CHAPTER 3

The Document Object Model

What this chapter covers:

- The concept of nodes
- Five very handy DOM methods: `getElementById`, `getElementsByTagName`, `getElementsByClassName`, `getAttribute`, and `setAttribute`

It's time to meet the DOM. This chapter introduces you to the DOM and shows you the world through its eyes.

D is for document

The DOM can't work without a document. When you create a web page and load it in a web browser, the DOM comes to life. It takes the document that you have written and turns it into an object.

In normal everyday English, the word *object* isn't very descriptive. It simply means *thing*. In programming languages, on the other hand, *object* has a very specific meaning.

Objects of desire

At the end of the previous chapter, you saw some examples of objects in JavaScript. You'll remember that objects are self-contained bundles of data. Variables associated with an object are called *properties* of the object. Functions that can be executed by an object are called *methods* of the object.

There are three kinds of objects in JavaScript:

- User-defined objects created from scratch by the programmer. We won't be dealing with these in this book.
- Native objects like `Array`, `Math`, and `Date`, which are built in to JavaScript.
- Host objects that are provided by the browser.

From the earliest days of JavaScript, some very important host objects have been made available for scripting. The most fundamental of these is the `window` object.

The `window` object is nothing less than a representation of the browser window itself. The properties and methods of the `window` object are often referred to as the Browser Object Model (although perhaps Window Object Model would be more semantically correct). The Browser Object Model has methods like `window.open` and `window.blur`. These methods, incidentally, are responsible for all those annoying pop-up and pop-under windows that have plagued the Web. No wonder JavaScript has a bad reputation!

Fortunately, we won't be dealing with the Browser Object Model very much. Instead, we'll focus on what's inside the browser window. The object that handles the contents of a web page is the `document` object. For the rest of this book, we're going to be dealing almost exclusively with the properties and methods of the `document` object.

That explains the letter *D* (document) and the letter *O* (object) in DOM. But what about the letter *M*?

Dial M for model

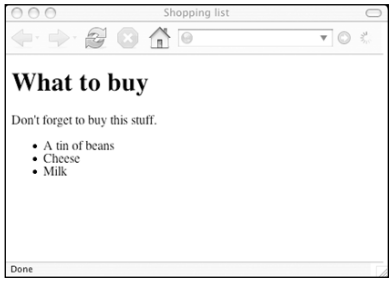
The *M* in DOM stands for Model, but it could just as easily stand for Map. A model, like a map, is a representation of something. A model train represents a real train. A street map of a city represents the real city. The DOM represents the web page that's currently loaded in the browser window. The browser provides a map (or model) of the page. You can use JavaScript to read this map.

Maps make use of conventions like direction, contours, and scale. In order to read a map, you need to understand these conventions—and it's the same with the DOM. In order to gain information from the model, you need to understand what conventions are being used to represent the document.

The most important convention used by the DOM is the representation of a document as a tree. More specifically, the document is represented as a family tree.

A family tree is another example of a model. A family tree represents a real family, describes the relationships between family members, and uses conventions like *parent*, *child,* and *sibling*. These can be used to represent some fairly complex relationships. For example, one member of a family can be a parent to others while also being the child of another family member and the sibling of yet another family member.

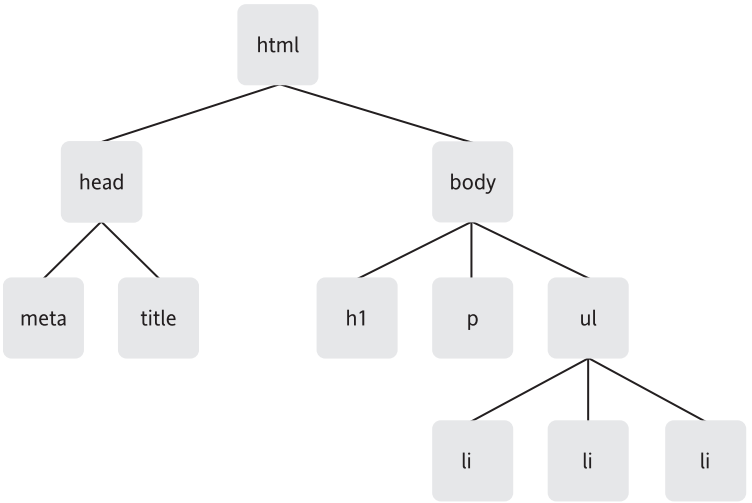
The family tree model works just as well in representing a document written in (X)HTML. Take a look at this very basic web page:




```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li class="sale">Cheese</li>
      <li class="sale important">Milk</li>
    </ul>
  </body>
```

```
</html>
```

This can be represented by the model shown here.



Let's walk through the web page structure to see what it's made up of, and show why it's represented so well by the model shown previously. After the `DOCTYPE` declaration, the document begins by opening an `<html>` tag. All the other elements of the web page are contained within this element, meaning it is a parent. Because all the other elements are inside, the `<html>` tag has no parent itself. It also has no siblings. If this were a tree, the `<html>` tag would be the root.

The root element is `html`. For all intents and purposes, the `html` element *is* the document.

If we move one level deeper, we find two branches: `<head>` and `<body>`. They exist side by side, which makes them siblings. They share the same parent, `<html>`, but they also contain children, so they are parents themselves.

The `<head>` element has two children: `<meta>` and `<title>` (siblings of one another). The children of the `<body>` element are `<h1>`, `<p>`, and `<ul>` (all siblings of one another). If we drill down deeper still, we find that `<ul>` is also a parent. It has three children, all of them `<li>` elements with a few `class` attributes.

By using this simple convention of familial relationships, we can access a lot of information about the relationship between elements. For example, what is the relationship between `<h1>` and `<p>`? The answer is that they are siblings. What is the relationship between `<body>` and `<ul>`? `<body>` is the parent of `<ul>`. `<ul>` is a child of `<body>`.

If you can think of the elements of a document in terms of a tree of familial relationships, then you're using the same terms as the DOM. However, instead of using the term *family tree*, it's more accurate to call a document a *node tree*.

Nodes

The term *node* comes from networking, where it used to denote a point of connection in a network. A network is a collection of nodes.

In the real world, everything is made up of atoms. Atoms are the nodes of the real world. But atoms can themselves be broken down into smaller, subatomic particles. These subatomic particles are also considered nodes.

It's a similar situation with the DOM. A document is a collection of nodes, with nodes as the branches and leaves on the document tree.

There are a number of different types of nodes. Just as atoms contain subatomic particles, some types of nodes contain other types of nodes. Let's take a quick look at three of them: element, text, and attribute nodes.

Element nodes

The DOM's equivalent of the atom is the element node**.**

When I described the structure of the shopping list document, I did so in terms of elements such as `<body>`, `<p>`, and `<ul>`. Elements are the basic building blocks of documents on the Web, and it's the arrangement of these elements in a document that gives the document its structure.

The tag provides the name of an element. Paragraph elements have the name `p`, unordered lists have the name `ul`, and list items have the name `li`.

Elements can contain other elements. All the list item elements in our document are contained within an unordered list element. In fact, the only element that isn't contained within another element is the `<html>` element. It's the root of our node tree.

Text nodes

Elements are just one type of node. If a document consisted purely of empty elements, it would have a structure, but the document itself wouldn't contain much content. On the Web, where content is king, most content is provided by text.

In our example, the `<p>` element contains the text "Don't forget to buy this stuff." This is a text node**.**

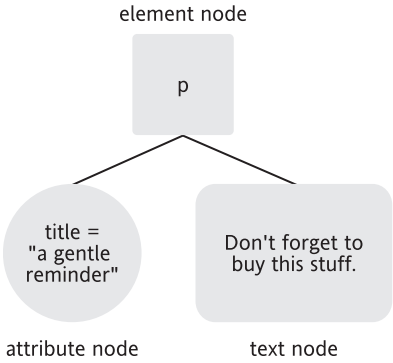
In XHTML, text nodes are always enclosed within element nodes. But not all elements contain text nodes. In our shopping list document, the `<ul>` element doesn't contain any text directly. It contains other element nodes (the `<li>` elements), and these contain text nodes.

Attribute nodes

Attributes are used to give more specific information about an element. The `title` attribute, for example, can be used on just about any element to specify exactly what the element contains:

```
<p title="a gentle reminder">Don't forget to buy this stuff.</p>
```

In the DOM, `title="a gentle reminder"` is an attribute node, as shown in the diagram here. Because attributes are always placed within opening tags, attribute nodes are always contained within element nodes. Not all elements contain attributes, but all attributes are contained by elements.



In our example document, the unordered list (`<ul>`) has been marked with the `id` attribute and some of the list elements (`<li>`) have been marked with the `class` attribute. You're probably familiar with the `id` and `class` attributes from using CSS. Just in case you do not have much experience with CSS, the next section briefly recaps the basics.

Cascading Style Sheets

The DOM isn't the only technology that interacts with the structure of web pages. CSS is used to instruct a browser how to display the contents of a document.

Styles are declared either in the `<head>` of a document (between `<style>` tags) or in an external style sheet (you'll be working with styles in Chapter 4). The syntax for styling an element with CSS is similar to that of JavaScript functions:

```
selector {
 property: value;
}
```

Style declarations can be used to specify the colors, fonts, and sizes used by the browser to display elements:

```
p {
  color: yellow;
  font-family: "arial", sans-serif;
  font-size: 1.2em;
}
```

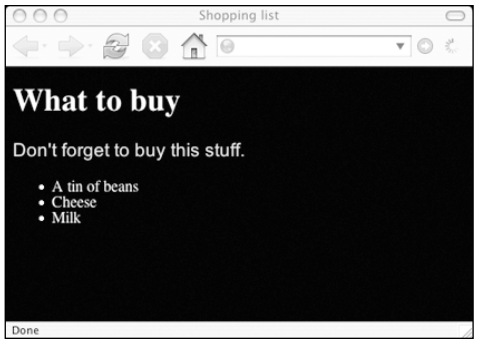
Inheritance is a powerful feature of CSS. Like the DOM, CSS views the contents of a document as a node tree. Elements that are nested within the node tree will inherit the style properties of their parents.

For instance, declaring colors or fonts on the `body` element will automatically apply those styles to all the elements contained within the `body`:

```
body {
  color: white;
  background-color: black;
}
```

Those colors will be applied not just to content contained directly by the `<body>` tag, but also by elements nested within the `body`.

The image shown here is a basic web page with styles applied.



When you're applying styles to a document, there are times when you will want to target specific elements. You might want to make one paragraph a certain size and color, but leave other paragraphs unaffected. To get this level of precision, you need to insert something into the document itself to mark the paragraph as a special case. To mark elements for special treatment, you can use one of two attributes: `class` or `id`.

The class attribute

The `class` attribute can be applied as often as you like to as many different elements as you like:

```
<p class="special">This paragraph has the special class</p>
<h2 class="special">So does this headline</h2>
```

In a style sheet, styles can then be applied to all the elements of this `class`:

```
.special {
  font-style: italic;
}
```

You can also target specific types of elements with this `class`:

```
h2.special {
  text-transform: uppercase;
}
```

The id attribute

The `id` attribute can be used once in a web page to uniquely identify an element:

```
<ul id="purchases">
```

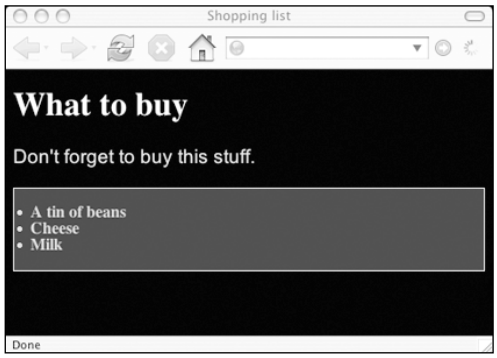
In a style sheet, styles can then be applied specifically to this element:

```
#purchases {
  border: 1px solid white;
  background-color: #333;
  color: #ccc;
  padding: 1em;
}
```

Although the `id` itself can be applied only once, a style sheet can use the `id` to apply styles to elements nested within the uniquely identified element:

```
#purchases li {
  font-weight: bold;
}
```

Here is an example of styles applied to a list with a unique `id`:



The `id` attribute acts as a kind of "hook" that can be targeted by CSS. The DOM can use the same hook.

Getting Elements

Three hn 306.210dy DOM methods allow you to access element nodes by ID, tag name, and class name.

getElementById

The DOM has a method called `getElementById`, which does exactly what it sounds like: it allows you to get straight to the element node with the specified `id`. Remember that JavaScript is case-sensitive, so `getElementById` must always be written with case preserved. If you write `GetElementById` or `getElementbyid`, you won't get the results you expect.

`getElementById` is a function associated with the `document` object. Functions are always followed by parentheses that contain the function's arguments. `getElementById` takes just one argument: the `id` of the element you want to get to, contained in either single or double quotes.

```
document.getElementById(id)
```

Here's an example:

```
document.getElementById("purchases")
```

This is referencing the unique element that has been assigned the HTML `id` attribute `"purchases"` in the `document` object. This element also corresponds to an object. You can test this for yourself by using the `typeof` operator. This will tell you whether something is a string, a number, a function, a Boolean value, or an object.

I don't recommend mixing a lot of inline JavaScript into a document, but purely for testing purposes, insert this `<script>` with JavaScript into the shopping list document. Put it directly before the closing `</body>` tag:

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li class="sale">Cheese</li>
      <li class="sale important">Milk</li>
    </ul>
    <script>
      alert(typeof document.getElementById("purchases"));
    </script>
  </body>
</html>
```

When you load the XHTML file in a web browser, you will be greeted with an annoying pop-up box stating the nature of `document.getElementById("purchases")`. It is an object.

In fact, each element in a document is an object. Using the DOM, you can "get" at all of these elements. Obviously you shouldn't give a unique `id` to every single element in a document. That would be overkill. Fortunately, the `document` object provides another method for getting at elements that don't have unique identifiers.

getElementsByTagName

If you use the method `getElementsByTagName`, you have instant access to an array populated with every occurrence of a specified tag. Like `getElementById`, `getElementsByTagName` is a function that takes one argument. In this case, the argument is the name of a tag:

element`.getElementsByTagName(`*tag*`)`

It looks very similar to `getElementById`, but notice that this time you can get elements—plural. Be careful when you are writing your scripts that you don't inadvertently write `getElementsById` or `getElementByTagName`.

Here it is in action:

```
document.getElementsByTagName("li")
```

This is an array populated with all the list items in the `document` object. Just as with any other array, you can use the `length` property to get the total number of elements.

Delete the `alert` statement you placed between `<script>` tags earlier and replace it with this:

```
alert(document.getElementsByTagName("li").length);
```

This will tell you the total number of list items in the document, which in this case is three. Every value in the array is an object. You can test this yourself by looping through the array and using `typeof` on each value. For example, try this with a `for` loop:

```
for (var i=0; i < document.getElementsByTagName("li").length; i++) {
  alert(typeof document.getElementsByTagName("li")[i]);
}
```

Even if there is only one element with the specified tag name, `getElementsByTagName` still returns an array. The `length` of the array will simply be 1.

Now you will begin to notice that typing out `document.getElementsByTagName("li")` every time is tedious, and that the code is getting harder to read. You can reduce the amount of unnecessary typing and improve readability by assigning a variable to contain `document.getElementsByTagName("li")`. Replace the `alert` statement between the `<script>` tags with these statements:

```
var items = document.getElementsByTagName("li");
for (var i=0; i < items.length; i++) {
  alert(typeof items[i]);
}
```

Now you'll get three annoying alert boxes, each saying the same thing: **object**.

You can also use a wildcard with `getElementsByTagName`, which means you can make an array with every single element. The wildcard symbol (the asterisk) must be contained in quotes to distinguish it from the multiplication operator. The wildcard will give you the total number of element nodes in a document:

```
alert(document.getElementsByTagName("*").length);
```

You can also combine `getElementsByTagName` with `getElementById`. So far, we've applied `getElementsByTagName` to only the `document` object. But if you're interested in finding out how many list items are inside the element with the `id` `"purchases"`, you could apply `getElementsByTagName` to that specific object:

```
var shopping = document.getElementById("purchases");
var items = shopping.getElementsByTagName("*");
```

Now the `items` array holds just the elements contained by the `"purchases"` list. In this case, that happens to be the same as the total number of the list items in the document:

```
alert (items.length);
```

And, if any further proof is needed, you can test that each one is an object:

```
for (var i=0; i < items.length; i++) {
  alert(typeof items[i]);
}
```

getElementsByClassName

A welcome addition to the HTML5 DOM (`http://www.whatwg.org/specs/web-apps/current-work/`) is the `getElementsByClassName` method. This method adds a way of accessing elements by their class names in the `class` attribute. That said, it is still a relatively new addition to the DOM, so you need to be cautious

when you choose to use it. First, let's see what it does, and then we'll discuss how we can make it work reliably.

Like `getElementsByTagName`, the `getElementsByClassName` method takes one argument. In this case, the argument is the name of a class:

```
getElementsByClassName(class)
```

It also behaves in the same way as `getElementsByTagName` by returning an array of elements with a common class name. Here's an example that will return all the elements that have `"sale"` in their `class` attribute:

```
document.getElementsByClassName("sale")
```

You can also include multiple class names to locate elements that have more than one class. To include more than one class, simply separate the class names with a space. For example, add the following `alert` statement between the `<script>` tags:

```
alert(document.getElementsByClassName("important sale").length);
```

Notice that you'll get a **1** in the alert box. Only one element matched, because only one has both the `"important"` and `"sale"` class. Also notice that the function still matches, even though the `class` attribute in the document is `"sale important"` not `"important sale"`. The order of the class names doesn't matter, and it would match if the element had additional class names as well.

As with `getElementsByTagName`, you can also combine `getElementsByClassName` with `getElementById`. If you want to find out how many list items have the `"sale"` class inside the element with the `id` `"purchases"`, you can apply `getElementsByClassName` to that specific object:

```
var shopping = document.getElementById("purchases");
var sales = shopping.getElementsByClassName("sale");
```

Now the `sales` array contains just the elements with the `"sale"` class that are also contained by the `"purchases"` list. If you try the following `alert`, you'll see that happens to be two items:

```
alert (sales.length);
```

The `getElementsByClassName` method is quite useful, but it's supported by only modern browsers. To make up for this lack of support, DOM scripters have needed to roll their own `getElementsByClassName` function using existing DOM methods, sort of like a rite of passage. In most cases, these functions look similar to the following `getElementsByClassName` function, and are written to work in both older and newer browsers:

```
function getElementsByClassName(node, classname) {
  if (node.getElementsByClassName) {
    // Use the existing method
    return node.getElementsByClassName(classname);
  } else {
    var results = new Array();
    var elems = node.getElementsByTagName("*");
    for (var i=0; i<elems.length; i++) {
      if (elems[i].className.indexOf(classname) != -1) {
        results[results.length] = elems[i];
      }
    }
    return results;
  }
}
```

This `getElementsByClassName` function takes two arguments. The first is `node`, which represents the point in the tree where the search will start, and the second is `classname`, which is the class to search for. If the proper `getElementsByClassName` function already exists as a method on the `node`, then the new function will return the list of nodes as expected. If the `getElementsByClassName` doesn't exist, the new function loops through all the tags and crudely looks for the elements with the appropriate class name. (This example doesn't work as well for multiple class names). Using this function to mimic what we did earlier with the shopping list, we might do something like this:

```
var shopping = document.getElementById("purchases");
var sales = getElementsByClassName(shopping, "sale");
```

How you choose to search for the matching DOM elements can vary widely, however, there are a few excellent examples, such as "The Ultimate getElementsByClassName" by Robert Nyman (`http://robertnyman.com/2008/05/27/the-ultimate-getelementsbyclassname-anno-2008/`).

In Chapter 5, we'll discuss these support issues and how to best deal with them. In Chapter 7, we'll look at the DOM manipulation methods in more detail.

Taking stock

By now, you are probably truly fed up with seeing alert boxes containing the word **object**. I think you get the point: every element node in a document is an object. Not only that, but every single one of these objects comes with an arsenal of methods, courtesy of the DOM. Using these supplied methods, you can retrieve information about any element in a document. You can even alter the properties of an element.

Here's a quick summary of what you've seen so far:

- A document is a tree of nodes.
- There are different types of nodes: elements, attributes, text, and so on.
- You can go straight to a specific element node using `getElementById`.
- You can go directly to a collection of element nodes using `getElementsByTagName` or `getElementsByClassName`.
- Every one of these nodes is an object.

Now let's look at some of the properties and methods associated with these objects.

Getting and Setting Attributes

So far, you've seen three different ways of getting to element nodes: using `getElementById`, `getElementsByTagName`, or `getElementsByClassName`. Once you have the element, you can find out the values of any of its attributes. You can do this with the `getAttribute` method. And using the `setAttribute` method, you can change the value of an attribute node.

getAttribute

`getAttribute` is a function. It takes only one argument, which is the attribute that you want to get:

`object``.getAttribute(`*`attribute`*`)`

Unlike the other methods you've seen in this chapter, you can't use `getAttribute` on the `document` object. It can be used on only an element node object. For example, you can use it in combination with `getElementsByTagName` to get the `title` attribute of every `<p>` element:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i < paras.length; i++ ) {
  alert(paras[i].getAttribute("title"));
}
```

If you include this code at the end of our shopping list document and then load the page in a web browser, you'll be greeted with an alert box containing the text **a gentle reminder**.

In our shopping list, there is only one `<p>` element, and it has a `title` attribute. If there were more `<p>` elements and they didn't have `title` attributes, then `getAttribute("title")` would return the value `null`. In JavaScript, `null` means that there is no value. You can test this for yourself by inserting this paragraph directly after the existing paragraph:

```
<p>This is just a test</p>
```

Now reload the page. This time, you'll see two alert boxes. The second one is either completely empty or simply says **null**, depending on how the browser chooses to display `null` values.

We can modify our script so that it pops up a message only when a `title` attribute exists. We will add an `if` statement to check that the value returned by `getAttribute` is not `null`. While we're at it, let's use a few more variables to make the script easier to read.

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text != null) {
    alert(title_text);
  }
}
```

Now if you reload the page, you will see only the alert box that contains the value **a gentle reminder**.

We can shorten the code even more. Whenever you want to check that something isn't `null`, you're really checking to see if it exists. A shorthand way of doing that is to use it as the condition in an `if` statement. `if (something)` is a shorthand way of writing `if (something != null)`. The condition of the `if` statement will be true if `something` exists. It will be false if `something` doesn't exist.

We can tighten up our code by simply writing `if (title_text)` instead of `if (title_text != null)`. While we're at it, we can put the `alert` statement on the same line as the `if` statement so that it reads more like English:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text) alert(title_text);
}
```

setAttribute

All of the methods you've seen so far have dealt with retrieving information. `setAttribute` is a bit different. It allows you to change the value of an attribute node. Like `getAttribute`, `setAttribute` works only on element nodes.

`setAttribute` takes two arguments:

object.setAttribute(*attribute*,*value*)

In the following example, we're getting the element with the `id` `"purchases"` and giving it a `title` attribute with the value `"a list of goods"`.

```
var shopping = document.getElementById("purchases");
shopping.setAttribute("title","a list of goods");
```

You can use `getAttribute` to test that the `title` attribute has been set:

```
var shopping = document.getElementById("purchases");
alert(shopping.getAttribute("title"));
shopping.setAttribute("title","a list of goods");
alert(shopping.getAttribute("title"));
```

Loading the page will now give you two alert boxes. The first one, which is executed before `setAttribute`, is empty or displays **null**. The second one, which is executed after the `title` attribute has been set, says **a list of goods**.

In our example, we set an attribute where previously none had existed. The `setAttribute` method created the attribute and then set its value. If you use `setAttribute` on an element node that already has the specified attribute, the old value will be overwritten.

In our shopping list document, the `<p>` element already has a `title` attribute with the value `"a gentle reminder"`. You can use `setAttribute` to change this value:

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text) {
    paras[i].setAttribute("title","brand new title text");
    alert(paras[i].getAttribute("title"));
  }
}
```

This will apply the value `"brand new title text"` to the `title` attribute of every `<p>` element in the document that already had a `title` attribute. In our shopping list document, the value `"a gentle reminder"` has been overwritten.

Note that even when a document has been changed by `setAttribute`, you won't see that change reflected when you use the view source option in your web browser. This is because the DOM has

dynamically updated the contents of the page after it has loaded. The real power of the DOM is that the contents of a page can be updated without refreshing the page in the browser.

What's next?

This chapter demonstrated five methods provided by the DOM:

```
getElementById
getElementsByTagName
getElementsByClassName
getAttribute
setAttribute
```

These five methods will be the cornerstones for many of the DOM scripts you're going to write.

The DOM offers many more methods and properties. There's `nodeName`, `nodeValue`, `childNodes`, `nextSibling`, and `parentNode`—to name just a few. I'll explain each one in turn as and when they're needed. I'm mentioning them now just to whet your appetite.

You've read through a lot of theory in this chapter. I hope by now you're itching to test the power of the DOM using something other than alert boxes. It's high time we applied the DOM to a case study. In the next chapter, we'll build a JavaScript image gallery using the DOM methods introduced in this chapter.

CHAPTER 4

A JavaScript Image Gallery

- What this chapter covers:Beginning with a well-marked-up document
- Writing a JavaScript function to display a requested image
- Triggering the function from within the markup
- Expanding the function using some new methods

It's time to put the Document Object Model to work. This chapter will show you how to make an image gallery using JavaScript and the DOM.

There are a number of ways to put a gallery of images online. You could simply put all the images on one web page, for example. However, if you want to display more than a handful of images, the page is going to get big and bloated fairly quickly. The weight of the markup by itself might not be all that much, but the combined weight of the markup and images can result in a hefty download. Let's face it—nobody likes waiting a long time for pages to download.

A better solution might be to create a web page for each image. Then, instead of having one large page to download, you have lots of reasonably sized pages. But, at the same time, making all of those pages could be very time-consuming. Each page would probably need to have some kind of navigation to show the position of the current image in the gallery.

Using JavaScript, you can create an image gallery that offers the best of both worlds. You can put the navigation for the entire gallery on one page and then download each image as, and when, it's required.

The markup

For this example, I'm going to use a handful of snapshots I've taken with a digital camera. I've scaled them down to the web-friendly size of 400 pixels wide by 300 pixels tall. Feel free to use any images of your own to follow along with this exercise.

I'll start by creating a list of links that point to the images. I haven't arranged the images in any particular order, so I'm going to use an unordered list (`<ul>`) to do this. If your images are arranged sequentially, then an ordered list (`<ol>`) is probably the best way to mark them up.

Here's my document:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Image Gallery</title>
</head>
<body>
  <h1>Snapshots</h1>
  <ul>
```

```
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display"> Fireworks</a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee"> Coffee</a>
    </li>
    <li>
      <a href="images/rose.jpg" title="A red, red rose">Rose</a>
    </li>
    <li>
      <a href="images/bigben.jpg" title="The famous clock"> Big Ben</a>
    </li>
  </ul>
</body>
</html>
```

I'm going to save that file as `gallery.html` and place my pictures in a directory called `images`. My `images` directory is inside the same directory as `gallery.html`. Each of the links in the unordered list points to a different image. Clicking any of the links takes you to the image, but in order to get back to the list, you have to use the browser's back button. Here's an illustration of the bare-bones list of links.

This is a perfectly fine web page, but the default behavior isn't ideal. These are the things that should happen instead:

- When I click a link, I should remain on the same page.
- When I click a link, I should see the image on the same page that has my original list of images.

This is how I'm going to do that:

- Put a placeholder image on the same page as the list.
- When the user clicks a link, intercept the default behavior.
- Replace the placeholder image with the image from the link.

Let's start with the placeholder image. My example uses a kind of title card, but you can just as easily use a blank image.

Insert this line right after the list:

```
<img id="placeholder" src="images/placeholder.gif" alt="my image gallery" />
```

This adds an `id` attribute to this image, which I can use to style the image from an external style sheet. For instance, I might want to float the image so that it appears next to the link list instead of underneath it. I can also use this `id` in my JavaScript. An illustration of the placeholder image added to the page is shown next.

The markup is ready. Now it's time to stir in the JavaScript.

The JavaScript

In order to swap out the placeholder image, I need to change its `src` attribute. Luckily, the `setAttribute` method is perfect for this task. I'll write a function using this method that accepts a single argument: a link to an image. This function will then update the `src` attribute of the placeholder image with this image.

To start, I need a good name for my function. It should be descriptive, but not too verbose. I'm going to call it `showPic`. I also need a name for the argument that is passed to the function; I'll call it `whichpic`:

```
function showPic(whichpic)
```

Note that whichpic represents an element node. Specifically, it's an <a> element that leads to an image. I need to extract the path to the image, which I can do by using the getAttribute method on the whichpic element. By passing "href" as the argument, I can find out what the path to the image is:

```
whichpic.getAttribute("href")
```

I'll store this value in a variable to retrieve later. Let's call the variable source:

```
var source = whichpic.getAttribute("href");
```

The next thing I need to do is get to the placeholder image. This is easily done using getElementById:

```
document.getElementById("placeholder")
```

I then assign this element to a variable so I don't have to type out document.getElementById("placeholder") more than once. I'll call this variable placeholder:

```
var placeholder = document.getElementById("placeholder");
```

Now I've assigned values to two variables, source and placeholder. That will help keep the script readable.

I'll update the src attribute of the placeholder element using setAttribute. Remember, this method takes two arguments: the attribute you want to set, and the value you want this attribute to have. In this case, I want to set the src attribute so that its first argument is "src". The second argument, the value I want the src attribute to have, has already been stored as source:

```
placeholder.setAttribute("src",source);
```

That's a lot easier to read than this:

```
document.getElementById("placeholder").setAttribute("src",
➥ showpic.getAttribute("href"));
```

A DOM diversion

It's also possible to change the src attribute of an image without using setAttribute.

The setAttribute method is part of the DOM Level 1 specification, which allows you to set any attribute for any element node. Before DOM Level 1, it was still possible to set the attributes of many, though not all, elements, but it was done differently. You can still change attributes in this way, however.

For instance, to change the value attribute of an input element, you could use the value method:

```
element.value = "the new value"
```

That would have the same effect as this:

```
element.setAttribute("value","the new value");
```

Similarly, there's another way to change the source of an image. I could write my image gallery script to use this method instead of setAttribute:

```
placeholder.src = source;
```

Personally, I prefer to use setAttribute. For one thing, I don't have to remember which pre-DOM methods work on which elements. Though the old methods work fine on images, forms, and some other elements in a document, I can use setAttribute on any element in a document that I'd like.

Also, DOM Level 1 is more portable. While the older methods deal specifically with web documents, DOM methods can be used on any markup. Admittedly, that doesn't apply to what we're doing here, but it's worth bearing in mind. Remember, the DOM is an API that can be applied using many programming languages in many environments. If you ever need to apply the DOM skills you're learning now to situations outside the browser, sticking with DOM Level 1 will avoid any compatibility issues.

Finishing the function

Here's what the finished function looks like:

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
}
```

Now it's time to hook up the JavaScript with the markup.

Applying the JavaScript

I need to make sure that my newly written `showPic` function is available to my image gallery document. Save the function in a text file with the extension `.js`. In this case let's call it `showPic.js`.

> *When you have multiple JavaScript files for a site you should combine them into a single file to reduce the number of requests on your published web site. In many of the examples we'll use multiple files so that it's easier to explain but in chapter 5 "Best Practices" we'll discuss this as well as other performance improvements.*

Just as I've placed all of this example's pictures in a directory called `images`, it's a good idea to keep all your JavaScript files in one place. Create a directory called `scripts` and save the `showPic.js` file there.

Now I need to point the image gallery document to the JavaScript file. Add this line to the HTML document (right before the closing `</body>` tag, for instance):

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

The `showPic` function is now available to the image gallery document. As it stands, the function will never be invoked. I need to add the behavior to the links in my list; I'll do that by adding an *event handler*.

Event handlers

Event handlers are used to invoke some JavaScript when a certain action happens. If you want some behavior to be triggered when the user moves their cursor over an element, you use the `onmouseover` event handler. There's a corresponding `onmouseout` event. For my image gallery, I want to add a behavior when the user clicks a link. The event handler for this is `onclick`.

Also note that I can't invoke the `showPic` function without sending it some information. `showPic` expects one argument: an element node that has an `href` attribute. When I place the `onclick` event handler in a link, I want to send that link to the `showPic` function as the argument.

Luckily, there's a very short but powerful way of doing just that. The keyword `this` is a shorthand way of saying "this object." In this case, I'll use `this` to mean "this `<a>` element node":

```
showPic(this)
```

I can add that using the `onclick` event handler. Here's the syntax for adding JavaScript using an event handler:

```
event = "JavaScript statement(s)"
```

Notice that the JavaScript itself is contained within quotes. You can put as many JavaScript statements as you like between the quotes, as long as they are separated with semicolons.

This will invoke the `showPic` function with the onclick event handler:

```
onclick = "showPic(this);"
```

However, if I simply add this event handler to a link in my list, I'll be faced with a problem. The function will be invoked, but the default behavior for clicking on a link will also be invoked. This means that the user will be taken to the image—exactly what we didn't want to happen. I need to stop the default behavior from being invoked.

Let's take a closer look at how event handling works. When you attach an event handler to an element, you can trigger JavaScript statements with the event. The JavaScript can return a result that is then passed back to the event handler. For example, you can attach some JavaScript to the `onclick` event of a link so that it returns a Boolean value of true or false. If you click the link, and the event handler receives a value of true, it's getting the message "yes, this link has been clicked." If you add some JavaScript to the event handler so that it returns false, then the message being sent back is "no, this link has not been clicked."

You can see this for yourself with this simple test:

```
<a href="http://www.example.com" onclick="return false;">Click me</a>
```

If you click that link, the default behavior will not be triggered, because the JavaScript is effectively canceling the default behavior.

By adding a `return false` statement to the JavaScript contained by the `onclick` event handler, I can stop the user from being taken straight to the destination of the link:

```
onclick = "showPic(this); return false;"
```

This is how it would look in the context of the document:

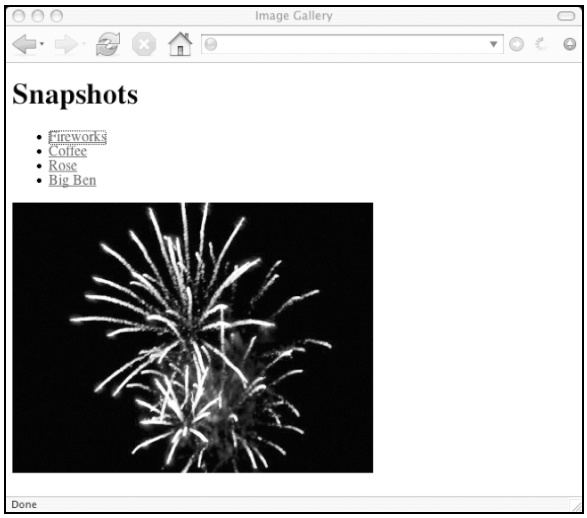
```
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥ return false;" title="A fireworks display">Fireworks</a>
</li>
```

Now I'll need to add that event handler to every link in my list. That's a tedious task, but in Chapter 6 under "Adding the event handler" you'll see a way to avoid it completely. For now, I'll dive into the markup and add the event handler by hand:

```
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥ return false;" title="A fireworks display">Fireworks</a>
  </li>
  <li>
    <a href="images/coffee.jpg" onclick="showPic(this);
➥ return false;" title="A cup of black coffee">Coffee</a>
  </li>
  <li>
    <a href="images/rose.jpg" onclick="showPic(this); return false;"
➥ title="A red, red rose">Rose</a>
  </li>
  <li>
```

```
    <a href="images/bigben.jpg" onclick="showPic(this); return false;"
➥ title="The famous clock">Big Ben</a>
  </li>
```

If you load the page in a web browser, you will see a working JavaScript image gallery. Click on a link in the list and you will see the image displayed on the same page.



Expanding the function

Swapping out images on a web page isn't really all that impressive. You could do it even prior to WC3 DOM JavaScript. It was the basis for all those rollover scripts that have proven so popular.

What would be really great, you're thinking, would be the ability to update the text on a web page. Using JavaScript and the DOM, you can do just that.

Every link in my list has a `title` attribute. I'd like to take the value of that attribute and display it on the page along with the image. I can get the value of the `title` attribute easily enough using `getAttribute`:

```
var text = whichpic.getAttribute("title");
```

Just retrieving the text isn't going to do much; I also need to insert it into the document. To do that, I need to introduce some new DOM properties.

Introducing childNodes

The `childNodes` property is a way of getting information about the children of any element in a document's node tree. Itreturns an array containing all the children of an element node:

`element.childNodes`

Let's say you wanted to get all the children of the `body` element. We can use `getElementsByTagName` to get at the `body` element. We know that there is only one body element, so it will be the first (and only) element of the array `getElementsByTagName("body")`:

```
var body_element = document.getElementsByTagName("body")[0];
```

Now the variable `body_element` refers to the `body` element in our document. To access the children of the body element, you just need to use

```
body_element.childNodes
```

That's a lot easier than writing this:

```
document.getElementsByTagName("body")[0].childNodes
```

Now that you can get at all the children of the `body` element, let's take a look at what you can do with that information.

First, you can find out exactly how many children the `body` element has. Because `childNodes` returns an array, you can use the `length` property to find out how many elements it contains:

```
body_element.childNodes.length;
```

Try adding this little function to the `showPic.js` file:

```
function countBodyChildren() {
  var body_element = document.getElementsByTagName("body")[0];
  alert (body_element.childNodes.length);
}
```

This simple function will pop up an alert dialog with the total number of the `body` element's children.

You'll want this function to be executed when the page loads, and you can use the `onload` event handler to do this. Add this line to the end of the code:

```
window.onload = countBodyChildren;
```

When the document loads, the `countBodyChildren` function will be invoked.

Refresh the `gallery.html` file in your web browser. You will be greeted with an alert dialog containing the total number of children of the `body` element. The result may surprise you.

Introducing the nodeType property

Looking at the structure of the `gallery.html` file, it would appear that the `body` element has just three children: the `h1` element, the `ul` element, and the `img` element. Yet, when we invoke the `countBodyChildren` function, we get a much higher figure. This is because elements are just one type of node.

The `childNodes` property returns an array containing all types of nodes, not just element nodes. It will bring back all the attribute nodes and text nodes as well. In fact, just about everything in a document is some kind of node. Even spaces and line breaks are interpreted as nodes and are included in the `childNodes` array.

That explains why the result produced by `countBodyChildren` is so high.

Fortunately, we can use the `nodeType` property on any node in a document. This will tell us exactly what kind of node we're dealing with. Unfortunately, it won't tell us in plain English.

The `nodeType` property is called with the following syntax:

```
node.nodeType
```

However, instead of returning a string like "element" or "attribute," it returns a number.

Change the `alert` statement in the `countBodyChildren` function so that it now gives us the `nodeType` of `body_element`:

```
alert(body_element.nodeType);
```

Refresh the browser window that's displaying gallery.html. Now you'll see an alert dialog containing the number 1. Element nodes have a `nodeType` value of 1.

There are twelve possible values for `nodeType`, but only three of them are going to be of much practical use:

- Element nodes have a `nodeType` value of 1.
- Attribute nodes have a `nodeType` value of 2.
- Text nodes have a `nodeType` value of 3.

This means that you can target specific types of nodes in your functions. For instance, you could create a function that only affects element nodes.

Adding a description in the markup

To improve the image gallery function, I want to manipulate a text node. I want to replace its value with a value taken from an attribute node (the `title` attribute in a link).

First, I need to have somewhere to put the text. I'm going to add a new paragraph to `gallery.html`. I'll place it right after the `img` tag. I'm going to give it a unique `id` so that I can reference it easily from the `JavaScript function`:

```
<p id="description">Choose an image.</p>
```

I've given the `id` attribute the value "description", which describes its role fairly accurately. For now, it contains the text "Choose an image." Here, you can see that a new paragraph has been added.

I plan to replace this text with text taken from a link's `title` attribute. I want this to happen at the same time as the placeholder image is replaced with the image taken from the link's `href` attribute. To achieve this, I need to update the `showPic` function.

Changing the description with JavaScript

I'm going to update the `showPic` function so that the text in the description paragraph is replaced with the text from a `title` attribute in a link.

This is how the `showPic` function looks right now:

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
}
```

I'm going to begin my improvements by getting the value of the `title` attribute of `whichpic`. I'll store this value in a variable called `text`. This is easily done using `getAttribute`:

```
var text = whichpic.getAttribute("title");
```

Now I want to create a new variable so that I have an easy way of referencing the paragraph with the `id` "description". I'll call this variable `description` also:

```
var description = document.getElementById("description");
```

I have my variables:

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
}
```

Now it's time to do the text swapping.

Introducing the nodeValue property

If you want to change the value of a text node, there is a DOM property called `nodeValue` that can be used to get (and set) the value of a node:

```
node.nodeValue
```

Here's a tricky little point. If you retrieve the `nodeValue` for `description`, you *won't* get the text within the paragraph. You can test this with an `alert` statement:

```
alert (description.nodeValue);
```

This will return a value of null. The `nodeValue` of the paragraph element itself is empty. What you actually want is the value of the text within the paragraph.

The text within the paragraph is a different node. This text is the first child node of the paragraph. Therefore, you want to retrieve the `nodeValue` of this child node.

This `alert` statement will give you the value you're looking for:

```
alert(description.childNodes[0].nodeValue);
```

This will return a value of "Choose an image." This means that you're accessing the `childNodes` array and getting the value of the first element (index number zero).

Introducing firstChild and lastChild

There is a shorthand way of writing `childNodes[0]`. Whenever you want to get the value of the first node in the `childNodes` array, you can use `firstChild`:

node`.firstChild`

This is equivalent to

node`.childNodes[0]`

This is a handy shortcut and it's also a lot easier to read.
The DOM also provides a corresponding `lastChild` property:

node`.lastChild`

This refers to the last node in the `childNodes` array. If you wanted to access this node without using the `lastChild` property, you would have to write

node`.childNodes[`*node*`.childNodes.length-1]`

That's clearly very unwieldy. It's a lot easier to simply use `lastChild`.

Using nodeValue to update the description

Now we'll go back to the `showPic` function. I'm going to update the `nodeValue` of the text node within the `description` paragraph.

In the case of the `description` paragraph, only one child node exists. I can use either `description.firstChild` or `description.lastChild`. I'm going to use `firstChild` in this case.

I can rewrite my `alert` statement so that it now reads

```
alert(description.firstChild.nodeValue);
```

The value is the same ("Choose an image"), but now the code is more readable.

The `nodeValue` method is very versatile. It can be used to retrieve the value of a node, but it can also be used to set the value of a node. That's exactly what I want to do in this case.

If you recall, I've already set aside a string in the variable `text`, which I retrieved from the `title` attribute of the link that has been clicked. I'm now going to update the value of the first child node of the `description` paragraph:

```
description.firstChild.nodeValue = text;
```

These are the three new lines that I've added to `showPic` function:

```
var text = whichpic.getAttribute("title");
var description = document.getElementById("description");
description.firstChild.nodeValue = text;
```

In plain English, I'm saying:

- Get the value of the `title` attribute of the link that has just been clicked and store this value in a variable called `text`.
- Get the element with the `id` "description" and store this object as the variable `description`.
- Update the value of the first child node of the `description` object with the value of `text`.

This is how the final function looks:

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
}
```

If you update the `showPic.js` file with these new lines and then refresh `gallery.html` in your browser, you can test the expanded functionality. Clicking a link to an image now produces two results. The `placeholder` image is replaced with the new image, and the `description` text is replaced with the `title` text from the link.

You can find my image gallery script and markup at `http://friendsofed.com/`. All of my images are there, too, but if you want to have some fun with this script, try using your own pictures.

If you want to liven up the image gallery, you can add a style sheet like this one:

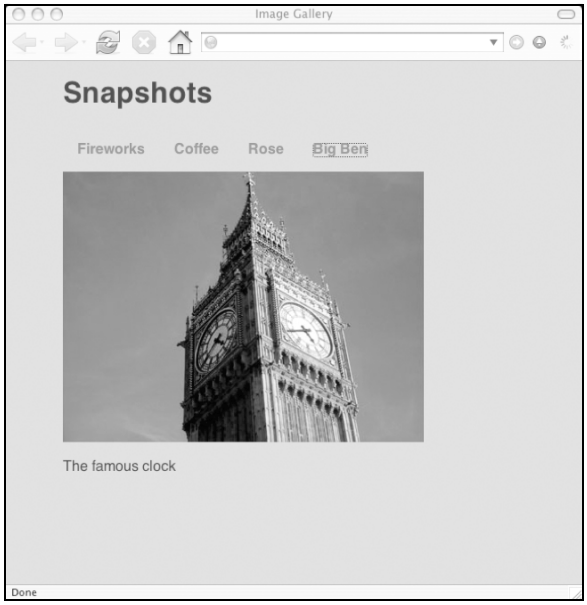
```
body {
  font-family: "Helvetica","Arial",serif;
  color: #333;
  background-color: #ccc;
  margin: 1em 10%;
}
h1 {
  color: #333;
  background-color: transparent;
}
a {
  color: #c60;
```

```
  background-color: transparent;
  font-weight: bold;
  text-decoration: none;
}
ul {
  padding: 0;
}
li {
  float: left;
  padding: 1em;
  list-style: none;
}
img {
  display:block;
  clear:both;
}
```

You can save that CSS in a file called `layout.css` in a directory called `styles`. You can then reference this file from a `<link>` tag in the `<head>` of `gallery.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Image Gallery</title>
  <link rel="stylesheet" href="styles/layout.css" media="screen" />
</head>
<body>
  <h1>Snapshots</h1>
  <ul>
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display"
➥ onclick="showPic(this); return false;">Fireworks</a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee"
➥ onclick="showPic(this); return false;">Coffee</a>
    </li>
    <li>
      <a href="images/rose.jpg" title="A red, red rose"
➥ onclick="showPic(this); return false;">Rose</a>
    </li>
    <li>
      <a href="images/bigben.jpg" title="The famous clock"
➥ onclick="showPic(this); return false;">Big Ben</a>
    </li>
  </ul>
  <img id="placeholder" src="images/placeholder.gif" alt="my image gallery" />
  <p id="description">Choose an image.</p>
  <script src="scripts/showPic.js"></script>
</body>
</html>
```

Following is an example of the image gallery with a simple style sheet attached.



What's next?

In this chapter, you've seen some applied JavaScript. You've also been introduced to some new DOM properties, such as

- `childNodes`
- `nodeType`
- `nodeValue`
- `firstChild`
- `lastChild`

You've learned how to put together an image gallery script using some of the methods offered by the Document Object Model. You've also learned how to integrate JavaScript into your web pages using event handlers.

On the surface, this JavaScript image gallery probably appears to be a complete success. However, there's actually quite a lot of room for improvement, which I'll be covering shortly.

The next chapter covers the best practices that should be used when writing JavaScript. You'll see that how you achieve a final result is as important as the result itself.

Then, in Chapter 6, you'll see how to apply those best practices to the image gallery script.