead of just showing them in the browser. Let's take a look at some examples.

Dynamic Slide Shows

Let's take another HTML list and turn it into an example of a dynamic slide show. Start with the HTML, this time a list containing thumbnail images linked to larger images:

exampleMiniSlides.html

```
<ul class="minislides">
<li>
  <a href="pictures/thumbs/cat2.jpg">
     <img src="pictures/minithumbs/cat2.jpg" alt="Lazy Cat" />
  </a>
</li>
<li>
  <a href="pictures/thumbs/dog63.jpg">
     <img src="pictures/minithumbs/dog63.jpg"_
     alt="Dog cooling off in the sand" /></a>
  </li>
  <li>
    <a href="pictures/thumbs/dog7.jpg">
      <img src="pictures/minithumbs/dog7.jpg"  alt="Very flat dog" />
    </a>
  </li>
  <li>
    <a href="pictures/thumbs/kittenflat.jpg">
      <img src="pictures/minithumbs/kittenflat.jpg" alt="Ginger and White Cat" />
    </a>
  </li>
</ul>
```

If you open the example in a browser with JavaScript enabled, you get a list of small thumbnails and a larger image. Clicking the thumbnails will replace the larger image with the one the thumbnail points to, as shown in Figure 6-9.

Figure 6-9. *A slide show with small preview images (thumbnails)*

Visitors without JavaScript will only get a row of images linking to larger images, as shown in Figure 6-10.

Figure 6-10. *The slide show with small preview images without JavaScript*

Again, let's plan the skeleton of the script: you define a class to recognize which lists are to become slide shows, a class to give to the list item that contains the large photo, and an alternative text to add to the large photo.

The methods are the same as the last time: one global initialization method, one to initialize each slide show, and one to show the current photo.

`miniSlides.js` *(skeleton)*

```
minislides = {
  // CSS classes
  triggerClass : 'minislides',
  largeImgClass : 'photo',
  // Text added to the title attribute of the big picture
  alternativeText : ' large view',

  init : function(){  },
  initShow : function( o ){ },
  showPic : function( e ){  }
}
DOMhelp.addEvent( window, 'load', minislides.init, false );
```

The CSS for the slide show is pretty simple:

miniSlides.css *(excerpt)*

```
.minislides, .minislides * {
  margin:0;
  padding:0;
  list-style:none;
  border:none;
}
.minislides{
  clear:both;
  margin:10px 0;
  background:#333;
}
.minislides,.minislides li{
  float:left;
}
.minislides li img{
  display:block;
}
.minislides li{
  padding:1px;
}
.minislides li.photo{
  clear:both;
  padding-top:0;
}
```

First, you do a global reset on anything inside the list with the right class and the list itself. A global reset means setting all margins and paddings to `0` and the borders and list styles to `none`. This prevents having to deal with cross-browser differences and also makes the CSS document a lot shorter, as you don't need to reset these values for every element.

Then float the lists and all the list items to the left to make them appear inline rather than one under the other. You need to float the main list to make sure it contains the others.

Set the images to display as block elements to avoid gaps around them and add padding on each list item of one pixel to show the background color.

The "photo" list item needs a float clearing to appear below the others. Set its top padding to `0` to avoid a double line between the smaller images and the big image.

The `init()` method functions analogously to the one in the last slide show. You test for DOM support, loop through all the lists in the document, and skip those that don't have the right class. The ones that have the right class get sent to the `initShow()` method as a parameter.

miniSlides.js *(excerpt)*

```
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
    var lists = document.getElementsByTagName( 'ul' );
    for( var i = 0; i < lists.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', lists[i],➥
      minislides.triggerClass ) ) {
       continue;
      }
      minislides.initShow( lists[i] );
    }
  },
```

The `initShow()` method starts by creating a new list item, a new image, and assigning the large image class to the new list item. It adds the image as a child of the list item and the new list item as a child of the main list. Together with the CSS defined earlier, this displays the new image below the others.

miniSlides.js *(excerpt)*

```
  initShow : function( o ) {
    var newli = document.createElement( 'li' );
    var newimg = document.createElement( 'img' );
    newli.appendChild( newimg );
    DOMhelp.cssjs( 'add', newli, minislides.largeImgClass );
    o.appendChild( newli );
```

Then you grab the first image in the list and read its alternative text stored in the `alt` attribute. Add this text as the alternative text to the new image, add the text stored in the `alternativeText` property to it, and store the resulting string as a `title` attribute of the new image.

miniSlides.js *(excerpt)*

```
    var firstPic = o.getElementsByTagName( 'img' )[0];
    var alt = firstPic.getAttribute( 'alt' );
    newimg.setAttribute( 'alt', alt );
    newimg.setAttribute( 'title', alt + minislides.alternativeText );
```

Next, retrieve all the links in the list and apply an event pointing to `showPic` when the user clicks each link. Store the new image as a property called `photo` in the list object and set the `src` attribute of the newly created image to the target location of the first link.

miniSlides.js *(excerpt)*

```
    var links = o.getElementsByTagName( 'a' );
    for(i = 0; i < links.length; i++){
      DOMhelp.addEvent( links[i], 'click', minislides.showPic,➥
      false );
      links[i].onclick = function() { return false; } // Safari
    }
    o.photo = newimg;
    newimg.setAttribute( 'src', o.getElementsByTagName('a')[0].href );
  },
```

Showing the pictures the links point to when they were clicked is a piece of cake. In the `showPic()` method, retrieve the event target via `getTarget()` and get the old picture by reading out the list's `photo` property.

This time you know the element the visitor will click is an image, which is why you don't need to loop and test the name of the element. Instead, you go up three parent nodes (`A`, `LI`, and `UL`) and read the previously stored image. Then you set the alternative text, the title, and the image `src` and stop the link's default behavior via `cancelClick()`. Add a handler that fires the `init()` method when the window has finished loading to complete your mini slide show.

miniSlides.js *(excerpt)*

```
  showPic : function( e ) {
    var t = DOMhelp.getTarget( e );
    var oldimg = t.parentNode.parentNode.parentNode.photo;
    oldimg.setAttribute( 'alt', t.getAttribute( 'alt' ) );
    oldimg.setAttribute( 'title', t.getAttribute('alt') +➥
    minislides.alternativeText );
    oldimg.setAttribute( 'src', t.parentNode.getAttribute( 'href' ) );
    DOMhelp.cancelClick( e );
  }
}
DOMhelp.addEvent( window, 'load', minislides.init, false );
```

Summary of Images and JavaScript

This concludes this introduction to images and JavaScript. Hopefully, you are not too overwhelmed by what is possible and how to approach scripting in this fashion and are interested in playing with the examples to see what else is possible. Keep these slide shows and how they work in mind though—at the end of this chapter we will come back to the document embedded slide show and make it play automatically. In Chapter 10, you will see how to develop a larger JavaScript-enabled gallery.

Things to Remember About Images and JavaScript

- You can do a lot of preloading with the image objects, but it is not a bulletproof approach. Browser cache settings, broken image links, and proxy servers might mess with your preload script. A lot of DHTML scripts use image preloader progress bars before showing the main page. If these scripts fail, all the user gets is a stuck or constantly changing progress bar, which is frustrating.
- While you can access the properties of each image directly, it might not be the best approach—leave the visuals to the CSS and your scripts will not be changed by others who might not know what is going on.
- CSS has come a long way since the days of DHTML, and these days it might be a better approach to help the CSS designer by providing hooks rather than achieving a visual effect purely by scripting—an example being the parent rollover you saw earlier.

Windows and JavaScript

Spawning new browser windows and altering the current window are very common uses for JavaScript. These are also very annoying and unsafe, as you can never be sure if the visitor of your web page can deal with resized windows or will be notified by her user agent when there is a new window. Think of screen reader users listening to your site or text browser users.

Windows have been used for unsolicited advertising (pop-up windows) and executing code in hidden windows for data retrieval purposes (phishing) in the past, which is why browser vendors and third-party software providers have come up with a lot of software and browser settings to stop this kind of abuse. Mozilla Firefox users can choose whether they want pop-up windows and what properties of the window can be changed by JavaScript as shown in Figure 6-11.

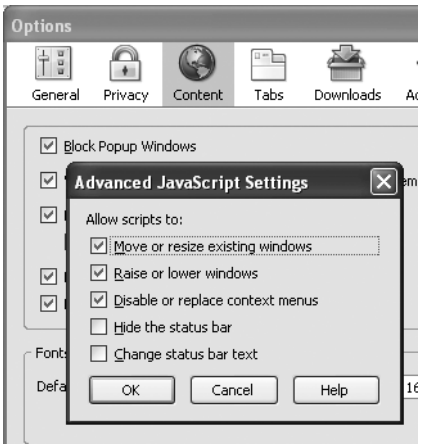

Figure 6-11. *The advanced JavaScript settings in Mozilla Firefox*

Other browsers like MSIE 7 or Opera 8 disallow hiding the location bar in new windows and can impose size and position constraints on new windows.

■ **Note** This is a point of agreement between different browser makers wanting to stop security vulnerabilities. Opening a new window without a visible location bar can allow a malicious attacker to open a pop-up on a third-party site via Cross-Site Scripting (XSS), make it appear as if the window belonged to the site, and ask users to enter information. More about XSS can be found on Wikipedia: `http://en.wikipedia.org/wiki/XSS`.

This is all great news for web surfers, as they can stop unwanted advertising and are less likely to be fooled into giving away their data to the wrong people. It is not such great news for you, as it means that when you want to use windows extensively in JavaScript, you have to test a lot of different scenarios.

Such considerations aside, you can still do a lot with windows and JavaScript. Every JavaScript-capable browser provides you with an object called `window`, whose properties are listed in the next section, as well as some examples on how to use them.

Window Properties

■ **Note** The following list does not show all the `window` properties available, only those that are supported across numerous browsers (Mozilla/Firefox, Opera 7 and higher, MSIE 5.5 and higher, and Safari). If a given property is not supported by a certain browser, I will note that where the property is discussed.

- `closed`: Boolean if the window was closed or not (read-only)
- `defaultStatus`: The default status message in the status bar (not supported by Safari)
- `innerHeight`: The height of the document part of the window
- `innerWidth`: The width of the document part of the window
- `outerHeight`: The height of the whole window
- `outerWidth`: The width of the whole window
- `pageXOffset`: Current horizontal start position of the document inside the window (read-only)
- `pageYOffset`: Current vertical start position of the document inside the window (read-only)
- `status`: The text content of the status bar
- `name`: The name of the window
- `toolbar`: Property that returns an object with a visible property of `true` when the window has a toolbar (read-only)

For example, if you want to obtain the inner size of a window, you could use some of these properties:

exampleWindowProperties.html *(excerpt)*

```
function winProps() {
  var winWidth = window.outerWidth;
  var winHeight = window.outerHeight;
  var docWidth = window.innerWidth;
  var docHeight = window.innerHeight;
  var message = 'This window is ';
  message += winWidth + ' pixels wide and ';
  message += winHeight + ' pixels high.\n';
  message += 'The inner dimensions are:';
  message += docWidth + ' * ' + docHeight + ' pixels';
  alert( message );
}
```

Some possible output from this function is shown in Figure 6-12.

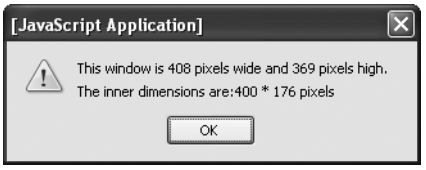

Figure 6-12. *Reading out window properties*

There are other properties that are not supported by MSIE and Opera. These are `scrollbars`, `locationbar`, `statusbar`, `menubar`, and `personalbar`. Each of them stores an object with a read-only `visible` property having a value of `true` or `false`. In order to test whether the user has a menu bar open, you check the object and the property:

```
if ( window.menubar && window.menubar.visible == true ){
 // Code
}
```

Window Methods

The `window` object also has a number of methods, some of which have been discussed in earlier chapters. The most common ones apart from those that provide user feedback are for opening new windows and timed execution of functions.

User Feedback Methods

The user feedback methods listed here are covered in detail in Chapter 4.

- `alert('message')`: Displays an alert
- `confirm('message')`: Displays a dialog to confirm an action
- `prompt('message','preset')`: Displays a dialog to enter a value

Opening New Windows

Opening and closing of windows is technically quite easy; however, with browsers suppressing different properties and methods or badly written pop-up blocking software even blocking "good" pop-ups, doing so can become a nightmare and needs proper testing. The specifics of opening new windows are pretty easy. You have four methods at your disposal:

- `open('url','name','properties')`: Opens a new window called "name", loads the URL in it, and sets the window properties
- `close()`: Closes the window (If the window is not a pop-up window, this will cause a security alert.)
- `blur()`: Moves the focus of browser away from the window
- `focus()`: Moves the focus of browser to the window

The `properties` string of the `open` method has a very unique syntax: it lists all the properties of the window as strings—each consisting of a name and a value joined with an equal sign—which are separated by commas:

```
myWindow = window.open( 'demo.html', 'my', 'p1=v1,p2=v2,p3=v3' );
```

Not all of the properties listed here are supported in all browsers, but they can be used in most of them:

- `height`: Defines the height of the window in pixels.
- `width`: Defines the width of the window in pixels.
- `left`: Defines the horizontal position of the window on the screen in pixels.
- `top`: Defines the vertical position of the window on the screen in pixels.
- `location`: Defines whether the window has a location bar or not (`yes` or `no`)—remember this will be a fixed `yes` state in future browsers.
- `menubar`: Defines whether the window has a menu bar or not (`yes` or `no`)—this is not supported by Opera and Safari (being a Mac tool, Safari has no menu bar on the window but on top of the screen).
- `resizable`: Defines whether the user is allowed to resize the window in case it is too small or too big. Opera does not allow this property, so Opera users can resize any window.
- `scrollbars`: Defines whether the window has scrollbars or not (`yes` or `no`). Opera does not allow suppressing scrollbars.
- `status`: Defines whether the window has a status bar or not (`yes` or `no`). Opera does not allow turning off the status bar.
- `toolbar`: Defines whether the window has a toolbar or not (`yes` or `no`)—turning the toolbar off is not supported by Opera.

■ **Note** Other Mozilla/Firefox-only properties are `hotkeys` (which turns the keyboard shortcuts on and off in a given window), `innerHeight` and `innerWidth` (which define the display size as opposed to window size), and `dependent` (which determines whether the window should be closed when the window that opened it is closed). The latter is also supported by Konqueror on Linux.

To open a window that is 200 pixels wide and high and 100 pixels from the top-left corner of the screen, and then load into it the document `grid.html`, you have to set the appropriate properties as shown here:

```
var windowprops = "width=200,height=200,top=100,left=100";
```

You can try to open the window when the page loads:

`exampleWindowPopUp.html` *(excerpt)*

```
function popup() {
  var windowprops = "width=200,height=200,top=100,left=100";
  var myWin =  window.open( "grid.html", "mynewwin" ,windowprops );
}
window.onload = popup;
```

Notice that the outcome is slightly different from browser to browser. MSIE 6 shows the window without any toolbars, MSIE 7 shows the location bar, and Firefox and Opera warn you that the page is trying to open a new window and asks you whether you want to allow it to do so.

This is handled slightly differently when the window is opened via a link:

exampleLinkWindowPopUp.html

```
<a href="#" onclick="popup();return false">
  Open grid
</a>
```

Neither Opera nor Firefox complain about the pop-up window now. However, if JavaScript is disabled, then there won't be a new window, and the link doesn't do anything. Therefore, you might want to link to the document in the `href` and send the URL as a parameter instead.

`exampleParameterLinkWindowPopUp.html` *(excerpts)*

```
function popup( url ) {
  var windowprops = "width=200,height=200,top=100,left=100";
  var myWin =  window.open( url, "mynewwin", windowprops );
}

<a href="grid.html" onclick="popup(this.href);return false">
  Open grid
</a>
```

Notice the `name` parameter of `window.open()` method. This one is seemingly pretty pointless for JavaScript, as it does not do anything (there is no `windows` collection that you could use to reach the window via `windows.mynewwin`, for example). However, it is used in the HTML `target` attribute to make links open their linked documents in the pop-up window instead of the main window.

In this example, you define the name of the window as "mynewwin", and you target a link to open `http://www.yahoo.co.uk/` in there:

`exampleParameterLinkWindowPopUp.html` *(excerpts)*

```
function popup( url ) {
  var windowprops = "width=200,height=200,top=100,left=100";
  var myWin = window.open( url, "mynewwin", windowprops );
}
<a href="http://www.yahoo.co.uk/" target="mynewwin">Open Yahoo</a>
```

Unless you use nontransitional XHTML or strict HTML (where `target` is deprecated), you can also use the `target` attribute with a value of `_blank` to open a new window regardless of the availability of JavaScript. It is good practice then to tell the visitor that the link is going to open in a new window inside the link to avoid confusion or accessibility issues:

```
<a href="grid.html" onclick="popup(this.href);return false" target="blank">
  Open grid  (opens in a new window)
</a>
```

However, as you might want to use strict HTML and XHTML, and you have a lot more control over the pop-up when using JavaScript, it might be a better solution to not rely on `target` but turn links into pop-up links only if and when scripting is available. For that you need something to hook into and identify the link as a pop-up link. You could for example use a class called `popup`.

`exampleAutomaticPopupLinks.html` *(excerpt)*

```
<p><a href="grid.html" class="popup">Open grid</a></p>
<p><a href="http://www.yahoo.co.uk/" class="popup">Open Yahoo</a></p>
```

Planning the script doesn't entail much: you need the class to trigger the pop-up, the textual add-on, and the window parameters as properties, an `init()` method to identify the links and add the changes, and an `openPopup()` method to trigger the pop-ups.

`automaticPopupLinks.js` *(skeleton)*

```
poplinks = {
  triggerClass : 'popup',
  popupLabel : ' (opens in a new window)',
  windowProps : 'width=200,height=200,top=100,left=100',
  init : function(){ },
  openPopup : function( e ){ },
}
```

The two methods are pretty basic. The `init()` method checks for DOM support and loops through all the links in the document. If the current link has the CSS trigger class, it adds the label to the link by creating a new text node from the label and adding it as a new child to the link. It adds an event when the link is clicked pointing to the `openPopup()` method and applies the Safari fix to stop the link from being followed in that browser.

automaticPopupLinks.js *(excerpt)*

```
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
     return;
    }
    var label;
    var allLinks = document.getElementsByTagName( 'a' );
    for( var i = 0; i < allLinks.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', allLinks[i],➥
      poplinks.triggerClass ) ) {
        continue;
      }
      label = document.createTextNode( poplinks.popupLabel );
      allLinks[i].appendChild( label );
      DOMhelp.addEvent( allLinks[i], 'click',➥
      poplinks.openPopup, false );
      allLinks[i].onclick = DOMhelp.safariClickFix;
    }
  },
```

The `openPopup()` method retrieves the event target, makes sure that it was a link, and opens a new window by calling `window.open()` with the event target's `href` attribute as the URL, a blank name, and the window properties stored in `windowProps`. It finishes by calling the `cancelClick()` method to stop the link from being followed.

automaticPopupLinks.js *(excerpt)*

```
  openPopup : function( e ) {
    var t = DOMhelp.getTarget( e );
    if( t.nodeName.toLowerCase() != 'a' ) {
      t = t.parentNode;
    }
    var win = window.open( t.getAttribute('href'), '',➥
    poplinks.windowProps );
    DOMhelp.cancelClick( e );
  }
```

There is more to this topic, especially when it comes to usability and accessibility concerns, and making sure that pop-up windows are used only when they really are opened and not blocked by some software or otherwise fail to open; however, going deeper into the matter is not within the scope of the current discussion. It suffices to say that it is not easy to rely on pop-ups of any kind in today's environment.

Window Interaction

Windows can interact with other windows using a number of their properties and methods. First of all there is `focus()` and `blur()`: the former brings the pop-up window to the front, the latter pushes it to the back of the current window.

You can use the `close()` method to get rid of windows, and you can reach the window that opened the pop-up window via the `window.opener` property. Suppose that you've opened two new windows from a main document:

```
w1 = window.open( 'document1.html', 'win1' );
w2 = window.open( 'document2.html', 'win2' );
```

You can hide the first window behind the other by calling its `blur()` method:

```
w1.blur();
```

■ **Note** In online advertising, the trick of opening an unsolicited window and hiding it immediately via `blur()` is called a **pop-under** and is considered by advertisers to be less annoying than pop-up windows, as they don't cover the current page. If you've ever found several windows you don't remember opening when you shut down the browser, this is likely what has happened.

You can close the window by calling its `close()` method:

```
w1.close();
```

If you want to reach the initial window from any of the documents in the pop-ups, you can do this via

```
var parentWin = window.opener;
```

If you want to reach the second window from the first window, you'll also need to go through the `window.opener`, as the second window was opened from this one:

```
var parentWin = window.opener;
var otherWin = parentWin.w2;
```

Notice that you need to use the variable name the window was assigned to, and not the window name.

You can use any of the window methods of any of the windows this way. Say for example you want to close `w2` from within `document1.html`; you do this by calling

```
var parentWin = window.opener;
var otherWin = parentWin.w2.close();
```

You can also call functions of the main window. If the main window has a JavaScript function that is called `demo()`, you could reach it from `document1.html` via

```
var parentWin = window.opener;
parentWin.demo();
```

■ **Caution** If you try to close the initial window with `window.opener.close()`, some browsers will give a security alert asking the user if he wants to allow that. This is another security feature to prevent site owners with malicious intentions spoofing a different web site. A lot of design agencies used to close the original browser in favor of a window with a predefined size—this is not possible any longer without the aforementioned security alert, and it might be a good idea to avoid this kind of behavior unless you want to scare or annoy your visitors.

Changing the Position and the Dimensions of a Window

Each of the methods in the following list has `x` and `y` parameters. `x` is the horizontal position and `y` the vertical position in pixels from the top left of the screen. The `moveBy()`, `resizeBy()`, and `scrollBy()` methods allow for negative values, which would move the window or the content to the left and up or make the window smaller by the stated amount of pixels.

- `moveBy(x,y)`: Moves the window by `x` and `y` pixels
- `moveTo(x,y)`: Moves the window to the coordinates `x` and `y`
- `resizeBy(x,y)`: Resizes the window by `x` and `y`
- `resizeTo(x,y)`: Resizes the window to `x` and `y`
- `scrollBy(x,y)`: Scrolls the window's content by `x` and `y`
- `scrollTo(x,y)`: Scrolls the window's content to `x` and `y`

If you check the example document `exampleWindowPosition.html`, you can test out the different methods as shown in Figure 6-13. Note that this example appears in Firefox. In Opera 8 or MSIE 7, the small window will have a location bar.

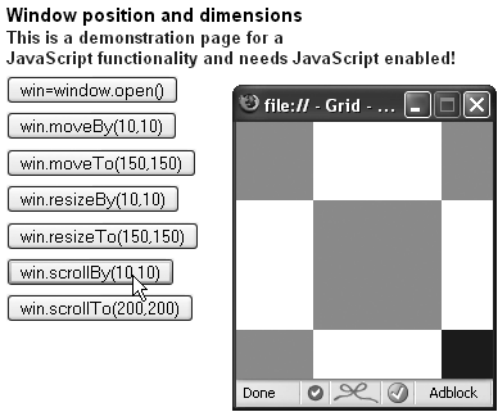

Figure 6-13. *Changing the window position and dimension*

Animation with Window Intervals and Timeouts

You can use the `setInterval()` and `setTimeout()` window methods to allow for timed execution of code. `setTimeout` means you wait a certain time before executing the code (once only); `setInterval()` executes the code every time the given time period has passed.

- `name = setInterval( 'someCode', x )`: Executes the JavaScript code passed to it as `someCode` every `x` milliseconds
- `clearInterval( name )`: Cancels the execution of the interval called `name` (keeps the code from being executed again)
- `name=setTimeout( 'someCode', x )`: Executes the JavaScript code `someCode` once, after waiting `x` milliseconds
- `clearTimeout( name )`: Stops the timeout called `name` if the code hasn't yet been run

---

■ **Note** The parameter `someCode` in `setInterval()` and `setTimeout()` is a string and can be any valid JavaScript code. Usually, it's most convenient simply to call a function you've already defined elsewhere.

Classic examples for the use of these methods are news tickers, clocks, and animation.

However, you can also use them to make your site less obtrusive and more user beneficial. One example of this is a warning message that vanishes after a certain amount of time. The demo `exampleTimeout.html` shows how you can use `setTimeout()` to display a very obvious warning message for a short period or allow the user to get rid of it immediately. The HTML has a paragraph warning the user that the document is out of date:

`exampleTimeout.html` *(excerpt)*

```
<p id="warning">This document is outdated
 and kept only for archive purposes.</p>
```

A basic style sheet colors this warning red and shows it in bold for non-JavaScript users. For users that have JavaScript enabled, add a dynamic class that makes the warning a lot more obvious.

`timeout.css`

```
#warning{
  font-weight:bold;
  color:#c00;
}
```

```
.warning{
  width:300px;
  padding:2em;
  background:#fcc;
  border:1px solid #c00;
  font-size:2em;
}
```

The difference between the two is shown in Figure 6-14.

Figure 6-14. *Warning message without and with JavaScript*

The user can click the "remove warning" link to get rid of the warning or wait, and it will vanish automatically after 10 seconds.

The script is pretty easy; you check whether DOM is supported and whether the warning message with the right ID exists. Then you add the dynamic `warning` class to the message and create a new link with an event handler pointing to the `removeWarning()` method. You append this link as a new child node to the warning message and define a timeout that automatically triggers `removeWarning()` when 10 seconds are over.

`timeout.js` *(excerpt)*

```
warn = {
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
    warn.w = document.getElementById( 'warning' );
    if( !warn.w ){ return; }
    DOMhelp.cssjs( 'add', warn.w, 'warning' );
    var temp = DOMhelp.createLink( '#', 'remove warning' );
    DOMhelp.addEvent( temp, 'click', warn.removeWarning, false );
    temp.onclick = DOMhelp.safariClickFix;
    warn.w.appendChild( temp );
    warn.timer = window.setTimeout( 'warn.removeWarning()', 10000 );
  },
```

All the removeWarning() method needs to do is remove the warning message from the document, clear the timeout, and stop the default action of the link.

timeout.js *(continued)*

```
  removeWarning : function( e ){
    warn.w.parentNode.removeChild( warn.w );
    window.clearTimeout( warn.timer );
    DOMhelp.cancelClick( e );
  }
}
DOMhelp.addEvent( window, 'load', warn.init, false )
```

One web application that pioneered this kind of effect is Basecamp (http://www.basecamphq.com/), which highlights the recent changes to the document in yellow when the page loads and gradually fades out the highlight. You can see the effect rationale at 37signals (http://www.37signals.com/svn/archives/000558.php) and a demo JavaScript at YourTotalSite (http://www.yourtotalsite.com/archives/javascript/yellowfade_technique_for/).

Timeouts are tempting to use on web sites because they give the impression of a very dynamic site and allow you to transition smoothly from one state to the other.

■ **Tip** There are several JavaScript effect libraries available that offer you premade scripts to achieve tansition and animation effects. While most of them are very outdated, there are some exceptions, like script.aculo.us (http://script.aculo.us/) and FACE (http://kurafire.net/projects/face).

However, it might be a good idea to reconsider using a lot of animation and transitions in your web site. Remember that the code is executed on the user's computer, and depending on how old or how busy it is with other tasks, the transitions and animations may look very clumsy and become a nuisance rather than a richer site experience.

Animations can also be an accessibility issue if the functionality of the site is dependent on them, as they might make it impossible for some groups of disabled visitors to use the site (cognitive disabilities, epilepsy). The Section 508 accessibility law (http://www.section508.gov/) states quite clearly that for software development you need to provide an option to turn off animation:

> *(h) When animation is displayed, the information shall be displayable in at least one non-animated presentation mode at the option of the user.*
>
> —http://www.section508.gov/index.cfm?FuseAction=Content&ID=12#Software

However, for web sites this is not formulated as clearly. The W3C accessibility guidelines, on the other hand, state explicitly in a level-two priority that you should avoid any movement in web pages.

Until user agents allow users to freeze moving content, avoid movement in pages.

—http://www.w3.org/TR/WCAG10-TECHS/#tech-avoid-movement

Let's try an example that allows the user to start and stop an animation. Take the embedded slide show developed earlier in the chapter and instead of giving it forward and backward links, you'll add a link to start and stop an automated slide show using a `setInterval()`.

The HTML and the CSS will stay the same, but the JavaScript has to change a lot.

If you open `exampleAutoSlideShow.html` in a browser, you'll get a slide show with a Play button that starts the show when you click it. You could easily start the animation when the page loads, but it is a good idea to leave the choice to the user. This is especially the case when you need to comply with accessibility guidelines, as unsolicited animation can cause problems for users suffering from disabilities like epilepsy. Once clicked, the button turns into a Stop button that stops the slide show when activated. You can see how it looks in Firefox in Figure 6-15.

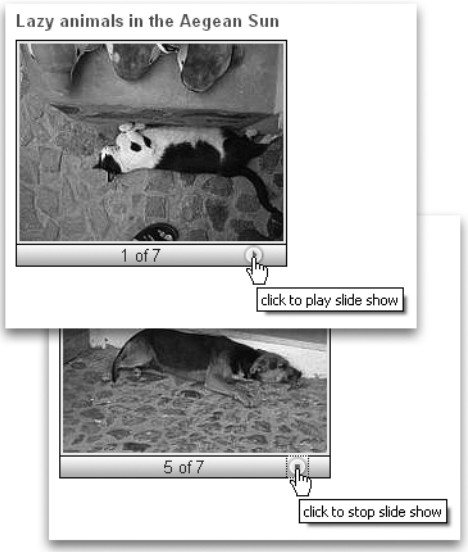

Figure 6-15. *An automatic slide show changing the Play button to a Stop button*

You start with the necessary CSS classes, which are the same as in the first slide show example with the exception of the `hide` class. As you won't hide any buttons this time, there is no need for it.

autoSlides.js *(excerpt)*

```
autoSlides = {

  // CSS classes
  slideClass : 'slides',
  dynamicSlideClass : 'dynslides',
  showClass : 'show',
  slideCounterClass : 'slidecounter',
```

The other properties change slightly; instead of backward and forward labels, you now need play and stop labels. The counter indicating which picture of how many is currently shown stays the same. One new property is the delay of the slide show in milliseconds.

autoSlides.js *(continued)*

```
  // Labels
  // Play and stop links, you can use any HTML here
  playLabel : '<img src="control_play_blue.png" alt="play" />',
  stopLabel : '<img src="control_stop_blue.png" alt="stop" />',
  // Counter text, # will be replaced by the current image count
  // and % by the number of all pictures
  counterLabel : '# of %',

  // Animation delay in milliseconds
  delay : 1000,
```

The `init()` method checks for DOM support and adds a new array called `slideLists` that will store all the lists that are to be turned into slide shows. This is necessary to be able to tell the function which list to apply the changes to.

autoSlides.js *(continued)*

```
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
    var uls = document.getElementsByTagName( 'ul' );
    autoSlides.slideLists = new Array();
```

First, loop through all the lists in the document and check for the class to turn them into slide shows. If a list has the class, you initialize the `currentSlide` property to `0` and store the loop counter in a new list property called `showCounter`. Once again, this will be needed to tell the interval which list to change. You call the `initSlideShow()` method with the list as a parameter and add the lists to the `slideLists` array.

autoSlides.js *(continued)*

```
    for( var i = 0; i < uls.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', uls[i],➥
      autoSlides.slideClass ) ){
        continue;
      }
      DOMhelp.cssjs( 'swap', uls[i], autoSlides.slideClass,_
      autoSlides.dynamicSlideClass );
      uls[i].currentSlide = 0;
      uls[i]. showIndex = i;
      autoSlides.initSlideShow( uls[i] );
      autoSlides.slideLists.push( uls[i] );
    }
  },
```

The `initSlideShow()` method does not differ much from the method of the same name that you used in `photoListInlineSlidesSafariFix.js`; the only difference is that you create one link instead of two and apply the `playLabel` as the content of the new link:

autoSlides.js *(continued)*

```
  initSlideShow : function( o ){
    var p, temp ;
    p = document.createElement( 'p' );
    DOMhelp.cssjs( 'add', p, autoSlides.slideCounterClass );
    o.parentNode.insertBefore( p, o.nextSibling );
    o.play = DOMhelp.createLink( '#', ' ' );
    o.play.innerHTML = autoSlides.playLabel;
    DOMhelp.addEvent( o.play, 'click', autoSlides.playSlide, false );
    o.count = document.createElement( 'span' );
    temp = autoSlides.counterLabel.replace( /#/, o.currentSlide + 1 );
    temp = temp.replace( /%/, o.getElementsByTagName( 'li' ).length );
    o.count.appendChild( document.createTextNode( temp ) );
    p.appendChild( o.count );
    p.appendChild( o.play );
    temp = o.getElementsByTagName( 'li' )[o.currentSlide];
    DOMhelp.cssjs('add', temp,autoSlides.showClass );
    o.play.onclick = DOMhelp.safariClickFix;
  },
```

The `playSlide()` method is new, but it starts pretty much like the old `showSlide()` method. You check the target and its node name and retrieve the parent list.

autoSlides.js *(continued)*

```
  playSlide : function( e ) {
    var t = DOMhelp.getTarget( e );
    while( t.nodeName.toLowerCase() != 'a'
        && t.nodeName.toLowerCase() != 'body' ){
      t = t.parentNode;
    }
    var parentList = DOMhelp.closestSibling( t.parentNode, -1 );
```

You test if the parent list already has a property called `loop`. This is the property that stores the instance of `setInterval()`. You use a property of the list instead of a variable to allow for more than one automated slide show in the same document.

You define the string to use in `setInterval()` as a call of the `showSlide()` method with the `showIndex` property of the parent list as a parameter. This is necessary as `setInterval()` is a method of the `window` object and not in the scope of the main `autoSlides` object.

You use `setInterval()` with the delay defined in the `autoSlides.delay` property and store it in the `loop` property before changing the content of the link that was activated to the Stop button.

autoSlides.js *(continued)*

```
    if( !parentList.loop ) {
      var loopCall = "autoSlides.showSlide( '" +➥
      parentList.showIndex + "' )";
      parentList.loop = window.setInterval( loopCall,➥
      autoSlides.delay );
      t.innerHTML = autoSlides.stopLabel;
```

If the list already has a property called `loop`, it means the slide show is currently running; therefore you clear it, set the `loop` property to `null`, and change the button back to the Play button. You then stop the default link behavior by calling `cancelClick()`.

autoSlides.js *(continued)*

```
    } else {
      window.clearInterval( parentList.loop );
      parentList.loop = null;
      t.innerHTML = autoSlides.playLabel;
    }
    DOMhelp.cancelClick( e );
  },
```

The showSlide() method changes quite drastically, but you will see that some of the initially confusing parts of the other methods (like what the slideLists array is good for) make the method rather easy.

Remember that you define in playSlide() that the interval should call the showSlide() method with the showIndex property of the list as a parameter. You can use this index now to retrieve the list you need to loop through simply by retrieving the list from the slideLists array.

autoSlides.js *(continued)*

```
  showSlide : function( showIndex ) {
    var currentShow = autoSlides.slideLists[showIndex];
```

Once you have the list, you can read out the current slide and the number of slides. Remove the showClass from the current slide to hide it.

autoSlides.js *(continued)*

```
    var count = currentShow.currentSlide;
    var photoCount = currentShow.getElementsByTagName('li').length;
    var photo = currentShow.getElementsByTagName( 'li' )[count];
    DOMhelp.cssjs( 'remove', photo, autoSlides.showClass );
```

Increment the counter to show the next slide. Compare the counter with the number of all the slides and set it to 0 if the last slide was already shown—thus restarting the slide show at the first slide.

Show the slide by retrieving the list element and adding the show class. Update the counter and reset the currentSlide property of the list to the new list element.

autoSlides.js *(continued)*

```
    count++;
    if( count == photoCount ){ count = 0 };
    photo = currentShow.getElementsByTagName( 'li' )[count];
    DOMhelp.cssjs( 'add', photo, autoSlides.showClass );
    var counterText = currentShow.count.firstChild;
    counterText.nodeValue = counterText.nodeValue.➥
    replace( /\d/, count + 1 );
    currentShow.currentSlide = count;
  }
}
DOMhelp.addEvent( window, 'load', autoSlides.init, false );
```

This complexity is just a taste of what awaits the JavaScript developer when it comes to animation and timed execution of code. Creating a smooth, stable, cross-browser animation in JavaScript is tricky and needs a lot of testing and knowledge of browser issues. Luckily, there are ready-made animation libraries available that help you with this task and have been tested for stability by a lot of developers with different operating systems and browsers. You'll get to know one of them with the examples in Chapter 11.

Navigation Methods of the Browser Window

Following is a list of methods for navigating a browser window:

- `back()`: Goes back one page in the browser history (If you use frames, this goes back to the last document without frames, not the last change inside a frame.)
- `forward()`: Goes one page forward in the browser history (If you use frames, this goes back to the last document without frames, not the last change inside a frame.)
- `home()`: Acts as if the user had clicked the home button (only for Firefox and Opera)
- `stop()`: Stops the loading of the document in the window (not supported by MSIE)
- `print()`: Starts the browser's print dialog

It is rather tempting to use these methods to provide navigation on pages that should simply link back to the previous page via something like the following:

```
<a href="javascript:window.back()">Back to previous page</a>
```

With accessibility and modern scripting in mind, this means that a user without JavaScript will have promised to him something that does not exist. A better solution is either to generate a real "back to previous page" link via server-side includes (SSIs) or to provide an actual HTML hyperlink to the right document. If neither is possible, use a placeholder and replace it with a generated link when and if JavaScript is available, as in the following example:

`exampleBackLink.html` *(to reach via* `exampleForBackLink.html`*)*

```
HTML:
<p id="back">Please use your browser's back button or
keyboard shortcut to go to the previous page</p>
JavaScript:
function backlink() {
  var p = document.getElementById( 'back' );
  if( p ) {
    var newa = document.createElement( 'a' );
    newa.setAttribute( 'href', '#' );
```

```
    newa.appendChild( document.createTextNode➥
    ( 'back to previous page' ) );
    newa.onclick = function() { window.back();return false; }
    p.replaceChild( newa, p.firstChild );
  }
}
window.onload = backlink;
```

■ **Caution** The danger of these methods is that you offer functionality that the browser already provides the user. The difference is that the browser does it better, as it does support more input devices. In Firefox on a PC, for example, you can print the document by pressing Ctrl+P, close the window or tab by pressing Ctrl+W, and move forward and backward in the browser history via Alt and the left or right arrow keys.

Even worse, the functionality offered with these methods is dependent on scripting support. It is up to you to decide if the preceding method—creating the links invoking these methods, which is probably the cleanest way to deal with the issue—is worth the effort, or if you should just allow the user to decide how to trigger browser functionality.

Alternatives to Opening New Windows: Layer Ads

Sometimes there is no way to avoid pop-up windows, as the site design or functionality requires them, and you just cannot make them work because of the browser issues and options explained earlier. One workaround is **layer ads**, which are basically absolutely positioned page elements put on top of the main content.

Let's try an example of those. Imagine your company wants to advertise the latest offers quite obviously when the page is loaded. The easiest way is to add the information at the end of the document and use a script to turn it into a layer ad, which means that visitors without JavaScript will still get the information, but nothing that covers the content without giving them a chance to get rid of it. The HTML can be a simple `DIV` with an ID (the real links have been replaced with "#" for the sake of brevity):

`exampleLayerAd.html` *(excerpt)*

```
<div id="layerad">
  <h2>We've got some special offers!</h2>
  <ul>
    <li><a href="#">TDK DVD-R 8x 50 pack $12</a></li>
    <li><a href="#">Datawrite DVD-R 16x 100 pack $50</a></li>
    <li><a href="#">NEC 3500A DVD-RW 16x $30</a></li>
  </ul>
</div>
```

The CSS designer can style the ad for the non-JavaScript version, and the script will add a class to allow the ad to show up above the main content. If you call the class dyn, the CSS might look like this:

layerAd.css *(excerpt)*

```
#layerad{
  margin:.5em;
  padding:.5em;
}
#layerad.dyn{
  position:absolute;
  top:1em;
  left:1em;
  background:#eef;
  border:1px solid #999;
}
#layerad.dyn a.adclose{
  display:block;
  text-align:right;
}
```

The last selector is a style for a dynamic link that the script will add to the ad, allowing the user to remove it.

The script itself does not contain any surprises. First, define the ID of the ad, the dynamic class, and the class and the text content of the "close" link as properties.

layerAd.js

```
ad = {
  adID : 'layerad',
  adDynamicClass : 'dyn',
  closeLinkClass : 'adclose',
  closeLinkLabel : 'close',
```

The `init()` method checks for DOM and for the ad and adds the dynamic class to it. It creates a new link and adds the text and the class of the "close" link to it. It adds an event handler to this link pointing to the `killAd()` method and inserts the new link before the first child node of the ad.

layerAd.js *(continued)*

```
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
```

```
    ad.offer = document.getElementById( ad.adID );
    if( !ad.offer ) { return; }
    DOMhelp.cssjs( 'add', ad.offer, ad.adDynamicClass );
    var closeLink = DOMhelp.createLink( '#', ad.closeLinkLabel );
    DOMhelp.cssjs( 'add', closeLink, ad.closeLinkClass );
    DOMhelp.addEvent( closeLink, 'click', ad.killAd,false );
    closeLink.onclick = DOMhelp.safariClickFix;
    ad.offer.insertBefore( closeLink, ad.offer.firstChild );
  },
```

The `killAd()` method removes the ad from the document and cancels the link's default behavior.

`layerAd.js` *(continued)*

```
  killAd : function( e ) {
    ad.offer.parentNode.removeChild( ad.offer );
    DOMhelp.cancelClick( e );
  }
}
DOMhelp.addEvent( window, 'load', ad.init, false );
```

You can test the effect by opening `exampleLayerAd.html` in a browser; if you have JavaScript enabled, you'll see the ad covering the content, as shown in Figure 6-16. You can get rid of it by using the "close" link.

Figure 6-16. *An example of a layer ad*

One other common use of pop-up windows is for displaying another document or file without leaving the current page. Classic examples include a long list of boring terms and conditions or a photo. Especially in the case of photos, a pop-up is a suboptimal solution, as you can open a window with the same dimensions as the picture, but there will be gaps around the image because the browser's internal styles have padding settings on the body. You could work around this problem by using a blank HTML document with a style sheet that sets the body margins and padding to 0 and adding the image to the document in the window via JavaScript. Another option is to show the photo in a newly generated and positioned element covering the main document. The demo `examplePicturePopup.html` does that; all the script needs is a class with a certain name on a link pointing to the photo.

examplePicturePopup.html *(excerpt)*

```
<a class="picturepop" href="pictures/thumbs/dog7.jpg">Sleeping Dog</a>
```

The script needs to do something that wasn't explained here before, namely reading the position of elements. You cover the main document with an element by positioning the element absolutely. As you don't know where the link pointing to the photo is in the document, you need to read its position and show the photo there.

But that is for later; first up you need to predefine properties. You need a class that triggers the script, applying a link class to the link when the photo is shown, and another class to the element containing the photo. You also need to define a prefix to be added to the link when the photo is displayed and a property acting as a shortcut reference to the newly created element.

picturePopup.js *(excerpt)*

```
pop={
  triggerClass:'picturepop',
  openPopupLinkClass:'popuplink',
  popupClass:'popup',
  displayPrefix:'Hide ',
  popContainer:null,
```

The `init()` method checks for DOM support and loops through all the links of the document, testing whether they have the right CSS class to trigger the pop-up. For those that do, the method adds an event handler pointing to the `openPopup()` method, and then stores the link's `innerHTML` content in a `preset` property.

picturePopup.js *(continued)*

```
  init : function() {
    if( !document.getElementById || !document.createTextNode ) {
      return;
    }
    var allLinks = document.getElementsByTagName( 'a' );
    for( var i = 0; i < allLinks.length; i++ ) {
      if( !DOMhelp.cssjs( 'check', allLinks[i],➥
      pop.triggerClass ) ) {
        continue;
      }
      DOMhelp.addEvent( allLinks[i], 'click', pop.openPopup, false );
      allLinks[i].onclick = DOMhelp.safariClickFix;
      allLinks[i].preset = allLinks[i].innerHTML;
    }
  },
```

The `openPopup()` method retrieves the event target and ensures that it is a link. It then tests whether there is already a `popContainer`, which would mean that the photo is shown. If this is not the case, the method adds the prefix to the link's content and adds the dynamic class to make the link look different.

picturePopup.js *(continued)*

```
  openPopup : function( e ) {
    var t = DOMhelp.getTarget( e );
    if( t.nodeName.toLowerCase() != 'a' ) {
      t = t.parentNode;
    }
    if( !pop.popContainer ) {
      t.innerHTML = pop.displayPrefix + t.preset;
      DOMhelp.cssjs( 'add', pop.popContainer, pop.popupClass );
```

The method then creates a new DIV that acts as the photo container, adds the appropriate class, and adds a new image as a child node of the container DIV. It shows the image by setting the src attribute of the new image to the value of the href attribute of the original link. The newly created photo container is then added to the document (as a child of the body element). Finally, openPopup() calls the positionPopup() method with the link object as a parameter.

picturePopup.js *(continued)*

```
      pop.popContainer = document.createElement( 'div' );
      DOMhelp.cssjs( 'add', t,pop.openPopupLinkClass );
      var newimg = document.createElement( 'img' );
      pop.popContainer.appendChild( newimg );
      newimg.setAttribute( 'src', t.getAttribute( 'href' ) );
      document.body.appendChild( pop.popContainer );
      pop.positionPopup( t );
```

If the popContainer already exists, all the method does is call the killPopup() method, reset the link to its original content, and remove the class that indicates that the photo is shown. Calling cancelClick() prevents the link from simply showing the photo in the browser.

picturePopup.js *(continued)*

```
    } else {
      pop.killPopup();
      t.innerHTML = t.preset;
      DOMhelp.cssjs( 'remove', t,pop.openPopupLinkClass );
    }
    DOMhelp.cancelClick( e );
  },
```

The positionPopup() method defines two variables, x and y, initializes them both to 0, and then reads the height of the element from its offsetHeight property. It next reads the vertical and horizontal position of the element and all its parent elements and adds those to x and y. The result is the position of the element relative to the document. The method then positions the photo container below the original link by adding the height of the link to the vertical variable y and altering the popContainer style properties.

picturePopup.js *(continued)*

```
  positionPopup : function( o ) {
    var x = 0;
    var y = 0;
    var h = o.offsetHeight;
    while ( o != null ) {
      x += o.offsetLeft;
      y += o.offsetTop;
      o = o.offsetParent;
    }
    pop.popContainer.style.left = x + 'px';
    pop.popContainer.style.top = y + h + 'px';
  },
```

The `killPopup()` method removes the `popContainer` from the document—clearing the property by setting its value to `null`—and prevents the default link action from taking place by calling `cancelClick()`.

■ **Note** You can remove a node from the document by calling the `removeChild()` method of its parent node, with the node itself as the child node to remove. However, as you use a property that points to the node rather than checking for the node itself, you also need to set this property to `null`.

---

picturePopup.js *(continued)*

```
  killPopup : function( e ) {
    pop.popContainer.parentNode.removeChild( pop.popContainer );
    pop.popContainer = null;
    DOMhelp.cancelClick( e );
  }
}
DOMhelp.addEvent( window, 'load', pop.init, false );
```

The result is that you can click any image pointing to a photo with the correct class, and it'll show the image below it. Figure 6-17 shows an example.

Figure 6-17. *An example of a dynamically displayed photo*

The beauty of this approach is that it is not only limited to photos. With just a simple modification, you can display other documents inside the current one. The trick is to dynamically add an `IFRAME` element to the `photoContainer` and set its `src` attribute to the document you want to embed. The demonstration `exampleIframeForPopup.html` does exactly that to show a lengthy terms-and-conditions document inside the main one.

The only differences (apart from different property names, as the method does not show a photo) are in the `openPopup` method where you add the new `IFRAME`:

`iframeForPopup.js` *(excerpt)*

```
var ifr = document.createElement( 'iframe' );
pop.ifrContainer.appendChild( ifr );
ifr.setAttribute( 'src', t.getAttribute( 'href' ) );
```

Figure 6-18 shows what this could look like.

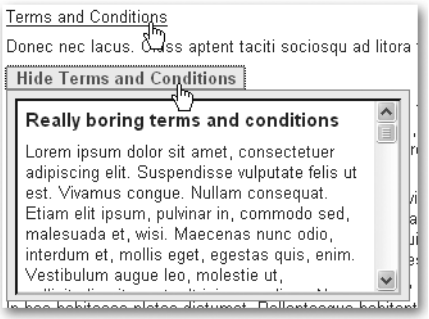

Figure 6-18. *An example of a dynamically included and displayed document*

Including other documents via `IFRAME` elements is an easy and well-supported method, but it is not the most accessible way. What you could do instead is use a server-side language like PHP to retrieve the content of the document and include it in the current one and use the same trick the layer ad example earlier uses. For more modern browsers, you can also do this "on the fly" using Ajax, but that will be explained in its own chapter, Chapter 8.

Summary: Windows and JavaScript

Controlling the current window and opening new ones has traditionally been a large part of JavaScript development, especially when it comes to web application development. In recent years, however, due to an increase in restrictions fueled by concerns about browser security and web surfers simply being fed up with lots of pop-ups and installing blocking software, it has become progressively harder to use new windows—even if you want to use them for a good cause. These and accessibility concerns make using multiple browser windows an increasingly unreliable method of web communication. Luckily, thanks to some of the replacement techniques discussed here (as well as Ajax, to which I devote Chapter 8), there is hardly any need for them any longer.

Things to Remember About Windows and JavaScript

- Test, test, and test again whether the window you have opened really exists before you try to do anything to it.
- Always remember that although the windows are on the same screen, they are completely separate instances of the browser. You need to go via the `window.opener` if you want to reach one pop-up from another or a function in the main script from any of the pop-ups you opened.
- Refrain from trying to control windows by taking away toolbars, moving them around the screen, or showing and hiding them via `blur()` and `focus()`. Most of these capabilities are still available now but are very likely to be blocked in future browsers.
- The web markup language of the future—XHTML—does not support the concept of different browser window instances (which is why the `target` attribute is deprecated). While you can circumvent this problem with scripting, it is still a bad idea to use XHTML STRICT and rely on several windows.
- You can simulate a lot of browser behavior or interactive elements with the window object methods—like closing and printing a window or moving back in the browser history. However, it might be better to leave this choice to the user. If you want to offer your own controls to the user, create these with JavaScript, too. Otherwise, the user will get a promise you don't keep when JavaScript is unavailable.

- If you use pop-up windows, tell visitors in the link opening the window that there will be a new window. This informs visitors with user agents that don't necessarily support multiple windows that there might be a change they have to deal with; it also prevents visitors from accidentally closing the window your web site needs. Years of unsolicited advertising and pop-ups have conditioned web surfers to immediately close new windows without even glancing at them.
- Employing timed execution of code via the `window` methods `setTimeout()` and `setInterval()` is a bit like using makeup: as a girl you learn how to put it on; as a woman you learn when to take it off. You can use both of these methods—and they are tempting to create all kind of snazzy effects—but you should think of your users, and ask yourself, "Is there really a need for animation when a static interface might lead much more quickly to the same result?"

Summary

Well done! You have reached the end of this chapter and should be well equipped now to create your own JavaScript solutions with images and windows, or—even better—window replacement techniques.

If some of the examples were a bit tough to wrap your head around, don't get frustrated, as this is a feeling you'll experience often as a JavaScript developer. And it does not mean that you don't "get it." Dozens of ways exist to solve any given problem in JavaScript. While there are much easier ways than those explained here, getting used to this kind of scripting should make you well equipped for more advanced scripting tasks like using third-party APIs or web application development.

In the next chapter, you have the chance to become better acquainted with event and property handling, as we discuss navigation and forms.