From DHTML to DOM scripting

Written by Christian Heilmann March 2006

Please visit this document's homepage at
http://icant.co.uk/articles/from-dhtml-to-dom/
for downloads and updates.

This document is delivered for free and may not be sold without the author's consent.
If you want to reproduce parts in own publications, please contact the author.

Table of Contents

From DHTML to DOM scripting 1
- What is DHTML and what is DOM scripting? 2
  - What is the DOM? 3
- Creating the page in DHTML 6
  - The goal of DHTML 6
  - The tabbed navigation in DHTML 6
  - DHTML issue #1: Script dependence 7
  - Showing, hiding and tab highlight in DHTML 7
  - DHTML issue #2: Mixing presentation and functionality 9
  - The Product Photo in DHTML 10
  - DHTML issue #3: Assuming functionality without testing for it 11
  - The Slide Show in DHTML 12
  - DHTML issue #5: Keeping maintenance JavaScript based 14
  - DHTML issue #6: Mixing HTML and JavaScript 14
  - DHTML issue #7: Blaming the user 14
  - DHTML issue #8: Taking over the document 15
- Introducing DOM scripting 16
  - The goal of DOM scripting 17
  - DOM scripting asset #1: Progressive Enhancement 18
  - DOM scripting asset #2: Ease of maintenance 20
  - DOM scripting asset #3: Separation of Presentation and Behaviour 21
  - DOM scripting asset #4: Separation of Structure and Behaviour 22
  - DOM scripting asset #5: Using modern event handling 24
  - DOM scripting asset #6: Avoiding clashes with other scripts 26
- Re-creating the demo page with DOM scripting 29
  - The tab navigation in DOM scripting 32
  - The popup window in DOM scripting 35
  - The slide show in DOM scripting 37
  - Fixing for Safari 40
- Where to now? 40

In this article we will try to help JavaScript beginners to spot old and outdated JavaScript techniques and explain what their issues are. We do this by looking at a web page that might have been developed around the millennium with development ideas that were good at the time (DHTML), but result in inaccessible or even broken pages in today's web environment.
This article is aimed at developers who are new to JavaScript, or those who haven't touched it in a while and wonder why people tell them off for using techniques that were the bee's knees in 1999.

We will take a demonstration page that features three dynamic elements using JavaScript, take a look at how they were achieved, and give an example of a modern way of re-creating them more future-proof and less in the way of the visitor (DOM scripting).

Let's start with a bit of background knowledge about what DHTML and the DOM are.

What is DHTML and what is DOM scripting?

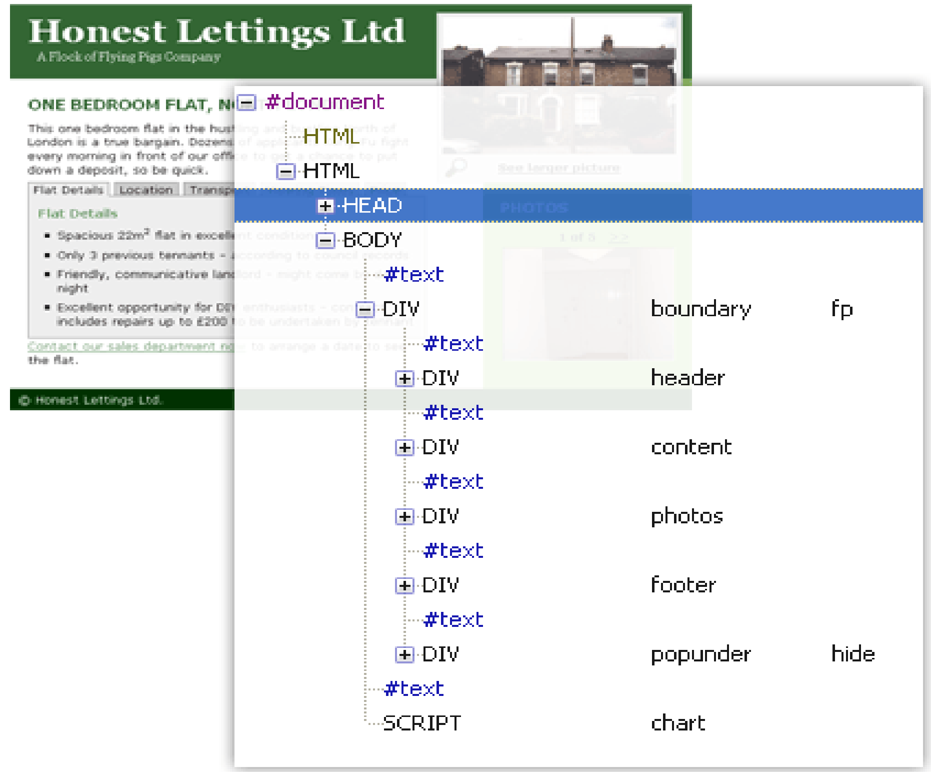
All technologies used on the web that are to become a worldwide standard have to be agreed by the World Wide Web Consortium (W3C) which can be found at http://www.w3.org. DHTML or Dynamic HTML is not one of these standards or a standalone technology. DHTML means using the web standards JavaScript, CSS and HTML together to create web pages that appear to be dynamic.

They only appear to be dynamic as they change their look without having to reload. Real dynamic pages on the other hand retrieve and send data from and to the backend or other systems. (This is something you can do nowadays without reloading the full page as well, using XHR or AJAX but we won't talk about that here as it is a different topic)

Making pages dynamic in the sense of making the user interface slicker and avoiding page reloads is a good thing, however as DHTML never really matured it wasn't done properly. This is partly because of the browser environment at that time. At the high time of DHTML browsers didn't support the standard Document Object Model (DOM) but had own implementations of the same idea.

What is the DOM?

The DOM is a programmable representation of a document. While we see a page as a visual construct, the browser recognises it as a tree structure of elements and their properties:



The DOM recognizes every part of the document as a node, and a node can be various things: A text, an element or an attribute. In XML it can be a lot more but these three are enough for HTML examples.

The DOM offers methods to reach all of these elements and their attributes and change them, remove them, shift them around or even create new ones. It functions on a node basis, which means you that cannot do something like “reach the first word in the third paragraph and replace it with ‘badger’” using DOM methods exclusively. You can however reach the third paragraph element and read its text value though. Then you use string manipulation to change the first word.

In DHTML you might have done this the following way:

```
isIE=document.all?true:false;
isNS4=document.layers?true:false;
isNS6=document.getElementById?true:false;
isOpera=navigator.appName.indexOf('opera')!=-1?true:false;
if(isIE && !isOpera){
  content=document.all.tags('p')[2].innerText;
  content=content.replace(/^\w+/,'badger');
  document.all.tags('p')[2].innerText=content;
}
if(isNS6 && !isOpera){
  content=document.getElementsByTagName('p')[2].innerHTML;
  content=content.replace(/^\w+/,'badger');
  document.getElementsByTagName('p')[2].innerHTML =content;
}
if(isNS4 || isOpera){
  alert('Your browser is not supported, please download a newer one');
}
```

- You'd have tested which browser is in use by testing which DOM is supported: MSIE had the Microsoft `document.all` DOM, Netscape 4 had an own implementation via `document.layers` and Netscape 6 was the first browser to support the W3C `document.getElementById` DOM.
- You'd then use the variables from this test as a means to execute different code for the different browsers.
- Both code examples read the third paragraph's `innerHTML` or `innerText` property, replace the word and write it back to the paragraph.

These were the better scripts – older ones than that would read out the name of the browser and test the name and version to send different code to different browsers. In the above example we did this with Opera.

In modern DOM scripting, you could do this via the following code:

```
if(!document.getElementsByTagName){return;}
var paragraphs=document.getElementsByTagName('p');
if(paragraphs.length>=3){
  var content=paragraphs[2].firstChild.nodeValue;
  content=content.replace(/^\w+/,'badger');
  paragraphs[2].firstChild.nodeValue=content;
}
```

- You test if the W3C DOM is available and if not you stop the rest of the script from executing (via the `return` statement)
- You retrieve all the paragraphs in the document via the `getElementsByTagName()` method.
- You test if the there are at least three paragraphs by reading the length of the paragraphs array and comparing it with 3
- If that is the case, you read the content of the paragraph and replace the word.

Can you already spot the difference? Let's try a real life example. Imagine you were asked to create a web page with a tabbed navigation, an image slide show and a preview image of a product that shows a large picture when you click it.

The [example page] has all of these and you can take a look by opening it in another browser window. We will now go through the code of it as it could have been done in DHTML and flag up issues that arise as we go along.

Creating the page in DHTML

The goal of DHTML

The main goal of DHTML was to make the formerly impossible possible – make pages look very dynamic, move things around on the click of a mouse or with timed animation and generally make web pages more engaging.

Before JavaScript and proper support in browsers this was science fiction – any change to the document meant a reload. The main problem of DHTML was that the browsers in use were prone to change quickly, and all of them followed a different path when it comes to providing the programmer with a DOM to change the page. This is why you will find a lot of browser testing and repeated code in DHTML scripts.

In the example page we omitted these tests, partly because they are unnecessary these days unless you work on a project that needs to support Netscape 4.x or MSIE 4. In order to keep the example easy to understand we omitted all the outdated browser tests and support for the MSIE 4 DOM (`document.all`) and Netscape 4 DOM (`document.layers`) in favor of the W3C DOM; real old code would be twice as big.

Let's take a look at the source code of the different effects:

The tabbed navigation in DHTML

```
<table id="flatdetails" border="0" width="420" cellpadding="5"
cellspacing="0">
<thead>
  <tr>
    <td><a href="javascript:void(showtab(0))">Flat Details</a></td>
    <td><a href="javascript:void(showtab(1))">Location</a></td>
    <td><a href="javascript:void(showtab(2))">Transport</a></td>
    <td><a href="javascript:void(showtab(3))">Viewing Times</a></td>
    <td><a href="javascript:void(showtab(4))">Price</a></td>
  </tr>
 </thead>
  <tbody>
    <tr id="details" style="display:none;"><td colspan="5">
     [… code for the different content sections…]
   </tbody>
 </table>
```

Notice that the `void()` statement is something that you are not likely to find in older pages, but we needed to add it to make this work with modern browsers.

DHTML issue #1: Script dependence

This shows one of the biggest problems with DHTML: It relies on JavaScript being available. The above navigation will not do anything in browsers that have JavaScript disabled, or if a script error in another function causes the script to stop executing. As we hide the table rows with `display:none` and expect JavaScript to undo this, users without JavaScript or a browser that doesn't understand some of the functionality will end up with a blank content section with 5 links that don't do anything when you click them:

ONE BEDROOM FLAT, NORTH LONDON

This one bedroom flat in the hustling and bustling North of London is a true bargain. Dozens of applicants Kung-Fu fight every morning in front of our office to get a chance to put down a deposit, so be quick.

Flat Details Location Transport Viewing Times Price

Contact our sales department now to arrange a da[illegible] to see the flat.

Showing, hiding and tab highlight in DHTML

However, if JavaScript is available, we can use the methods the DOM offers us to hide and show elements.

The DOM offers us various methods we can use:

- `var myNode=document.getElementById('id')` retrieves the node with the given id and stores it in the variable `myNode`.
- `var myNodes=document.getElementsByTagName('name')` retrieves all the elements with the given name (which could be for example 'p') and stores them as an array in the variable `myNodes`. We can then loop through all the elements in a `for` loop and do something to each of them. We use the `length` attribute of the array to learn how many elements there are:

```
// get all paragraphs from the document
var myParas=document.getElementsByTagName('p');
// store how many there are
var allParas=myParas.length;
// loop through all paragraphs
for(var i=0;i<allParas.i++){
  // get the current one
  currentPara=myParas[i];
  // code that does something with the current paragraph.
}
```

By using these two methods and the style collection it is rather easy to make the tabs work. The style collection contains all the style information of the node and allows you to change style properties dynamically. If you wanted to change the color of the second paragraph in the document to red you could use:

```
var myParas=document.getElementsByTagName('p');

myParas[1].style.color='red';
```

In the tab example we use this to show and hide the tabs and to color the current tab:

(### indicates a line-break we had to add to make this code readable)

```
function showtab(tab){
  // get all rows in the first body
  trs=document.getElementsByTagName('tbody')[0].      ###
  getElementsByTagName('tr');

  // loop through the rows and hide them
  for(i=0;i<trs.length;i++){
    trs[i].style.display='none';
  }
  // show the TR that was sent as a parameter
  trs[tab].style.display='';

  // get all cells in the thead
  trs2=document.getElementsByTagName('thead')[0]. ##
  getElementsByTagName('td');

  // loop through the cells and set their background
  // and text alignment
  for(i=0;i<trs2.length;i++){
    trs2[i].style.background='#fff';
    trs2[i].style.textAlign='center';
  }
  // set the background of the current tab
  trs2[tab].style.background='#ccc';
}
```

We retrieve the all the TR elements in the first TBODY element and loop through them. These are the different content sections in the document, for example the first one:

```
<tr style="display:none;">
  <td colspan="5">
    <h3>Flat Details</h3>
    [… details snipped …]
  </td>
</tr>
```

We use the style collection in JavaScript to set their display property to `none` – effectively hiding them. This might seem superfluous at first, as the inline styles on the TR element already did that. It does make sense though when the visitor clicks a different tab as we simply hide all of them before showing the currently chosen one.

```
// get all rows in the first body
  trs=document.getElementsByTagName('tbody')[0]. ###
  getElementsByTagName('tr');

  // loop through the rows and hide them
  for(i=0;i<trs.length;i++){
    trs[i].style.display='none';
  }
```

We show the current section that was sent via the `tab` parameter by setting its display property no nothing. This makes the browser fall back to the default display of the element in question, in this case `table-row`. If we set it to `block` like a lot of older scripts do, Firefox would not display the table properly.

```
  // show the TR that was sent as a parameter
  trs[tab].style.display='';
```

We do the same for the TD elements in the navigation table; however, in this case we change the background color instead of the display property. We also center the tab text by setting the `textAlign` property of the style collection accordingly. This makes it dead easy for us as JavaScript developers, but is also dangerous in terms of maintenance.

DHTML issue #2: Mixing presentation and functionality

Imagine you'd hand over the demo page we develop here to a client or another developer for maintenance. Some months down the line the colors or the sizes have to change and the developer will have to go through the JavaScript and change all the parts where you meddled with the style collection. Wouldn't it be a lot easier just to apply a different style sheet to change the look and feel?

The Product Photo in DHTML

The photo on the top right should show a large picture when the user clicks it. As we don't want the user to leave the page, we traditionally use a popup window for this. The code of the product section does exactly that:

```
<p>
  <a href="#"
  onclick="openwin('img/flat.jpg',700,470);return false;">
  <img src="img/flatshot.jpg" alt="Photo of the flat" />
  </a>
</p>
<p class="caption">
  <a href="#"
  onclick="openwin('img/flat.jpg',700,470);return false;">
  See larger picture
  </a>
</p>
```

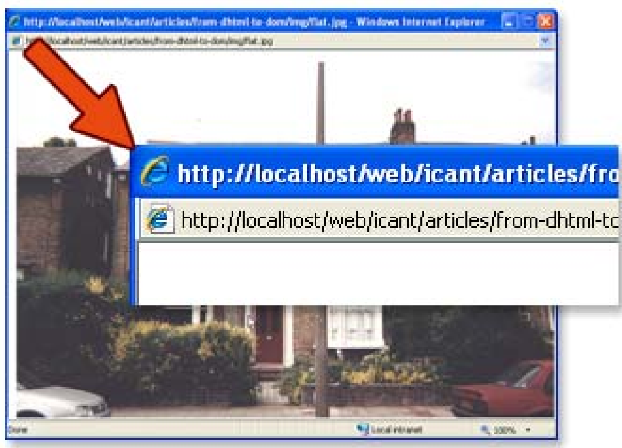
We use an event handler here called `onclick`. This attribute of the link calls the function `openwin` when the user activates the link. We stop the link from being followed by adding `return false` at the end.
The function `openwin` takes the three parameters: the image URL, its width and its height and uses them to center the popup window on the screen:

```
function openwin(url,x,y){
  pop = window.open(url,'pop',"height="+y+",width="+x+",status=no, ##
            toolbar=no,menubar=no,location=no");
  pop.moveTo((screen.width-x)/2,(screen.height-y)/2);
  pop.document.body.style.margin=0;
  pop.document.body.style.padding=0;
 }
```

We use the x and y dimensions passed on by the event handler to set the width and height in the `window.open()` method. Then we center the window by moving it via `window.moveTo()`. We calculate the location by reading the resolution of the user's computer via `screen.width` and `screen.height`, subtract the image dimensions and half the result. Lastly, we set the padding and margin of the document inside the popup to 0.
This has been a common way to show images in new windows for quite a while now, but popup windows have several issues:

- They might be blocked by software (although this one shouldn't as the user chose to open it instead of getting an unsolicited popup, but you never know how strict or badly thought-through the pop-up blocker software is).

- Users are conditioned by years of popup advertising to see every new window as a nuisance and might close it immediately
- Browser makers have agreed that the upcoming browsers should not allow the window location to be hidden for reasons of privacy and security. Right now, a malicious coder could use cross-server-scripting (XSS) to show a popup in an insecure web site that asks the user to enter information that will be sent to his server. If the URL of the window is visible this attack would be more obvious and less likely to fool the user. Opera 8 and the MSIE 7 beta already follow this agreement and our example on MSIE 7 beta looks like this:



DHTML issue #3: Assuming functionality without testing for it

The popup example is a classic example of the dangers of not testing what you want to achieve. MSIE7 disallows the moving of windows via the `moveTo()` method and throws an error stating "*Access is denied*". This could thoroughly spook out a visitor who doesn't know what you are trying to achieve and who read one too many Virus and Trojan vulnerability news article that morning. Many DHTML scripts don't test anything but assume the functionality is there.
The example popup script also has a link pointing to "#" instead of the image. It could be improved immensely by adding a real link to the image and use the `this` keyword to re-use this data:

```
<p class="caption">
  <a href="img/flat.jpg"
  onclick="openwin(this.href,700,470);return false;">
  See larger picture
  </a>
</p>
```

This would enable visitors without JavaScript to reach the large photo.

The Slide Show in DHTML

The slide show located below the product shot takes five pictures and allows the user to click through them with previous and next links. A counter indicates which picture of how many is shown at the moment.

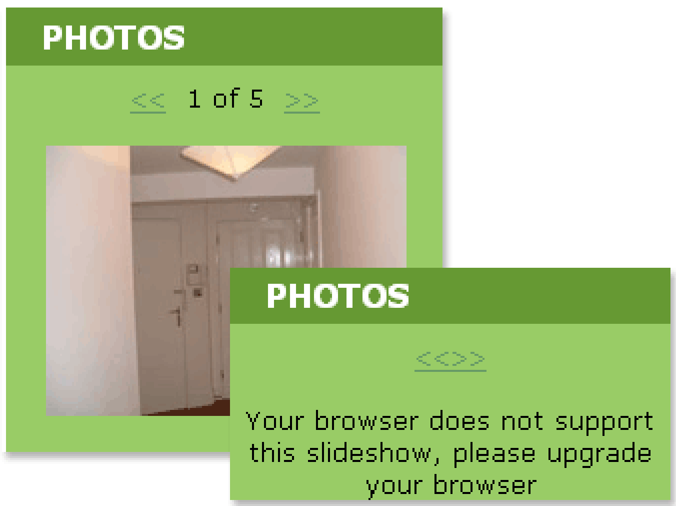
The script uses two variables – one is an array with all the image names and the other indicates which slide is currently shown:

```
photonames=new Array('flat1.jpg','flat2.jpg',
                     'flat3.jpg','flat4.jpg',
                     'flat5.jpg');
currentphoto=0;
```

The HTML part where the slide show should appear looks like this:

```
<p>
  <a href="#"
    onclick="slideshow(-1);return false">&#60;&#60;</a>
  <span id="counter"></span>
  <a href="#"
    onclick="slideshow(1);return false">&#62;&#62;</a>
</p><br />
<script type="text/javascript">
  document.write('<img src="img/'+photonames[0]+'" ##
  id="photo" alt="photo" />');
</script>
<noscript>
  <p>
     Your browser does not support this slideshow,
     please upgrade your browser
  </p>
</noscript>
```

Users without JavaScript get a message that their browser does not support the slide show and asks them to upgrade.



The script that creates the slide show receives one parameter via the `onclick` handlers in the links which indicates the direction of the show: -1 for backward and 1 for forward.

```
function slideshow(direction){
  if(currentphoto+direction<0){
    alert('This is the first picture');
    return;
  }
  if(currentphoto+direction>photonames.length-1){
    alert('This is the last picture');
    return;
  }
  currentphoto+=direction;
  updatecounter();
  document.images.photo.src='img/'+photonames[currentphoto];
}
```

The function tests if the current photo is the last or the first one and displays an `alert` if that is the case. Otherwise it adds the direction value to the variable `currentphoto` and calls the function `updatecounter()`. It then shows the appropriate image by setting the `src` attribute of to the `IMG` tag to the current array element of `photonames`.

The updatecounter() function changes the content of the `SPAN` with the id counter. We get the current picture count by adding 1 to currentphoto (as computers start counting at 0 whereas humans start counting at 1) and the number of all photos by reading out the `length` property of the `photonames` array.

```
function updatecounter(){
  counter='  '+(currentphoto+1)
  counter+=' of '+photonames.length+'  ';
  document.getElementById('counter').innerHTML=counter;
}
```

If the site maintainer wished to add more pictures in the future, all he'll have to do is to add their names to the `photonames` array. This seems fairly straight forward, however it assumes the maintainer knows about JavaScript.

DHTML issue #5: Keeping maintenance JavaScript based

This has proven over the years to be a real problem, as not all maintainers know about JavaScript and cause errors that break functionality. A lot of sites are maintained in Content Management Systems (CMSs) that allow adding content to the HTML document but not script blocks or even variables in the head of the document.

DHTML issue #6: Mixing HTML and JavaScript

Writing out the images via `document.write()` means we mix HTML and JavaScript – and thereby lose the option to maintain both independent of one another. Scripts inside the body of the page rob JavaScript of one of its great features: One script that has to be loaded once (as browsers cache JavaScript locally) gets applied to a lot of pages. It is as if you add the site's style sheet to every single page in it.

DHTML issue #7: Blaming the user

The other problem a lot of DHTML scripts had is that they expected JavaScript to be available (see issue#1) and blamed the user for not upgrading their browser or turning on JavaScript. It is not necessarily laziness of your visitors not to upgrade their browsers or keep JavaScript turned off, it might be their company's policy – banks tend to disallow JavaScript or filter it out via proxy servers. In any case, a message that the user's browser does not support your functionality and that he should upgrade is never good style. This is especially annoying if the upgrade message gets sent to very modern browsers as the testing script is outdated and reads out version numbers and browser names instead of testing for objects.

Our DHTML page is almost finished; the only thing that is missing is setting the first tab and the first image as the current ones when the page is loading.

The `initpage()` function does that by calling `updatecounter()` – which uses the variable `currentphoto` – and `showtab()` with 0 as the tab to highlight:

```
function initpage(){
  updatecounter();
  showtab(0);
}
```

We call this function inside the body tag and via the `onload` handler:

```
<body onload="initpage()">
```

DHTML issue #8: Taking over the document

DHTML scripts don't play well with other scripts. Most of them tend to consider the document their property and won't allow other scripts to deal with it. For starters, adding an onload event in each BODY means we need to maintain the HTML and the JavaScript if the function name were to change in the future.

Furthermore, if we wanted to add more scripts that need to get initiated when the page loads we'd need to add those to the `onload` attribute. The variables `photonames` and `currentphoto` are also defined globally, which means that if there is another function that sets a variable with the same name it'll overwrite them. The same applies to the functions defined in the script. Using several DHTML scripts in the same page meant most of the time re-writing a lot of the code to avoid scripts clashing.

Now that we know the problems to avoid, let's take a look how a DOM scripting solution to the same specifications would look like.

Introducing DOM scripting

The first difference between DHTML and DOM scripting is that you don't rely on JavaScript to make the page work. You start with a plain HTML page that makes sense and works without any scripting. In our case this means we replace the tabbed interface with a list of links pointing to targets in the page and links back to the list. We also make the picture link to the larger picture and show all the slides as a list instead of a slide show.

Main content:

```
<div id="flatdetails">
  <ul id="detailnav">
    <li><a href="#details">Flat Details</a></li>
    <li><a href="#location">Location</a></li>
    <li><a href="#transport">Public Transport</a></li>
    <li><a href="#viewingtimes">Viewing Times</a></li>
    <li><a href="#price">Price</a></li>
  </ul>
  <div>
    <h3><a id="details" name="details">Flat Details</a></h3>
    [... content ...]
    <p class="back"><a href="#detailnav">Back to menu</a></p>
  </div>
  <div>
    <h3><a id="location" name="location">Location</a></h3>
    [... content ...]
    <p class="back"><a href="#detailnav">Back to menu</a></p>
  </div>
[... and so on...]
</div>
```

```
Pop-Up:
<div id="flatimage">
  <p>
    <a href="img/flat.jpg">
        <img src="img/flatshot.jpg"
         alt="Photo of the flat" />
      </a>
  </p>
  <p class="caption">
    <a href="img/flat.jpg">See larger picture</a>
  </p>
</div>
Slides:
<ul id="flatshots">
  <li><img src="img/flat1.jpg" alt="Hallway" /></li>
  <li><img src="img/flat2.jpg" alt="Hob" /></li>
  <li><img src="img/flat3.jpg" alt="Bathroom" /></li>
  <li><img src="img/flat4.jpg" alt="Living Room" /></li>
  <li><img src="img/flat5.jpg" alt="Bedroom" /></li>
</ul>
```

This also ensures the accessibility of the page, as every photo has a proper alternative text and there aren't any links that don't work without JavaScript. Any interactive element on the page does what it is intended to do.

The goal of DOM scripting

Where DHTML was basically meant to "do the job" at a current point in time, DOM scripting is meant to also work in the future and be easy to maintain or change. To achieve this, DOM scripting follows certain ideas:

- Scripts should only apply themselves if they can be supported (Progressive Enhancement)
- Scripts should be easily changeable by maintainers that do not know JavaScript.
- Scripts should strictly follow the idea of separation of the different web design layers: Structure (HTML), Presentation (CSS) and Behaviour (JavaScript). This means that we should avoid embedding any scripting inside the HTML and not define any look and feel in the JavaScript but keep it to the CSS.
- Scripts should not interfere with other scripts that may be applied to the same document.

We use scripting to turn HTML into an interactive interface when JavaScript is available and the DOM is supported. We will also generate any HTML elements necessary for the functionality via the DOM, to avoid promising the user functionality that isn't there.

DOM scripting asset #1: Progressive Enhancement

This method is called *Progressive Enhancement* as we enhance the page increasingly with the support of the user agent (in most cases a visual browser, but it might also be a text-to-speech interface, a mobile phone or a plain text browser).

The opposite of *Progressive Enhancement* is *Graceful Degradation*, which means we make sure there is a fallback option for older browsers that don't support a sophisticated interface. One example of that would be keeping the slide show as it is, but offer the list of images inside a `NOSCRIPT` block for non-JavaScript visitors.
The difference is that Graceful Degradation has older user agents in mind and means double maintenance. Progressive Enhancement uses agreed web standards and tests for their availability before using them. This means that it will work in future user agents as well as the current ones – granted the user agents follow the agreed standards.

Technically this means that you check for the support of certain object in your scripts before you try to apply them. The first and most important check is to see if the W3C DOM is supported:

```
function amazeUser(){
  if(document.getElementById){
     [… your other code …]
  }
}
```

However, some browsers do support `getElementById` but fail to support other methods of the DOM, like `createTextNode`. Opera 6 is one of these. If you want to make sure that your script will not fail because of these inconsistencies, check for both:

```
function amazeUser(){
  if(document.getElementById && document.createTextNode){
     [… your other code …]
  }
}
```

Of course the most secure way is to check for each and every method you are going to use, however, as it is highly unlikely that user agents only lack support for some of them something like this might be overkill:

```
function amazeUser(){
  if(document.getElementById &&
     document.createTextNode &&
     document.createElement &&
     document.insertBefore &&
     document.appendChild){
     [… your other code …]
  }
}
```

The other issue is that you indent the whole code when you wrap the whole code block in an `if` condition. Therefore it is more pleasing to the eye and increases code legibility to simply return when the conditions are not met. For that we need to turn our condition the other way around. We return when neither `getElementById` nor `createTextNode` are supported.

```
function amazeUser(){
  if(!document.getElementById ||
     !document.createTextNode){
     return
  }
  [… your other code …]
}
```

The same applies to HTML we want to reach. It is a lot safer to test if an element exists first before changing its style collection:

```
// this will cause an error if there is no first heading
document.getElementsByTagName('h1')[0].style.color='red';

// this will never cause an error.
if(document.getElementsByTagName('h1').length>0){
  document.getElementsByTagName('h1')[0].style.color='red';
}
```

DOM scripting asset #2: Ease of maintenance

Debugging and altering DHTML scripts can be a time-consuming and frustrating experience. Imagine a script where the IDs of expected HTML elements and the look and feel are scattered all over the place.

Searching and replacing in editors doesn't always work, especially as you might cause real problems if the original developer used variable names or IDs that could be part of a command. Therefore DOM scripting tries to keep all the variables that might have to change in the future in one place – at the beginning of the script and not where they are defined for the first time.

```
// this might be hard to maintain
function amazeUser(){
  if(!document.getElementById ||
     !document.createTextNode){
     return
  }
  [… lots of other code…]
  var navigation=document.getElementById('nav');
  [… lots of other code…]
  var wrongTrousersMsg="It's the wrong trousers, Gromit!";
  [… lots of other code…]
}

// this is easier to maintain

function amazeUser(){
  if(!document.getElementById ||
     !document.createTextNode){
     return
  }

  // variables, change as applicable
  var navigation=document.getElementById('nav');
  var wrongTrousersMsg="It's the wrong trousers, Gromit!";
  // end variables

   [… lots of other code…]
}
```

This means a bit more planning and attention to detail but will make it a lot easier to change the script should it become necessary. Notice the `var` keywords, they are very important - more about that later in asset #5.
Another aspect of easing maintenance is making sure that we don't keep tasks to JavaScript that other technologies are more suitable for – like defining the look and feel.

DOM scripting asset #3: Separation of Presentation and Behaviour

The style collection allows you to totally change the look and feel of any element in the document; however it also means that a future maintainer needs to change the script to accommodate for a new look and feel.
This is exactly the same problem we had with old school HTML full of layout tables, `font` elements and `bgcolor` attributes. Changing the look of the site was a painful long search and replace process, which is why CSS was invented and got browser support.
By keeping all the look and feel in the CSS we can allow for skinning of a web application or web site.
We can avoid changing the style collection by dynamically applying classes via the `className` property.

```
// using the style collection
if(document.getElementsByTagName('h1').length>0){
  var h=document.getElementsByTagName('h1')[0];
}
if(h){
  h.style.color='red';
  h.style.fontFamily='Arial';
  h.style.fontWeight='bold';
  h.style.borderStyle='solid';
  h.style.borderColor='#c00';
  h.style.borderWidth='1px';
}

// using className
var errorClass='error';
if(document.getElementsByTagName('h1').length>0){
  var h=document.getElementsByTagName('h1')[0];
  h.className=errorClass;
}
```

```
// In the Style Sheet:
h1.error{
  font-family:Arial,Sans-serif;
  color:red;
  border:1px solid #c00;
}
```

There is one problem though: HTML elements can have more than one class. A construct like `<p class="kids withproduct highlight">`is valid HTML and sometimes not a bad idea at all. Therefore we need to test if the element already has a class before applying the new one and add a space before the new one if that is the case.

```
if(h){
  h.className+=h.className?' '+errorClass:errorClass;
}
```

This is called the ternary notation and is a shorter way of writing:

```
if(h){
  if(h.className){
    h.className=h.className+' '+errorClass;
  } else {
    h.className=errorClass;
  }
}
```

There are a few situations where we cannot separate presentation and behaviour completely, but these are seldom. One example would be a drag and drop interface. You will have to set the position of the element you drag dynamically via JavaScript. For normal day to day tasks it is much safer and cleaner to apply and remove CSS classes and keep the presentation in the hands of the CSS designer though.

DOM scripting asset #4: Separation of Structure and Behaviour

DOM scripting means trying to separate JavaScript from all the other development streams. When you use `document.write()` and inline event handlers like `onclick`, `onload` or onmouseover you mix HTML and JavaScript and make it harder to debug or maintain the site.
Instead of document.write() you use some other DOM methods than the aforementioned `getElementById` and `getElementsByTagName`.
You will now get a list of all the options available, don't worry if this is a bit overwhelming at first, we will later on recreate the HTML example page in DOM scripting and you'll see examples how to use them.

If you reached the part of the page you wanted to reach via `getElementById` or `getElementsByTagName` you can move around the document via the following methods and properties:

- `node.previousSibling` - retrieves the previous sibling node and stores it as an object
- `node.nextSibling` - retrieves the next sibling node and stores it as an object
- `node.childNodes` - retrieves all child nodes of the object and stores them in an list. There are shortcuts for the first and last child node, named `node.firstChild` and `node.lastChild`.
- `node.parentNode` - retrieves the node containing node.

- You can read and write attributes of each node and you can test their types and names:

- `node.getAttribute('attribute')` - retrieves the attribute with the name attribute
- `node.setAttribute('attribute')` - sets the attribute with the name attribute
- `node.nodeType` - reads the type of the node (1 = element, 3 = text node)
- `node.nodeName` - reads the name of the node (either element name or #textNode)
- `node.nodeValue` - reads or sets the value of the node (the text content in the case of text nodes)

If there are elements that don't make any sense without JavaScript – for example the previous and next links in the slide show – you can create them on the fly and insert them or replace others with them:

- `document.createElement('element')` - creates a new element with the name element
- `document.createTextNode('string')` - creates a new text node with the node value of string
- `node.appendChild(newNode)` - adds `newNode` as a new child node to node.
- `newNode=node.cloneNode(bool)` - creates `newNode` as a copy (clone) of node. If `bool` is true the clone includes clones of all the child nodes of the original.
- `node.insertBefore(newNode,oldNode)` - inserts `newNode` as a new child node of node before `oldNode`
- `node.removeChild(oldNode)` - removes the child `oldNode` from node
- `node.replaceChild(newNode, oldNode)` - replaces the child node `oldNode` of node with `newNode`

Using these methods and properties allows us to separate JavaScript and HTML. Keeping all the JavaScript in separate files from the HTML means you can apply your scripts to any number of pages in the site without having to change the HTML.
All it needs is a script element in the head of the documents and it'll apply your functionality:

```
<script type="text/javascript" src="scripts.js"></script>
```

Notice that `language="javascript"` is deprecated in modern HTML and using the mime type `text/javascript` in conjunction with the `type` attribute is the correct way.

DOM scripting asset #5: Using modern event handling

When we keep all our JavaScript in separate files we need to find a way how to make the HTML document call your functions where they are needed without inline event handlers like `onload`, `onclick` or `onmouseover`. Keeping them inside the HTML would still mean we need to change all HTML documents that use the function if, for example, it needed renaming.
The trick is to add the handler directly to the element you want, via the *onevent* handler. For example when you want to run the function popup when the first link inside the element with the ID `productshot` is clicked, you can use:

```
var productShot=document.getElementById('productshot');
if(!productShot){return;}
if(!productShot.getElementsByTagName('a').length>0){
  var a=productShot.getElementsByTagName('a')[0];
  a.onclick=popup;
}
```

If you wanted to trigger a function called `init` when the browser has finished loading the document, you can run it as an `onload` handler of the window object:

```
window.onload=init;
```

However, this means you can only start one function when the window loads. The workaround is to add an anonymous function that calls several functions:

```
window.onload=function(){
  init();
  otherFunction();
}
```

This is a step in the right direction, but what if you have several JavaScript includes?

In DOM scripting we stop assuming that our script is the only one that will ever be applied to the document, but assume instead that it is part of a whole group of scripts allof which fulfilling different tasks. Therefore it is important that we stop using event handlers that exclusively allow our script to run when the page has loaded or an element has been activated.

One step in the right direction was Simon Willisons' addLoadEvent function:

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}
```

If we use this function instead of a simple `onload` event handler, we can add the `init` function without overriding other `onload` calls of other scripts, as the `addLoadEvent` function checks for other functions already assigned via an handler and extends the initial handler call.

This solves the onload issue, but what about handlers on elements themselves? What if another script tries to call another function when the popup link gets clicked?

Scott Andrew LePera solved this issue on his blog as early as 2001 with the addEvent function:

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

The workings of this function and events in general can fill whole books at the moment, and there is an ongoing discussion how `addEvent` can be improved - for example to support retaining the option to send the current element as a parameter via `this` - and many clever solutions were developed so far. As each have different drawbacks we won't go into details here, but if you are interested, check the comments at the addEvent recoding contest page at quirksmode.org. Suffice to say that if you want to add the popup function to the link using this helper function you do the following:

```
var productShot=document.getElementById('productshot');
if(!productShot){return;}
if(!productShot.getElementsByTagName('a').length>0){
  var a=productShot.getElementsByTagName('a')[0];
  addEvent(a,'click',popup,false);
}
```

You can also use this function to provide the init call when the window has finished loading the document and all its assets:

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

This will trigger the `popup` function and not overwrite any other script calls in modern browsers. If you wanted to know which element was activated and stop the default action of the element (like stopping a link from being followed) you need more helper functions to do this for you, namely `getTarget` and `cancelClick`. You'll learn more about them in the DOM scripting example of the demo page.

Changing the event handling to this admittedly more complex but also more versatile way is one step to keep different scripts out of each other's hair. One other step is to avoid name clashing of functions and variables.

DOM scripting asset #6: Avoiding clashes with other scripts

In DOM scripting we assume we know nothing about other scripts that may be attached to the same document. Therefore it is important that we keep all variables contained in the current script. The demo DHTML page has a global array of images called `photonames` and an indicator which photo is shown called `currentphoto`. What if an other script uses the same variable names for a different purpose? The last script to get applied will overwrite the settings of the other one and most likely break it.

The same applies to functions with the same name. A lot of scripts use a function called `init` to initialize all the settings, add event handlers and check if all the necessary elements are available.

The first safety measure is to stop using global variables. Global variables are those that get defined outside of functions or without the `var` keyword inside functions. For example:

```
averageWeight=34;
function classOfNinetyFour(){
  averageWeight=50;
}
alert(averageWeight);
classOfNinetyFour();
alert(averageWeight);
```

This will result in two alerts, one stating 34, and the other stating 50. If you don't want the function to change the initial value you need to use the `var` keyword:

```
averageWeight=34;
function classOfNinetyFour(){
  var averageWeight=50;
}
alert(averageWeight);
classOfNinetyFour();
alert(averageWeight);
```

This will result in two alerts both stating 34.

You can work around the issue of function name clashing by adding a prefix or suffix that is unique to your script.

```
// generic names are very likely to be used by other scripts
var currentSection=0;
var linkName='validate';
function init(){
  [… code …]
}
function navigate(section){
  [… code …]
}
function validate(f){
  [… code …]
}
```

```
// adding a prefix makes that less likely
var coolnav_currentSection=0;
var coolnav_linkName='validate';
function coolnav_init(){
  [… code …]
}
function coolnav_navigate(section){
  [… code …]
}
function coolnav_validate(f){
  [… code …]
}
```

However, this means a lot more typing and still does not really contain all your functions and variables in one entity outside the reach of other scripts. There is a way to do that though and that is creating an object that contains all the variables as properties and all the functions as methods:

```
var coolnav=new Object();
coolnav.currentSection=0;
coolnav.linkName='validate';
coolnav.init=function(){…}
coolnav.navigate=function(section){…}
coolnav.validate=function(f){…}
```

You can work around repeating the name `coolnav` by using a different notation, called the object literal. In this notation the object surrounds all the methods and properties with curly braces and instead of the equal sign you assign values to names via a colon. You separate different properties and methods via commas:

```
var coolnav={
  currentSection:0,
  linkName:'validate',
  init:function(){…},
  navigate:function(section){…},
  validate:function(f){…}
}
```

Notice that the last entry must not have a comma!

Using the object literal means you can only overwrite the functions of another script by error if you choose the same object name, in this case coolnav. This is a lot less likely than overwriting a function called init.

Re-creating the demo page with DOM scripting

We will now take the HTML page and add the dynamic functionality with all these ideas in mind. As explained earlier, we changed the HTML to function without any JavaScript. You can test this by opening the [plain HTML page] in a browser. Next we plan for all the variables and functions we will need and add them inside an overall object. To keep things short, we call the object `fp` (for "flat page"):

```
fp={
// CSS classes
  dynamicClass:'fp',
  showClass:'show',
  currentClass:'current',
  hideClass:'hide',
// Page section IDs
  boundaryId:'boundary',
  detailsContainer:'flatdetails',
  detailsNav:'detailnav',
  popImageContainer:'flatimage',
  slideShowContainer:'flatshots',
  slides:'photos',
// ID of generated popunder
  popunderId:'popunder',
// slide show links
  slideBack:'&raquo;',
  slideFwd:'&laquo;',
// close popup label
  closeLabel: 'close',
// global parameters
  currentLink:null,
  currentSection:null,
  photoCount:null,
  curPhoto:null,

/* Initialise functionality */
  init:function(){...},
/* tabbed navigation */
  initTabs:function(){...},
  showTab:function(e){...},
/* pop-under functionality */
```

```
  initWin:function(){...},
  openWin:function(e){...},
  closeWin:function(e){...},
/* Photo slideshow */
  initPhotos:function(){...},
  showPhoto:function(e){...},
/* helper methods */
  getTarget:function(e){...},
  cancelClick:function(e){...},
  addEvent:function(elm, evType, fn, useCapture){...},
  cssjs:function(a,o,c1,c2){...}
}
// start the show.
fp.addEvent(window, 'load', fp.init, false);
```

You can see the outcome of our efforts when you open the dynamically enhanced page in a browser.
You start by defining all the variables that may have to change in the future and keep them at the beginning of the script. Some comments make it easier for the maintainer to know what is what:

```
// CSS classes
  dynamicClass:'fp',
  showClass:'show',
  currentClass:'current',
  hideClass:'hide',
// Page section IDs
  boundaryId:'boundary',
  detailsContainer:'flatdetails',
  detailsNav:'detailnav',
  popImageContainer:'flatimage',
  slideShowContainer:'flatshots',
  slides:'photos',
// ID of generated popunder
  popunderId:'popunder',
// slide show links
  slideBack:'&raquo;',
  slideFwd:'&laquo;',
// close popup label
  closeLabel: 'close',
```

```
// global properties
  currentLink:null,
  currentSection:null,
  photoCount:null,
  curPhoto:null,
```

The class names are the classes we will add or remove dynamically to achieve different looks or show and hide parts of the page.

The IDs are needed for us to reach what we want to change or assign event handlers to. We will generate links to navigate through the slides and keep their text content in two properties called `slideBack` and `slideFwd`. We will also generate a link to close the popup and store its content in the `closeLabel` property.

The property section of our script ends with four properties that will store later which link has to be highlighted in the tab-navigation, which section should be shown, how many slides there are and which is the currently shown slide.
The script commences with the init method:

```
  init:function(){
    if(!document.getElementById || !document.createTextNode){return;}
    var container=document.getElementById(fp.boundaryId);
    if(!container){return;}
    fp.cssjs('add',container,fp.dynamicClass);
    fp.initTabs();
    fp.initWin();
    fp.initPhotos();
  },
```

We check if the DOM is supported and there is an element with the `boundaryId` in the document. If there is the necessary element we use a method called `cssjs` to add a dynamic class to this element.
This enables the CSS designer to create a completely different look and feel for the dynamic and the static version of the page, simply by using the descendant selector with or without the dynamic class:

```
/* style for non-JS */
#boundary{
  [… CSS Settings …]
}
/* style for JS */
#boundary.fp {
  [… CSS Settings …]
}
```

Using this simple trick we can also hide all the elements via CSS that we want to show later on via scripting:

```
#boundary.fp #flatdetails div,
#boundary.fp #photos li
{
  position:absolute;
  top:-999px;
  clear:both;
}
```

The methods we create later will add a dynamic class called “show” to the element we want to show, which is why we overrule the hiding when this class is applied:

```
#boundary.fp #flatdetails div.show,
#boundary.fp #photos li.show{
  position:relative;
  top:0;
}
```

The cssjs method is a helper method that adds, removes or swaps classes dynamically. You can find out more about it on its homepage.
We conclude the init method by calling the respective initialization methods of the three dynamic parts of the page – the tabs, the popup and the slide show.

The tab navigation in DOM scripting

If you remember, the tab navigation is now a list of links pointing to anchors in the page. What the `initTabs` method does is overriding this functionality and adding event handlers to these links pointing to the `showTab` method.
We check if the navigation element exists and that it has at least one link.

```
  initTabs:function(){
    var nav=document.getElementById(fp.detailsNav);
    if(nav && nav.getElementsByTagName('a')[0]){
```

We then retrieve all the links and store them in the array links.

```
      var links=nav.getElementsByTagName('a');
```

We use a regular expression to read the name of the anchor the link points to. We then test if the element with this ID exists.

```
      var firstSection=links[0].href.toString().match(/#(.*)/)[1];
      firstSection=document.getElementById(firstSection);
      if(firstSection){
```

If it does exist we read store the `parentNode` of its `parentNode` in the variable `parentDIV` and apply the class to show the section to it.

```
        var parentDiv=firstSection.parentNode.parentNode;
        fp.cssjs('add',parentDiv,fp.showClass);
```

This is the element we want – the `DIV` – as the elements containing the target id are the named anchors inside the headings:

```
  <div>
    <h3><a id="details" name="details">Flat Details</a></h3>
    [... content ...]
    <p class="back"><a href="#detailnav">Back to menu</a></p>
  </div>
```

We then add the `current` class to the first link and store both the link and the currently visible section in properties of the main object.

```
        fp.cssjs('add',links[0],fp.currentClass);
        fp.currentSection=firstSection;
        fp.currentLink=links[0];
```

Lastly, we loop through all the links and point them towards the `showTab` method via `addEvent`. Notice that although both methods are contained in the same object we need to use `fp.showTab` and not simply `showTab`.

```
        for(var i=0;i<links.length;i++){
          fp.addEvent(links[i],'click',fp.showTab,false);
        }
      }
    }
  },
```

This is the initialization of the tabular navigation done; now we need a function to show the current section and highlight the current tab once it was clicked. We already did half the work by pointing all the links to this method.
Now we need the method to know which link was activated and stop it from following the link. For this we'll use two more helper methods, namely `getTarget` and `cancelClick`. Both of these are rather complex and if you wanted to understand every detail of what they do you'd need to get accustomed to the event handling model of the W3C. This would fill another few pages, but in a nutshell what it does is this:
Every event handler assigned to a node via the methods in `addEvent` (there are some W3C compliant ones, and fallback options for non-compliant browsers like MSIE or older browsers) sends an object to the event listener function (in this case `showTab`).

This object can be retrieved by means of a parameter called `e` and has a lot of very handy properties. One is `target` which is the element that has got the event handler attached to it.
In a W3C compliant browser this target could be read out via `e.target`, however as not all browsers are compliant we use the `getTarget` method that contains several browser specific ways of retrieving this element.
Once activated, there are two things that happen to every element:

- Firstly, it triggers something called event bubbling – which means that the same event (f.e. 'click') that starts the event listener method for this element will start all the ones assigned to parent elements of the current one.
- Secondly there is a default action of every element– like a link pointing to an anchor sending the browser there.
- As we want neither to happen we use a helper method called `cancelClick` that prevents both - once again in a cross browser fashion.

Using these helper methods we can bring our tabs to life. Firstly we use getTarget to check which tab was activated and read the ID the tab points to.

```
showTab:function(e){
  var section=fp.getTarget(e);
  var toshow=section.getAttribute('href').    ###
      toString().match(/#(.*)/)[1];
```

We then check if there is already a defined current section and current link and hide or un-highlight these by removing the show and current classes.

```
    if(fp.currentSection && fp.currentLink){
      fp.cssjs('remove',fp.currentSection.parentNode. ###
                parentNode,fp.showClass);
      fp.cssjs('remove',fp.currentLink,fp.currentClass);
    }
```

If the target the link points to really exists we add the show and the current class and re-set the properties of the main object to point to the new section and link.

```
    if(document.getElementById(toshow)){
      toshow=document.getElementById(toshow);
      fp.cssjs('add',toshow.parentNode.parentNode,fp.showClass);
      fp.cssjs('add',section,fp.currentClass);
      fp.currentSection=toshow;
      fp.currentLink=section;
    }
```

Finally we stop the browser from following the link.

```
    fp.cancelClick(e);
  },
```

The popup window in DOM scripting

In the DHTML version we used a popup window to show the image and realized that popups are not a safe way to show anything these days any longer. This is why we will replace the popup with something called a pop-under or layer ad. In essence this is a newly generated DIV element that gets absolutely positioned to cover the main content of the page.
We start by testing if the image container exists.

```
initWin:function(){
  var container=document.getElementById(fp.popImageContainer);
  if(!container){return;}
```

We then commence by creating a new DIV element, giving it the ID defined in the properties of our main object and hide it by applying the hiding class. We then add the newly created DIV as a child to the main element of the page.

```
  fp.popunder=document.createElement('div');
  fp.popunder.id=fp.popunderId;
  fp.cssjs('add',fp.popunder,fp.hideClass);
  document.getElementById(fp.boundaryId).appendChild(fp.popunder);
```

Next we need a link to hide the large picture once it is shown. We create a new link, apply the closing label defined in the properties and assign an even handler pointing to the `closeWin` method. We need to set the link's `href` attribute to something, as otherwise the link will not display as a link. We then append the link to the newly created DIV.

```
  var closeLink=document.createElement('a');
  closeLink.setAttribute('href','#');
  closeLink.innerHTML=fp.closeLabel;
  fp.addEvent(closeLink,'click',fp.closeWin,false);
  fp.popunder.appendChild(closeLink);
```

To conclude the init method we loop through all of the links inside the element with the photo preview and add an event handler that points to the `openWin` method.

```
  var links=container.getElementsByTagName('a');
  for(var i=0;i<links.length;i++){
    fp.addEvent(links[i],'click',fp.openWin,false);
  }
},
```

The `openWin` method checks if there is already an image inside the `popunder` DIV and creates a new one if that is not the case. The source of the picture is the URL the link the user clicked on points to, which is why we retrieve this one as the event target via `getTarget` and read its `href` attribute.

```
openWin:function(e){
  // if there is no image in popunder yet
  if(!fp.popunder.getElementsByTagName('img')[0]){
    // create image and grab link location as src
    var shot=document.createElement('img');
    var bigpic=fp.getTarget(e);
    shot.setAttribute('src',bigpic.href);
```

We add the new image to the `popunder` DIV and add an event handler pointing to the `closeWin` method when the user clicks the image.

```
    fp.popunder.appendChild(shot);
    // apply close functionality to image
    fp.addEvent(shot,'click',fp.closeWin,false);
  }
```

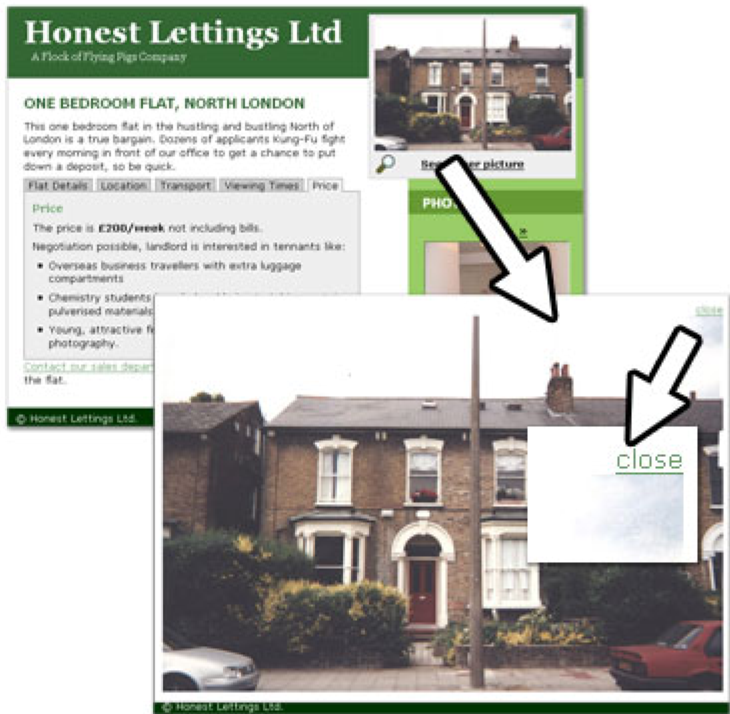
We then show the image by removing the hiding class and stop the link from being followed by calling the `cancelClick` method

```
  fp.cssjs('remove',fp.popunder, fp.hideClass);
  fp.cancelClick(e);
},
```

The `closeWin` method simply adds the hide class to the `popunder` DIV to hide it and calls `cancelClick` to stop the browser following the link.

```
closeWin:function(e){
  // hide popunder div and don't follow link
  fp.cssjs('add',fp.popunder, fp.hideClass);
  fp.cancelClick(e);
},
```

If the visitor now clicks on the photo of the house or the “see larger picture” link he’ll see the large picture covering the whole page. If he clicks the image or the close link the image will be hidden again.



The slide show in DOM scripting

The slide show has changed from a global array and a script block to a simple HTML list containing images:

```
<ul id="flatshots">
  <li><img src="img/flat1.jpg" alt="Hallway" /></li>
  <li><img src="img/flat2.jpg" alt="Hob" /></li>
  <li><img src="img/flat3.jpg" alt="Bathroom" /></li>
  <li><img src="img/flat4.jpg" alt="Living Room" /></li>
  <li><img src="img/flat5.jpg" alt="Bedroom" /></li>
</ul>
```

What the script now has to do is create the last and next links, the counter telling the user which slide of how many is shown currently and to hide all but the current slide. Notice that the hiding of the slides was already done in the CSS using the dynamic “fp” class. This saves us from having to loop through all the elements to hide them.

Once again, we start by testing if the necessary HTML elements are available, read out the amount of list items and store the initial values in the appropriate properties of the main object:

```
/* Photo slideshow */
  initPhotos:function(){
    var s=document.getElementById(fp.slideShowContainer);
    var p=document.getElementById(fp.slides);
    if(!s || !p){return;}
    // get all li elements and store the length
    // and current photo properties
    fp.photos=s.getElementsByTagName('li');
    fp.photoCount=fp.photos.length;
    fp.curPhoto=0;
```

We then create a new paragraph element to host the generated “back” and “next” links.

```
    // create new paragraph
    var newp=document.createElement('p');
```

We create the links like we did in the popunder example.

```
    // create next link
    fp.nextLink=document.createElement('a');
    fp.nextLink.innerHTML=fp.slideBack;
    fp.nextLink.setAttribute('href','#');
    // create previous link
    fp.prevLink=document.createElement('a');
    fp.prevLink.setAttribute('href','#');
    fp.prevLink.innerHTML=fp.slideFwd;
```

We then create a new SPAN to host the indicator which picture of how many is shown and add the information as a new text node.

```
    // create counter display x of y
    fp.photoCountDisp=document.createElement('span');
    fp.photoCountDisp.appendChild(document.createTextNode(  ###
    (fp.curPhoto+1)+' of '+fp.photoCount));
```

We add all of these elements to the generated paragraph and insert it before the list.

```
    // append all to new paragraph
    newp.appendChild(fp.prevLink);
    newp.appendChild(fp.photoCountDisp);
    newp.appendChild(fp.nextLink);
```

Last but not least, we add the event handlers pointing to the `showPhoto` method and show the first photo by calling the same method.

```
    // set handlers
    fp.addEvent(fp.prevLink,'click',fp.showPhoto,false);
    fp.addEvent(fp.nextLink,'click',fp.showPhoto,false);
    // show first photo
    fp.showPhoto(fp.curPhoto)
  },
```

The `showPhoto` method needs to be a bit tricky, as it needs to know when to offer the next and previous links and when to hide them. It starts by retrieving the target and hiding the picture that was shown before.

```
  showPhoto:function(e){
    var photo = fp.getTarget(e);
    // hide old photo
    fp.cssjs('remove',fp.photos[fp.curPhoto],fp.showClass);
```

We then need to determine if we need to show the next or the previous photo by comparing the target to the appropriate links. As the method is also called from the `initPhotos` method we also need to check if the target is neither. We change the `curPhoto` counter accordingly (either adding -1 if the previous link was clicked, one when the next link was clicked or 0 when there was no link clicked at all.

```
    // increase, decrease or keep counter value
    var add=(photo==fp.prevLink)?-1:+1;
    if(photo!=fp.prevLink && photo!=fp.nextLink){add=0;}
    fp.curPhoto=fp.curPhoto+add;
```

We then show the image by applying the `show` class. We hide the previous link when the photo is the first one and the next link when the photo is the last one.

```
    // show new image
    fp.cssjs('add',fp.photos[fp.curPhoto],fp.showClass);
    // show or hide previous or next link
    fp.cssjs(fp.curPhoto>0?'remove':'add',fp.prevLink,fp.hideClass);
    fp.cssjs(fp.curPhoto<(fp.photoCount-
1)?'remove':'add',fp.nextLink,fp.hideClass);
```

We change the counter display by replacing the first number in the node’s value via a regular expression and stop the browser from following the link that was clicked on.

```
    // increase or decrease counter display
    var disp=fp.photoCountDisp.firstChild;
    disp.nodeValue=disp.nodeValue.replace(/^\d/,(fp.curPhoto+1));
```

```
    fp.cancelClick(e);
  },
```

That's it. We created the same functionality the DHTML version had in a fashion that is less obtrusive, easier to maintain and works nicely in modern browsers and allows even non-JavaScript users to get all the content without any elements that don't do anything although they promise it.

Fixing for Safari

Almost, that is. There is one very modern browser that is a bit of a nightmare to support and that is Safari. Safari claims to understand the W3C event model, but does not implement it properly. Therefore we need to hack our script to make it work in Safari, too. The code example in the demo page does that already.

Safari will not stop the default action when we call `cancelClick`. Preventing links from being followed in Safari is only possible by applying an `onclick` handler that returns false. For example in the close link of the `popunder` we must do the following:

```
fp.addEvent(closeLink,'click',fp.closeWin,false);
closeLink.onclick=function(){return false;}
```

In the demo script this hack is applied in an own method called `fp.fixSafari`. It is a dirty trick but to date there is no fix.

Safari does not return the link element as the event target when it gets activated but instead it returns the text node contained in the link. Therefore we need to check in `getTarget` if the `nodeName` of the target is A and take the parent node if that isn't the case:

```
if (target.nodeName.toLowerCase() != 'a'){
        target = target.parentNode;
}
return target;
```

This renders `getTarget` a bit less useful as it should be, as we need to extend the allowed types of elements if we wanted to retrieve other elements than links via `getTarget`.

Where to now?

This was a lot to take in and you may want to check the source of the demo code for comments and fiddle around with it to see if you understand what can be done with these methods of scripting.
DOM scripting is going places at the moment and more and more updates of browsers implement the W3C standards, so that hopefully in a year's time we'll be able to get rid of some of the browser-specific hacks and amendments that are still needed. In any case, making it easier for maintainers, allowing for styling via CSS and allowing other scripts to use the same page without the need to fix our script is never a bad idea, right?