CHAPTER 3

From DHTML to DOM Scripting

In this chapter, you'll learn what DHTML was, why it is regarded as a bad way to go nowadays, and what modern techniques and ideas should be used instead. You'll learn what functions are, and how to use them. You'll also hear about variable and function scope and some state-of-the-art best practices that'll teach your scripts how to play extremely well with others.

If you are interested in JavaScript and you have searched the Web for scripts, you surely have come upon the term **DHTML**. DHTML was one of the big buzz words of the IT and web development industry in the late 1990s and beginning of the millennium. You may have seen a lot of tutorials about achieving a certain visual effect on now outdated browsers rather than explaining why this effect makes sense and what the script does. And that is exactly what DHTML was about.

Note DHTML, or Dynamic HTML, was never a real technology or World Wide Web Consortium (W3C) standard, merely a term invented by marketing and advertising agencies.

DHTML is JavaScript interacting with Cascading Style Sheets (CSS) and web documents (written in HTML) to create seemingly dynamic pages. Parts of the page had to fly around and zoom out and in, every element on the page needed to react when the visitor passed over it with the mouse cursor, and we invented a new way of web navigation every week.

While all this was great fun and a technological challenge, it did not help visitors much. The "wow" effect lost its impact rather quickly, especially when their browsers were unable to support it and they ended up on a page that was a dead end for them.

As DHTML sounded like a great term, it did bring out a kind of elitism: JavaScript developers who were "in the know" did not bother much with keeping the code maintainable, as anybody who didn't understand the art of making things move was not worth changing their code for in any case. For freelance developers, it also meant a steady income, as every change had to be done by them.

When the big money stopped coming in, a lot of this code got thrown out on the Web for other developers to use, either as large JavaScript DHTML libraries or as small scripts on script collection sites. Nobody bothered to update the code, which means that it is unlikely that these resources are a viable option to use in a modern professional environment.

Common DHTML scripts have several issues:

- **JavaScript dependence and lack of graceful degradation**: Visitors with JavaScript turned off (either by choice or because of their company security settings) will not get the functionality, but elements that don't do anything when they activate them or even pages that cannot be navigated at all.
- **Browser and version dependence**: A common way to test whether the script can be executed was to read out the browser name in the navigator object. As a lot of these scripts were created when Netscape 4 and Internet Explorer 5 were state-of-the-art, they fail to support newer browsers—the reason being browser detection that doesn't take newer versions into account and just tests for versions 4 or 5.
- **Code forking**: As different browsers supported different DOMs, a lot of code needed to be duplicated and several browser quirks avoided. This also made it difficult to write modular code.
- **High maintenance**: As most of the look and feel of the script was kept in the script, any change meant you needed to know at least basic JavaScript. As JavaScript was developed for several different browsers, you needed to apply the change in all of the different scripts targeted to each browser.
- **Markup dependence**: Instead of generating or accessing HTML via the DOM, a lot of scripts wrote out content via the `document.write` directive and added to each document body instead of keeping everything in a separate—cached—document.

All of these stand in a stark contrast to the requirements we currently have to fulfill:

- Code should be cheap to maintain and possible to reuse in several projects.
- Legal requirements like the Digital Discrimination Act (DDA) in the UK and Section 508 in the US strongly advise against or in some cases even forbid web products to be dependent on scripting.
- More browsers, user agents (UAs) on devices such as mobile phones, or assistive technology helping disabled users to take part in the Web make it impossible to keep our scripts dependent on browser identification.
- Newer marketing strategies make it a requirement to change the look and feel of a web site or a web application quickly and without high cost—possibly even by a content management system.

There is a clear need to rethink the way we approach JavaScript as a web technology, if we still want to use and sell it to clients and keep up with the challenge of the changing market.

The first step is to make JavaScript less of a show-stopper by making it a "nice to have" item rather than a requirement. No more empty pages or links that don't do anything when JavaScript is not available. The term **unobtrusive JavaScript** was christened by Stuart Langridge at `http://www.kryogenix.org`, and if you enter it in Google, you'll end up on an older self-training course on the subject by me.

Unobtrusive JavaScript refers to a script that does not force itself on users or stand in their way. It tests whether it can be applied and does so if it is possible. Unobtrusive JavaScript is like

a stagehand—doing what she is good at backstage for the good of the whole production rather than being a diva who takes the whole stage for herself and shouts at the orchestra and her colleagues every time something goes wrong or is not to her liking.

Later on, the term **DOM scripting** got introduced, and in the aftermath of the @media conference in London 2004, the WaSP DOM Scripting Task Force was formed. The task force consists of many coders, bloggers, and designers who want to see JavaScript used in a more mature and user-centered manner—you can check out what it has to say at `http://domscripting.webstandards.org`.

As JavaScript did not have a fixed place in common web development methodologies—instead being considered as either "something you can download from the web and change" or "something that will be generated by the editing tool if it is needed"—the term "behavior layer" came up in various web publications.

JavaScript As "the Behavior Layer"

Web development can be thought of as being made up of several different "layers," as shown in Figure 3-1.

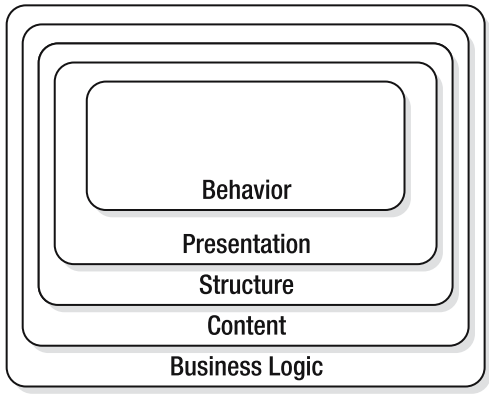

Figure 3-1. *The different layers of web development*

- **The behavior layer**: Is executed on the client and defines how different elements behave when the user interacts with them (JavaScript or ActionScript for Flash sites).
- **The presentation layer**: Is displayed on the client and is the look of the web page (CSS, imagery).
- **The structure layer**: Is converted or displayed by the user agent. This is the markup defining what a certain text or media is (XHTML).
- **The content layer**: Is stored on the server and consists of all the text, images, and multimedia content that are used on the site (XML, database, media assets).
- **The business logic layer (or back end)**: Runs on the server and determines what is done with incoming data and what gets returned to the user.

Notice that this simply defines what layers are available, not how they interact. For example, something needs to convert content to structure (such as XSLT), and something needs to connect the upper four layers with the business logic.

If you manage to keep all these layers separate, yet talking to each other, you will have succeeded in developing an accessible and easy-to-maintain web site. In the real development and business world, this is hardly the case. However, the more you make this your goal, the fewer annoying changes you'll have to face at a later stage. Cascading Style Sheets are powerful because they allow you to define the look and feel of numerous web documents in a single file that will be cached by the user agent. JavaScript can act in the same fashion by using the `src` attribute of the `script` tag and a separate `.js` file.

In earlier chapters of this book, we embedded JavaScript directly in HTML documents (and you may remember that this is tough to do for XHTML documents). This we will not do from this point on; instead, we'll create separate JavaScript files and link to them in the head of the document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html;➥
          charset=iso-8859-1">
    <title>Demo</title>
    <style type="text/css">@import 'styles.css';</style>
    <script type="text/javascript" src="scripts.js"></script>
    <script type="text/javascript" src="morescripts.js"></script>
  </head>
  <body>
  </body>
</html>
```

We should also try not to use any script blocks inside the document any longer, mainly because that would mix the structure and the behavior layers and can cause a user agent to stop showing the page should a JavaScript error occur. It is also a maintenance nightmare—adding all JavaScript to separate `.js` files means we can maintain the scripts for a whole site in one place rather than searching through all the documents.

■ **Note** While Firefox, Safari, and Opera display `.js` files as text, Microsoft Internet Explorer tries to execute them. This means that you cannot double-click or point your browser to a JavaScript file to debug it in MSIE—something that is pretty annoying when debugging code. This instant code execution is a security problem, which is why Windows XP2 does flag up any JavaScript content in local files as a security issue as shown in Figure 3-2. Do not get fooled by this—it is your code, and, unless you are a malicious coder, it is not a security issue.

Figure 3-2. *Microsoft Internet Explorer on Windows XP2 shows a warning message when you try to execute JavaScript locally.*

This separation of JavaScript into its own file makes it simpler to develop a web site that still works when the script is not available; and if a change in the site's behavior is needed, it is easy to change only the script files.

This is one of the bad habits of DHTML squashed: HTML can exist without JavaScript, and you do not need to scan a lot of documents to find out where a bug occurred. The next bad habit of DHTML was browser dependence, which we'll now replace with object detection.

Object Detection vs. Browser Dependence

One way to determine which browser is in use is by testing the `navigator` object, which reveals the name and the version of the browser in its `appName` and `appVersion` attributes.

For example, the following script gets the browser name and version and writes it out to the document:

```
<script type=" text/javascript">
  document.write("You are running " + navigator.appName);
  document.write(" and its version is " + navigator.appVersion);
</script>
```

On my computer, inside Macromedia HomeSite's preview mode, this script reports the following (as HomeSite uses the MSIE engine for previewing HTML).

```
You are running Microsoft Internet Explorer
and its version is 4.0 (compatible; MSIE 6.0;
Windows NT 5.1; SV1; .NET CLR 1.1.4322)
```

If I run the same script in Firefox 1.07 on the same computer, I get

```
You are running Netscape and its version is 5.0 (Windows; en-GB)
```

A lot of older scripts use this information to determine whether a browser is capable of supporting their functionality.

```
<script type="text/javascript">
 if(browserName.indexOf('Internet Explorer')!=-1
  && browserVersion.indexOf('6')!=-1)
  {
    document.write('<p>This is MSIE 6!</p>');
  }
  else
  {
    document.write('<p>This isn\'t MSIE 6!</p>');
  }
</script>
```

This appears rather clever at first glance, but as the output of Firefox shows, it is not a bullet-proof method of finding out which browser is in use. Some browsers like Opera won't even reveal themselves to scripts, but appear as Microsoft Internet Explorer instead.

■ **Tip** Opera is by default set up to tell scripts that it is MSIE. The reason for this is that Opera Software didn't want its browser to be blocked by web sites that were developed for MSIE only. Sadly enough, a lot of development went that route and Opera Software just didn't want to lose customers or answer a lot of angry e-mails why its "bad" browser doesn't work with web site XYZ. However, it also means that Opera doesn't show up in browser statistics of web sites and remains unimportant for those who just see the visitor numbers and what browser they use (a lot of statistics software uses the `navigator` object). If you are an Opera user and you want to turn off this preset, press F12 and choose "Identify as Opera" instead of "Identify as Internet Explorer."

Reading out the browser name and version—commonly known as **browser sniffing**—is not advisable, not only because of the inconsistencies we just encountered, but also because it makes your script dependent on a certain browser rather than supporting any user agent that is actually capable of supporting the script.

The solution to this problem is called **object detection**, and it basically means that we determine whether a user agent supports a certain object and make this our key differentiator. In really old scripts, like the first image rollovers, you might have seen something like this:

```
<script type="text/ javascript">
  // preloading images
  if(document.images)
```

```
  {
    // Images are supported
    var home=new Image();
    home.src='home.gif';
    var aboutus=new Image();
    aboutus.src='home.gif';
   }
</script>
```

The `if` condition checks whether the browser understands the `images` object, and only runs the code inside the condition if that is the case. For a long time, scripts like these were the standard way of dealing with images. In newer browsers, a lot of the JavaScript image effects can be achieved with CSS, effectively rendering scripts of this kind obsolete. However, JavaScript can manipulate images in ways CSS cannot, and we will come back to this in Chapter 6.

Every browser offers us the document it displays for manipulation, via something called the **Document Object Model**, or **DOM** for short. Older browsers supported their own DOMs, but there is a standard one, defined by the W3C, that is supported by most browsers these days. You might have encountered test scripts like this one in the past:

```
<script type="text/javascript">
  if(document.all)
  {
    // MSIE
  }
  else if (document.layers)
  {
    // Netscape Communicator
  }
  else if (document.getElementById)
  {
    // Mozilla, Opera 6, Safari
  }
</script>
```

The `document.all` DOM was invented by Microsoft and supported by MSIE 4 and 5, while Netscape 4 had its own DOM that supported `document.layers`. You can test for the W3C-recommended DOM via `document.getElementById`, and unless you have a genuine need to support MSIE versions previous to 5.5 or Netscape 4, it is the only one you really need to test for in this day and age.

One problem is that some in-between versions of Opera support the `getElementById` object, but fail to support the entire W3C DOM. To determine whether this is the case, test for support of the `createTextNode` as well:

```
<script type="text/javascript">
  if (document.getElementById && document.createTextNode)
  {
    // Mozilla, Opera 6, Safari
  }
</script>
```

If you embed this test in your scripts in this condition, you can be quite sure that only user agents that do support DOM scripting will execute what you've written. You can save yourself one code indentation by simply returning if neither of the two conditions is true.

```
<script type="text/javascript">
  if (!document.getElementById || !document.createTextNode){return;}
  // Other code
</script>
```

The same idea applies to any other methods to come in the future—as you are checking for a standard defined by the W3C, there is a big chance that user agents will support it—a much bigger chance than any other standard different vendors might come up with. You could also support different user agents on different levels, but this can lead to a lot of code duplication and less-optimized scripts.

Rather than catering to specific user agents, you test for the UA's capabilities before you apply your functionality—a process that is part of a bigger modern web design idea called **progressive enhancement**.

Progressive Enhancement

Progressive enhancement is the practice of providing functionality only to those who can see and use it by starting with a lowest common denominator and then testing whether successive improvements are supported. Users who don't have the capability to support those higher features will still be able to use the web site perfectly adequately. A comparable real-life process is to put on your clothes in the morning:

- You start with a naked body that is hopefully in full working condition—or at least in the same condition as it was yesterday, so that it is no shock to you (we discount PJs and/or underwear to keep this example easy).
- You may have a wonderful nude body, but it is insufficient in cold weather and might not appeal to other people around you—you'll need something to cover it with.
- If there are clothes available, you can check which fit the weather, your mood, the group of people you'll be seeing this day, and whether the different garments are in good order, clean, and are the right sizes.
- You put them on, and you can face the day. If you want to, you can start accessorizing, but please make sure to take other people into consideration when doing so (too much perfume might not be a good idea in a crowded train carriage).

In web development terms, this means the following:

- We start with a valid, semantically correct (X)HTML document with all the content—including relevant images with text alternatives as `alt` attributes—and a meaningful structure.
- We add a style sheet to improve this structure's appearance, legibility, and clarity—possibly we even add some simple rollover effects to liven it up a little.
- We add JavaScript:
  - The JavaScript starts when the document is loaded, by using the `window` object's `onload` event handler.
  - The JavaScript tests whether the current user agent supports the W3C DOM.
  - It then tests whether all the necessary elements are available and applies the desired functionality to them.

Before you can apply the idea of progressive enhancement in JavaScript, you'll need to learn how to access and interact with HTML and CSS from your scripts. I devote two chapters of this book to that task—Chapters 4 and 5. For the moment, however, it is enough to realize that the object detection we have practiced earlier helps us implement progressive enhancement—we make sure that only those browsers understanding the right objects will try to access them.

JavaScript and Accessibility

Web accessibility is the practice of making web sites usable by everybody, regardless of any disabilities they might have—for example, users with visual impairments may well use special software called **screen readers** to read out the web page content to them, and users with motor disabilities may well use a tool of some kind to manipulate the keyboard for navigating around the web, because they are unable to use the mouse. People with disabilities form a significant proportion of web users, so companies that choose not to allow them to use their web sites could well be missing out on a lot of business, and in some countries, legislation (such as Section 508 in the US) means that any sites that provide a public service have to be accessible, by law.

So where does JavaScript come into this? Outdated JavaScript techniques can be very bad for accessibility, because they can mess up the document flow, so, for example, screen readers cannot read them back to the user properly (especially when essential content is generated by JavaScript—there's a chance that the screen reader won't see it at all!) and thus force users to use the mouse to navigate their sites (for example, in the case of complicated DHTML whizbang navigation menus). The whole issue goes a lot deeper than this, but this is just to give you a feel for the area.

■ **Tip** If you want to read more about web accessibility, pick up a copy of *Web Accessibility: Web Standards and Regulatory Compliance*, by Jim Thatcher et al. (friends of ED, 2006).

JavaScript and accessibility is holy war material. Many a battle between disgruntled developers and accessibility gurus is fought on mailing lists, forums, and in chats, and the two sides all have their own—very good—arguments.

The developers who had to suffer bad browsers and illogical assumptions by marketing managers ("I saw it on my cousin's web site, surely you can also use it for our multinational portal") don't want to see years of research and trial and error go down the drain and not use JavaScript any longer.

The accessibility gurus point out that JavaScript can be turned off, that the accessibility guidelines by the W3C seem not to allow for it at all (a lot of confusion on that in the guidelines), and that a lot of scripts just assume that the visitors have and can use a mouse with the precision of a neurosurgeon.

Both are right, and both can have their cake: **there is no need to completely remove JavaScript from an accessible web site**.

What has to go is JavaScript that assumes too much. Accessible JavaScript has to ensure the following:

- The web document has to have the same content with and without JavaScript—no visitor should be blocked or forced to turn on JavaScript (as it is not always the visitor's decision whether he can turn it on).
- If there is content or HTML elements that only make sense when JavaScript is available, this content and those elements have to be created by JavaScript. Nothing is more frustrating than a link that does nothing or text explaining a slick functionality that is not available to you.
- All JavaScript functionality has to be independent of input device—for example, the user can be able to use a drag-and-drop interface, but should also be able to activate the element via clicking it or pressing a key.
- Elements that are not interactive elements in a page (practically anything but links and form elements) should not become interactive elements—unless you provide a fallback. Confusing? Imagine headlines that collapse and expand the piece of text that follows them. You can easily make them clickable in JavaScript, but that would mean that a visitor dependent on a keyboard will never be able to get to them. If you create a link inside the headlines while making them clickable, even that visitor will be able to activate the effect by "tabbing" to that link and hitting Enter.
- Scripts should not redirect the user automatically to other pages or submit forms without any user interaction. This is to avoid premature submission of forms—as some assistive technology will have problems with `onchange` event handlers. Furthermore, viruses and spyware send the user to other pages via JavaScript, and this is therefore blocked by some software these days.

That is all there is to make a web site with JavaScript accessible. That and, of course, all the assets of an accessible HTML document like allowing elements to resize with larger font settings, and providing enough contrast and colors that work for the color-blind as well as for people with normal vision.

Good Coding Practices

Now that I hopefully have gotten you into the mindset of forward-compatible and accessible scripting, let's go through some general best practices of JavaScript.

Naming Conventions

JavaScript is case dependent, which means that a variable or a function called `moveOption` is a different one than `moveoption` or `Moveoption`. Any name—no matter whether it is a function, an object, a variable, or an array, must only contain letters, numbers, the dollar sign, or the underscore character, and must not start with a number.

```
<script type="text/javascript">
  // Valid examples
  var dynamicFunctionalityId = 'dynamic';
  var parent_element2='mainnav';
  var _base=10;
  var error_Message='You forgot to enter some fields: ';

  // Invalid examples
  var dynamic ID='dynamic';  // Space not allowed!
  var 10base=10; // Starts with a number
  var while=10; // while is a JavaScript statement
</script>
```

The last example shows another issue: JavaScript has a lot of reserved words—basically all the JavaScript statements use reserved words like `while`, `if`, `continue`, `var`, or `for`. If you are unsure what you can use as a variable name, it might be a good idea to get a JavaScript reference. Good editors also highlight reserved words when you enter them to avoid the issue.

There is no length limitation on names in JavaScript; however, to avoid huge scripts that are hard to read and debug, it is a good idea to keep them as easy and descriptive as possible. Try to avoid generic names:

- `function1`
- `variable2`
- `doSomething()`

These do not mean much to somebody else (or yourself two months down the line) who tries to debug or understand the code. It is better to use descriptive names that tell exactly what the function does or what the variable is:

- `createTOC()`
- `calculateDifference()`
- `getCoordinates()`
- `setCoordinates()`
- `maximumWidth`
- `address_data_file`

As mentioned in previous chapters, you can use underscores or "camelCase" (that is, camel notation—lowercasing the first word and then capitalizing the first character of each word after that) to concatenate words; however, camelCase is more common (DOM itself uses it), and getting used to it will make it a lot easier for you to move on to more complex programming languages at a later stage. Another benefit of camelCase is that you can highlight a variable with a double-click in almost any editor, while you need to highlight an underscore-separated name with your mouse.

■ **Caution** Beware the lowercase letter *l* and the number *1*! Most editors will use a font face like courier, and they both look the same in this case, which can cause a lot of confusion and make for hours of fun trying to find bugs.

Code Layout

First and foremost, code is there to be converted by the interpreter to make a computer do something—or at least this is a very common myth. The interpreter will swallow the code without a hiccup when the code is valid—however, the real challenge for producing really good code is that a human will be able to edit, debug, amend, or extend it without spending hours trying to figure out what you wanted to achieve. Logical, succinct variable and function names are the first step to make it easier for the maintainer—the next one is proper code layout.

■ **Note** If you are really bored, go to any coder forum and drop an absolute like "spaces are better than tabs" or "every curly brace should get a new line." You are very likely to get hundreds of posts that point out the pros and cons of what you claimed. Code layout is a hotly discussed topic. The following examples work nicely for me and seem to be a quite common way of laying out code. It might be a good idea to check whether there are any contradictory standards to follow before joining a multideveloper team on a project and using the ones mentioned here.

Simply check the following code examples; you might not understand now what they do (they present a small function that opens every link that has a CSS class of `smallpopup` in a new window and adds a message that this is what will happen), but just consider which one would be easier to debug and change?

Without indentation:

```
function addPopUpLink(){
if(!document.getElementById||!document.createTextNode){return;}
var popupClass='smallpopup';
var popupMessage= '(opens in new window)';
var pop,t;
var as=document.getElementsByTagName('a');
for(var i=0;i<as.length;i++){
t=as[i].className;
if(t&&t.toString().indexOf(popupClass)!=-1){
as[i].appendChild(document.createTextNode(popupMessage));
as[i].onclick=function(){
pop=window.open(this.href,'popup','width=400,height=400');
returnfalse;
}}}}
window.onload=addPopUpLink;
```

With indentation:

```
function addPopUpLink(){
  if(!document.getElementById || !document.createTextNode){return;}
  var popupClass='smallpopup';
  var popupMessage= ' (opens in new window)';
  var pop,t;
  var as=document.getElementsByTagName('a');
  for(var i=0;i<as.length;i++){
    t=as[i].className;
    if(t && t.toString().indexOf(popupClass)!=-1){
      as[i].appendChild(popupMessage);
      as[i].onclick=function(){
        pop=window.open(this.href,'popup','width=400,height=400');
        return false;
      }
    }
  }
}
window.onload=addPopUpLink;
```

With indentation and curly braces on new lines:

```
function addPopUpLink()
{
  if(!document.getElementById || !document.createTextNode){return;}
  var popupClass='smallpopup';
  var popupMessage= ' (opens in new window)';
  var pop,t;
  var as=document.getElementsByTagName('a');
  for(var i=0;i<as.length;i++)
  {
    t=as[i].className;
    if(t && t.toString().indexOf(popupClass)!=-1)
    {
      as[i].appendChild(document.createTextNode(popupMessage));
      as[i].onclick=function()
      {
        pop=window.open(this.href,'popup','width=400,height=400');
        return false;
      }
    }
  }
}
window.onload=addPopUpLink;
```

I think it is rather obvious that indentation is a good idea; however, there is a big debate whether you should indent via tabs or spaces. Personally, I like tabs, mainly because they are easy to delete and less work to type in. Developers that work a lot on very basic (or pretty amazing, if you know all the cryptic keyboard shortcuts) editors like vi or emacs frown upon that, as the tabs might display as very large horizontal gaps. If that is the case, it is not much of a problem to replace all tabs with double spaces with a simple regular expression.

The question of whether the opening curly braces should get a new line or not is another you need to decide for yourself. The benefit of not using a new line is that it is easier to delete erroneous blocks, as they have one line less. The benefit of new lines is that the code does look less crammed. Personally, I keep the opening one on the same line in JavaScript and on a new line in PHP—as these seem to be the standard in those two developer communities.

Another question is line length. Most editors these days will have a line-wrap option that will make sure you don't have to scroll horizontally when you want to see the code. However, not all of them print out the code properly, and there may be a maintainer later on that has no fancy editor like that one. It is therefore a good idea to keep lines short—approximately 80 characters.

Commenting

Commenting is something that only humans benefit from—although in some higher programming languages, comments are indexed to generate documentation (one example is the PHP manual, which is at times a bit cryptic for nonprogrammers exactly because of this). While commenting is not a necessity for the code to work—if you use clear names and indent your

code, it should be rather self-explanatory—it can speed up debugging immensely. The previous example might make more sense for you with explanatory comments:

```
/*
  addPopUpLink
  opens the linked document of all links with a certain
  class in a pop-up window and adds a message to the
  link text that there will be a new window
*/
function addPopUpLink(){
  // Check for DOM and leave if it is not supported
  if(!document.getElementById || !document.createTextNode){return;}
  // Assets of the link - the class to find out which link should
  // get the functionality and the message to add to the link text
  var popupClass='smallpopup';
  var popupMessage= ' (opens in new window)';
  // Temporary variables to use in a loop
  var pop,t;
  // Get all links in the document
  var as=document.getElementsByTagName('a');
  // Loop over all links
  for(var i=0;i<as.length;i++)
  {
    t=as[i].className;
    // Check if the link has a class and the class is the right one
    if(t && t.toString().indexOf(popupClass)!=-1)
    {
      // Add the message
      as[i].appendChild(document.createTextNode(popupMessage));
      // Assign a function when the user clicks the link
      as[i].onclick=function()
      {
        // Open a new window with
        pop=window.open(this.href,'popup','width=400,height=400');
        // Don't follow the link (otherwise the linked document
        // would be opened in the pop-up and the document).
        return false;
      }
    }
  }
}
window.onload=addPopUpLink;
```

A lot easier to grasp, isn't it? It is also overkill. An example like this can be used in training documentation or a self-training course, but it is a bit much in a final product—moderation is always the key when it comes to commenting. In most cases, it is enough to explain what something does and what can be changed.

```
/*
  addPopUpLink
  opens the linked document of all links with a certain
  class in a pop-up window and adds a message to the
  link text that there will be a new window
*/
function addPopUpLink()
{
  if(!document.getElementById || !document.createTextNode){return;}

  // Assets of the link - the class to find out which link should
  // get the functionality and the message to add to the link text
  var popupClass='smallpopup';
  var popupMessage=document.createTextNode(' (opens in new window)');

  var pop,t;
  var as=document.getElementsByTagName('a');
  for(var i=0;i<as.length;i++)
  {
    t=as[i].className;
    if(t && t.toString().indexOf(popupClass)!=-1)
    {
      as[i].appendChild(popupMessage);
      as[i].onclick=function()
      {
        pop=window.open(this.href,'popup','width=400,height=400');
        return false;
      }
    }
  }
}
window.onload=addPopUpLink;
```

These comments make it easy to grasp what the whole function does and to find the spot where you can change some of the settings. This makes quick changes easier—changes in functionality would need the maintainer to analyze your code more closely anyway.

Functions

Functions are reusable blocks of code and are an integral part of most programs today, including those written in JavaScript. Imagine you have to do a calculation or need a certain conditional check over and over again. You could copy and paste the same lines of code where necessary; however, it is much more efficient to use a function.

Functions can get values as parameters (sometimes called **arguments**) and can return values after they finished testing and changing what has been given to them.

You create a function by using the `function` keyword followed by the function name and the parameters separated by commas inside parentheses:

```
function createLink(linkTarget, LinkName)
{
  // Code
}
```

There is no limit as to how many parameters a function can have, but it is a good idea not to use too many, as it can become rather confusing. If you check some DHTML code, you can find functions with 20 parameters or more, and remembering their order when calling those in other functions will make you almost wish to simply write the whole thing from scratch. When you do that, it is a good idea to remember that too many parameters mean a lot more maintenance work and make debugging a lot harder than it should be.

Unlike PHP, JavaScript has no option to preset the parameters should they not be available. You can work around this issue with some `if` conditions that check whether the parameter is `null` (which means "nothing, not even 0" in interpreter speak):

```
function createLink(linkTarget, LinkName)
{
  if (linkTarget == null)
  {
    linkTarget = '#';
  }
  if (linkName == null)
  {
    linkName = 'dummy';
  }
}
```

Functions report back what they have done via the `return` keyword. If a function that's invoked by an event handler returns the Boolean value `false`, then the sequence of events that is normally triggered by the event gets stopped. This is very handy when you want to apply functions to links and stop the browser from navigating to the link's `href`. We also used this in the "Object Detection vs. Browser Dependence" section.

Any other value following the `return` statement will be sent back to the calling code. Let's change our `createLink` function to create a link and return it once the function has finished creating it.

```
function createLink(linkTarget,linkName)
{
  if (linkTarget == null) { linkTarget = '#'; }
  if (linkName == null) { linkName = 'dummy'; }

  var tempLink=document.createElement('a');
```

```
  tempLink.setAttribute('href',linkTarget);
  tempLink.appendChild(document.createTextNode(linkName));

  return tempLink;
}
```

Another function could take these generated links and append them to an element. If there is no element ID defined, it should append the link to the body of the document.

```
function appendLink(sourceLink,elementId)
{
  var element=false;
  if (elementId==null || !document.getElementById(elementId))
  {
    element=document.body;
  }
  if(!element) {
    element=document.getElementById(elementId);
  }
  element.appendChild(sourceLink);
}
```

Now, to use both these functions, we can have another one call them with appropriate parameters:

```
function linksInit()
{
  if (!document.getElementById || !document.createTextNode) { return; }
  var openLink=createLink('#','open');
  appendLink(openLink);
  var closeLink=createLink('closed.html','close');
  appendLink(closeLink,'main');
}
```

The function `linksInit()` checks whether DOM is available (as it is the only function calling the others, we don't need to check for it inside them again) and creates a link with a target of # and `open` as the link text.

It then invokes the `appendLink()` function and sends the newly generated link as a parameter. Notice it doesn't send a target element, which means `elementId` is `null` and `appendLink()` adds the link to the main body of the document.

The second time `initLinks()` invokes `createLink()`, it sends the target `closed.html` and `close` as the link text and applies the link to the HTML element with the ID `main` via the `appendLink()` function. If there is an element with the ID `main`, `appendLink()` adds the link to this one; if not, it uses the document body as a fallback option.

If this is confusing now, don't worry, you will see more examples later on. For now, it is just important to remember what functions are and what they should do:

- Functions are there to do one task over and over again—keep each task inside its own function; don't create monster functions that do several things at once.
- Functions can have as many parameters as you wish, and each parameter can be of any type—string, object, number, variable, or array.
- You cannot predefine parameters in the function definition itself, but you can check whether they were defined or not and set defaults with an `if` condition. You can do this very succinctly via the ternary operator, which you will get to know in the next section of this chapter.
- Functions should have a logical name describing what they do; try to keep the name close to the task topic, as a generic `init()`, for example, could be in any of the other included JavaScript files and overwrite their functions. Object literals can provide one way to avoid this problem, as you'll see later in this chapter.

Short Code via Ternary Operator

Looking at the `appendLink()` function shown earlier, you might get a hunch that a lot of `if` conditions or `switch` statements can result in very long and complex code. A trick to avoid some of the bloating involves using something called the **ternary operator**. The ternary operator has the following syntax:

```
var variable = condition ? trueValue:falseValue;
```

This is very handy for Boolean conditions or very short values. For example, you can replace this long `if` condition with one line of code:

```
// Normal syntax
var direction;
If(x<200)
{
  direction=1;
}
else
{
  direction=-1
}
// Ternary operator
var direction = x < 200 ? 1 : -1;
```

Other examples:

```
t.className = t.className == 'hide' ? '' : 'hide';
var el = document.getElementById('nav')
       ? document.getElementById('nav')
       : document.body;
```

You can also nest the ternary selector, but that gets rather unreadable:

```
y = x <20 ? (x > 10 ? 1 : 2) : 3;
// equals
if(x<20)
{
  if(x>10)
  {
    y=1;
  }
  else
  {
    y=2;
  }
}
else
{
 y=3
}
```

Sorting and Reuse of Functions

If you have a large number of JavaScript functions, it might be a good idea to keep them in separate `.js` files and apply them only where they are needed. Name the `.js` files according to what the functions included in them do, for example, `formvalidation.js` or `dynamicmenu.js`.

This has been done to a certain extent for you, as there are a lot of prepackaged JavaScript libraries (collections of functions and methods) that help create special functionality. We will look at some of them in Chapter 11 and create our own during the next few chapters.

Variable and Function Scope

Variables defined inside a function with a new `var` are only valid inside this function, not outside it. This might seem a drawback, but it actually means that your scripts will not interfere with others—which could be fatal when you use JavaScript libraries or your own collections.

Variables defined outside functions are called `global` variables and are dangerous. We should try to keep all our variables contained inside functions. This ensures that our script will play nicely with other scripts that may be applied to the page. Many scripts use generic variable names like `navigation` or `currentSection`. If these are defined as global variables, the scripts will override each other's settings. Try running the following function to see what omitting a `var` keyword can cause:

```
<script type="text/javascript">
  var demoVar=1 // Global variable
  alert('Before withVar demoVar is' +demoVar);
  function withVar()
  {
    var demoVar=3;
  }
  withVar();
  alert('After withVar demoVar is' +demoVar);
  function withoutVar()
  {
    demoVar=3;
  }
  withoutVar();
  alert('After withoutVar demoVar is' +demoVar);
</script>
```

While `withVar` keeps the variable untouched, `withoutVar` changes it:

```
Before withVar demoVar is 1
After withVar demoVar is 1
After withoutVar demoVar is 3
```

Keeping Scripts Safe with the Object Literal

Earlier we talked about keeping variables safe by defining them locally via the `var` keyword. The reason was to avoid other functions relying on variables with the same name and the two functions overwriting each other's values. The same applies to functions. As you can include several JavaScripts to the same HTML document in separate script elements your functionality might break as another included document has a function with the same name. You can avoid this issue with a naming convention, like `myscript_init()` and `myscript_validate()` for your functions. However, this is a bit cumbersome, and JavaScript offers a better way to deal with this in the form of objects.

You can define a new object and use your functions as methods of this object—this is how JavaScript objects like `Date` and `Math` work. For example:

```
<script type="text/javascript">
  myscript=new Object();
  myscript.init=function()
  {
    // Some code
  };
  myscript.validate=function()
  {
   // Some code
  };
</script>
```

Notice that if you try to call the functions `init()` and `validate()`, you get an error, as they don't exist any longer. Instead, you need to use `myscript.init()` and `myscript.validate()`.

Wrapping all your functions in an object as methods is analogous to the programming classes used by some other languages such as C++ or Java. In such languages, you keep functions that apply to the same task inside the same class, thus making it easier to create large pieces of code without getting confused by hundreds of functions.

The syntax we used is still a bit cumbersome, as you have to repeat the object name over and over again. There is a shortcut notation called the **object literal** that makes it a lot easier.

The object literal has been around for a long time but has been pretty underused. It is becoming more and more fashionable nowadays, and you can pretty much presume that a script you find on the web using it is pretty good, modern JavaScript.

What the object literal does is use a shortcut notation to create the object and apply each of the functions as object methods instead of stand-alone functions. Let's see our three functions of the dynamic links example as a big object using the object literal:

```
var dynamicLinks={
  linksInit:function()
  {
    if (!document.getElementById || !document.createTextNode)➥
       { return; }
    var openLink=dynamicLinks.createLink('#','open');
    dynamicLinks.appendLink(openLink);
    var closeLink=dynamicLinks.createLink('closed.html','close');
    dynamicLinks.appendLink(closeLink,'main');
  },
  createLink:function(linkTarget,linkName)
  {
    if (linkTarget == null) { linkTarget = '#'; }
    if (linkName == null) { linkName = 'dummy'; }
    var tempLink=document.createElement('a');
    tempLink.setAttribute('href',linkTarget);
    tempLink.appendChild(document.createTextNode(linkName));
    return tempLink;
  },
  appendLink:function(sourceLink,elementId)
  {
    var element=false;
    if (elementId==null || !document.getElementById(elementId))
    {
      element=document.body;
    }
    if(!element){element=document.getElementById(elementId)}
    element.appendChild(sourceLink);
  }
}

window.onload=dynamicLinks.linksInit;
```

As you can see, all the functions are contained as methods inside the `dynamicLinks` object, which means that if we want to call them, we need to add the name of the object before the function name.

The syntax is a bit different; instead of placing the `function` keyword before the name of the function, we add it behind the name preceded by a colon. Additionally, each closing curly brace except for the very last one needs to be followed by a comma.

If you want to use variables that should be accessible by all methods inside the object, you can do that with syntax that is quite similar:

```
var myObject=
{
  objMainVar:'preset',
  objSecondaryVar:0,
  objArray:['one','two','three'],
  init:function(){},
  createLinks:function(){},
  appendLinks:function(){}
}
```

This might be a lot right now, but don't worry. This chapter is meant as a reference for you to come back to and remind you of a lot of good practices in one place. We will continue in the next chapter with more tangible examples, and rip open an HTML document to play with the different parts.

Summary

You have done it; you finished this chapter, and you should now be able to separate modern and old scripts when you see them on the Web. Older scripts are likely to

- Use a lot of `document.write()`.
- Check for browsers and versions instead of objects.
- Write out a lot of HTML instead of accessing what is already in the document.
- Use proprietary DOM objects like `document.all` for MSIE and `document.layers` for Netscape Navigator.
- Appear anywhere inside the document (rather in the `<head>` or included via `<script src="---.js">`) and rely on `javascript:` links instead of assigning events.

You've learned about putting JavaScript into stand-alone `.js` documents instead of embedding it into HTML and thereby separating behavior from structure.

You then heard about using object detection instead of relying on browser names and what progressive enhancement means and how it applies to web development. Testing user agent capabilities instead of names and versions will ensure that your scripts also work for user agents you might not have at hand to test yourself. It also means that you don't have to worry every time there is a new version of a browser out—if it supports the standards, you'll be fine.

We talked about accessibility and what it means for JavaScript, and you got a peek at a lot of coding practices. The general things to remember are

- Test for the objects you want to use in your scripts.
- Make improvements in an existing site that already works well without client-side scripting instead of adding scripts first and adding nonscripting fallbacks later on.
- Keep your code self-contained and don't use any global variables that might interfere with other scripts.
- Code with the idea in mind that you will have to hand this code over to someone else to maintain. This person might be you in three months' time, and you should be able to immediately understand what is going on.
- Comment your code's functionality and use readable formatting to make it easy to find bugs or change the functions.

This is the lot—except for something called an event handler, which I've talked about but not actually defined. I'll do so in Chapter 5. But for now, sit back, get a cup of coffee or tea, and relax for a bit, until you're ready to proceed with learning how JavaScript interacts with HTML and CSS.