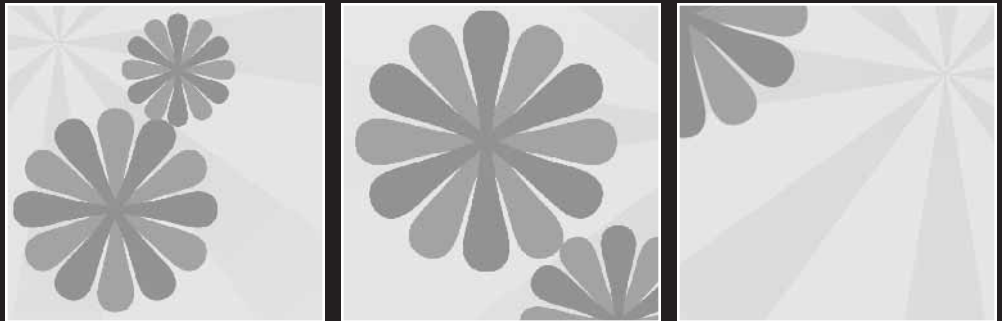


5 BEST PRACTICES



What this chapter covers:

- Graceful degradation: ensuring that your web pages still work without JavaScript.
- Unobtrusive JavaScript: separating structure from behavior.
- Backwards compatibility: ensuring that older browsers don't choke on your scripts.

Together, JavaScript and the Document Object Model form a powerful combination. It's important that you use this power wisely. In this chapter, I'm going to show you some best practices that you can use to ensure that your scripts don't do more harm than good.

Please don't let me be misunderstood

JavaScript has something of a bad reputation. Right from its inception, seasoned programmers treated it with skepticism. The convoluted origin of the name JavaScript certainly did not help (see Chapter 1). With a name like JavaScript, it was inevitably compared to the Java programming language and found lacking.

Programmers who were expecting to find the power of Java were disappointed by the simplicity of the first iterations of JavaScript. Despite a superficial lexical similarity, JavaScript was a much simpler language. Most importantly, where Java is very object-oriented, JavaScript was usually written in a procedural way.

JavaScript can also be used in an object-oriented fashion, but the stigma remains. Many programmers turn their noses up at JavaScript for being "just a scripting language."

JavaScript also received a less than enthusiastic welcome from web designers for just the opposite reasons. Whereas programmers bemoaned the language's lack of complexity, designers were intimidated by the thought of having to learn how to write code.

Designers who made quick progress in learning HTML would have been brought up short by objects, functions, arrays, variables, and the other building blocks of JavaScript. The fact that programmers were insisting that JavaScript was laughably simple probably increased the frustration felt by web designers trying to grapple with the language.

Nonetheless, the fact that JavaScript was effectively the only way of adding real-time interactivity in web pages ensured that it would be widely adopted. Unfortunately, in many cases, the way that JavaScript was implemented only served to increase the scorn and resentment designers and programmers felt for the language.

Don't blame the messenger

A low barrier to entry can be a double-edged sword. A technology that people can use speedily and easily will probably be adopted very quickly. However, there is likely to be a correspondingly low level of quality control.

HTML's ease of use is one of the reasons behind the explosive growth of the World Wide Web. Anybody can learn the basics of HTML in a short space of time and create a web page very quickly. It's even possible to use WYSIWYG editors (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et) to make web pages without ever seeing a line of markup.

The downside to this is that most pages on the Web are badly formed and don't validate. Browser manufacturers have to accept this state of affairs by making their software very forgiving and unfussy. Much of the code in browser software is dedicated to handling ambiguous use of HTML and trying to second-guess how authors want their web pages to be rendered.

In theory, there are billions of HTML documents on the Web. In practice, only a small fraction of those documents are made of well-formed, valid markup. This legacy makes the process of advancing web technologies like XHTML and CSS much more difficult.

HTML's low barrier to entry has been a mixed blessing for the World Wide Web.

The situation with JavaScript isn't quite as drastic. If JavaScript code isn't written with the correct syntax, the coder will be alerted because it will cause an error (as opposed to HTML, which, in most cases, will render anyway).

Nonetheless, there is a lot of very bad JavaScript out there on the Web.

Many web designers wanted to reap the benefits of using JavaScript for adding spice to their web pages without spending the time to learn the language. It wasn't long before WYSIWYG editors began offering snippets of code that could be attached to documents.

Even without a WYSIWYG editor, it was still quite easy to find snippets of JavaScript. Many websites and books appeared offering self-contained functions that could be easily added to web pages. Cutting and pasting was the order of the day.

Unfortunately, many of these functions weren't very well thought out. On the surface, they accomplished their tasks and added extra interactivity to web pages. In most cases, however, little thought was given to how these pages behaved when JavaScript was disabled. Poorly implemented scripts sometimes had the effect of inadvertently making JavaScript a requirement for navigating a website. From an accessibility viewpoint, this was clearly a problem. It wasn't long before the words "JavaScript" and "inaccessible" became linked in many people's minds.

In fact, there's nothing inherently inaccessible about JavaScript. It all depends on how it's used. In the words of the old song, "it ain't what you do, it's the way that you do it."

The Flash mob

In truth, there are no bad technologies. There are just bad uses of a technology. The cruel hand that fate had dealt JavaScript reminds me of another much maligned technology: Macromedia's Flash.

Many people today associate Flash with annoying splash pages, overlong download times, and unintuitive navigation. None of these things are intrinsic to Flash. They're simply by-products of poorly implemented Flash movies.

It's ironic that Flash has become associated with overlong download times. One of the great strengths of Flash is its ability to create light, compressed vector images and movies. But once it became the norm for Flash sites to have bloated splash intro movies, the trend was hard to reverse.

Similarly, JavaScript can and should be used to make web pages more usable. Yet it has a reputation for decreasing the usability and accessibility of websites.

The problem is one of both inertia and momentum. If a technology is used in a thoughtless way from very early on and that technology is then quickly adopted, it becomes very hard to change those initial bad habits.

I'm sure that all the pointless Flash intro movies on splash pages didn't spring up simultaneously. Instead, it was a case of "monkey see, monkey do." People wanted splash pages because other people had splash pages. Nobody stopped to ask why it was necessary.

JavaScript suffered a similar fate. Badly written functions, particularly those written by WYSIWYG editors, have shown a remarkable tenacity. The code has been copied and pasted. The functions have spread far and wide across the Web, but nobody has ever questioned whether they could have been better.

Question everything

Whenever you use JavaScript to alter the behavior of your web pages, you should question your decisions. First and foremost, you should question whether the extra behavior is really necessary.

JavaScript has been misused in the past to add all sorts of pointless bells and whistles to websites. You can use scripts that move the position of the browser window or even cause the browser window to shake.

Most notoriously of all, there are scripts that cause ad-containing pop-up windows to appear when a page loads. The really nefarious scripts cause these newly spawned windows to appear *under* the current browser window. These pop-under ads further cemented JavaScript's reputation as being user-unfriendly. Some people dealt with the problem by disabling JavaScript entirely. The browser manufacturers themselves took steps to deal with the problem by offering built-in pop-up blockers.

Pop-under windows are the epitome of JavaScript abuse. Ostensibly, they are supposed to solve a problem: how to deliver advertising to users. In practice, they only increase user frustration. The onus was on the user to close these windows, something that often turned into an unhappy game of whack-a-mole.

If only someone had asked, "How will this benefit the user?"

Fortunately, this question is being asked more often these days. User-centric web design is on the increase. Any other approach is, in the long term, doomed to failure.

Whenever you are using JavaScript, you should ask yourself how it will affect the user's experience. There's another very important question you should also ask yourself: what if the user doesn't have JavaScript?

Graceful degradation

It's always worth remembering that a visitor to your site might be using a browser that doesn't have JavaScript support. Or maybe the browser supports JavaScript but it has been disabled (perhaps after being exposed to one too many pop-under windows). If you don't consider this possibility, you could inadvertently stop visitors from using your site.

If you use JavaScript correctly, your site will still be navigable by users who don't have JavaScript. This is called **graceful degradation**. When a technology degrades gracefully, the functionality may be reduced but it doesn't fail completely.

Take the example of opening a link in a new window. Don't worry: I'm not talking about spawning a new window when the page loads. I'm talking about creating a pop-up window when the user clicks on a link.

This can be quite a useful feature. Many e-commerce websites will have links to terms of service or delivery rates from within the checkout process. Rather than having the user leave the current screen, a pop-up window can display the relevant information without interrupting the shopping experience.

Popping up a new window should only be used when it's absolutely required. There are accessibility issues involved: for example, some screen-reading software doesn't indicate that a new window has been opened. It's a good idea to make it clear in the link text that a new browser window will be opened.

JavaScript uses the `open()` method of the `window` object to create new browser windows. The method takes three arguments:

```
window.open(url, name, features)
```

All of the arguments are optional. The first argument is the URL for the document you want to open in a new window. If this is missing, an empty browser window will be created.

The second argument is the name that you can give this newly created window. You can use this name in your code to communicate with your newly created window.

The final argument accepted by the method is a comma-separated list of features that you want your new window to have. These include the size of the window (width and height) and aspects of the browser chrome that you want to enable or disable (including toolbars, menu bar, location, and so on). You don't have to list all of the features, and anyway, it's a good idea not to disable too many features.

This method is a good example of using the Browser Object Model (BOM). Nothing about the functionality affects the contents of the document (that's what the Document Object Model is for). The method is purely concerned with the browsing environment (in this case, the `window` object).

Here's an example of a typical function that uses `window.open()`:

```
function popUp(winURL) {  
    window.open(winURL, "popup", "width=320,height=480");  
}
```

This will open up a new window (called "popup") that's 320 pixels wide by 480 pixels tall. Because I'm setting the name of the window in the function, any time a new URL is passed to the function, the function will replace the existing spawned window rather than creating a second one.

I can save this function in an external file and link to it from a `<script>` tag within the `<head>` of a web page. By itself, this function doesn't have any usability implications. What matters is how I use it.

The javascript: pseudo-protocol

One way of calling the `popUp` function is to use what's known as a **pseudo-protocol**.

Real protocols are used to send packets of information between computers on the Internet. Examples are `http://`, `ftp://`, and so on. A pseudo-protocol is a non-standard take on this idea. The **javascript:** pseudo-protocol is supposed to be used to invoke JavaScript from within a link.

Here's how the `javascript:` pseudo-protocol would be used to call the `popUp` function:

```
<a href="javascript:popUp('http://www.example.com/')">Example</a>
```

This will work just fine in browsers that understand the `javascript:` pseudo-protocol. Older browsers, however, will attempt to follow the link and fail. Even in browsers that understand the pseudo-protocol, the link becomes useless if JavaScript has been disabled.

In short, using the `javascript:` pseudo-protocol is usually a very bad way of referencing JavaScript from within your markup.

Inline event handlers

You've already seen event handlers in action with the image gallery script. The image swapping function was invoked from within an `<a>` tag by adding the `onClick` event handler as an attribute of the same tag.

The same technique will work for the `popUp` function. If you use an `onClick` event handler from within a link to spawn a new window, then the `href` attribute might seem irrelevant.

After all, all the important information about where the link leads is now contained in the `onClick` attribute. That's why you'll often see links like this:

```
<a href="#" onclick="popUp('http://www.example.com/');
➔ return false;">Example<a>
```

The `return false` has been included so that the link isn't actually followed. The `#` symbol is used for internal links within documents. (In this case, it's an internal link to nowhere.) In some browsers, this will simply lead to the top of the current document. Using a value of `#` for the `href` attribute is an attempt to create a blank link. The real work is done by the `onClick` attribute.

This technique is just as bad as using the `javascript: pseudo-protocol`. It doesn't degrade gracefully. If JavaScript is disabled, the link is useless.

Who cares?

5

You might be wondering about this theoretical situation I keep mentioning. Is it really worth making sure that your site works for this kind of user?

Imagine a visitor to your website who browses the Web with both images and JavaScript disabled. You might imagine that this visitor is very much in the minority, and you'd be right. But this visitor is important.

The visitor that you've just imagined is a **searchbot**. A searchbot is an automated program that spiders the web in order to add pages to a search engine's index. All the major search engines have programs like this. Right now, very few searchbots understand JavaScript.

If your JavaScript doesn't degrade gracefully, your search engine rankings might be seriously damaged.

In the case of the `popUp` function, it's relatively easy to ensure that the JavaScript degrades gracefully. As long as there is a real URL in the `href` attribute, then the link can be followed:

```
<a href="http://www.example.com/"
➔ onclick="popUp('http://www.example.com'); return false;">Example</a>
```

That's quite long-winded bit of code because the URL appears twice. Fortunately, there's a shortcut that can be used within the JavaScript. The word `this` can be used to refer to the current element. Using a combination of `this` and `getAttribute`, you can extract the value of the `href` attribute:

```
<a href="http://www.example.com/"
➔ onclick="popUp(this.getAttribute('href')); return false;">Example</a>
```

Actually, that doesn't save all that much space. There's an even shorter way of referencing the `href` of the current link by using the pre-standardized DOM property, `this.href`:

```
<a href="http://www.example.com/"
➔ onclick="popUp(this.href; return false;">Example</a>
```

In either case, the important point is that the value of the href attribute is now a valid one. This is far better than using either href="javascript:..." or href="#".

So you see, if JavaScript isn't enabled (or if the visitor is a searchbot), the link can still be followed if you've used a real URL in the href attribute. The functionality is reduced (because the link doesn't open in a new window), but it doesn't fail completely. This is a classic example of graceful degradation.

This technique is certainly the best one I've covered so far, but it is not without its problems. The most obvious problem is the need to insert JavaScript into the markup whenever you want to open a window. It would be far better if all the JavaScript, including the event handlers, were contained in an external file.

The lessons of CSS

Earlier I referred to both JavaScript and Flash as examples of technologies that were often implemented badly in the wild, anarchic days when they were first unleashed. We can learn a lot from the mistakes of the past.

There are other technologies that were implemented in a thoughtful, considered manner right from their inception. We can learn even more from these.

Cascading Style Sheets are a wonderful technology. They allow for great control over every aspect of a site's design. Ostensibly, there's nothing new about this. It's always been possible to dictate a design using <table> and tags. The great advantage of CSS is that you can separate the structure of a web document (the markup) from the design (the CSS).

It's entirely possible to use CSS in an inefficient manner. You could just add style attributes in almost every element of your web document, for example. But the real benefits of CSS become apparent when all your styles are contained in external files.

CSS arrived on the scene much later than Flash and JavaScript. Perhaps as a result of the trauma caused by badly implemented Flash and JavaScript, web designers used CSS in a thoughtful, constructive way from day one.

The separation of structure and style makes life easier for everyone. If your job is writing content, you no longer have to worry about messing up the design. Instead of swimming through a tag soup of <table> and tags, you can now concentrate on marking up your content correctly. If your job is creating the design for a site, you can now concentrate on controlling colors, fonts, and positioning using external style sheets without touching the content. At most, you'll need to add the occasional class or id attribute.

A great advantage of this separation of structure and style is that it guarantees graceful degradation. Browsers that are capable of interpreting CSS will display the web pages in all their styled glory. Older browsers, or browsers with CSS disabled, will still be able to view the content of the pages, in a correctly structured way. The Googlebot doesn't understand CSS, but it has no problems navigating around sites that use CSS for layout.

When it comes to applying JavaScript, we can learn a lot from CSS.

Progressive enhancement

“Content is king” is an oft-used adage in web design. It’s true. Without any content, there’s little point in trying to create a website.

That said, you can’t simply put your content online without first describing what it is. The content needs to be wrapped up in a markup language like HTML or XHTML. Marking up content correctly is the first and perhaps the most important step in creating a website. A revised version of the web design adage might be “well-marked-up content is king.”

When a markup language is used correctly, it describes the content semantically. The markup provides information such as “this is an item in a list” or “this is a paragraph.” They are all pieces of content, but tags like `` and `<p>` distinguish them.

Once the content has been marked up, you can dictate how the content should look by using CSS. The instructions in the CSS form a presentation layer. This layer can then be draped over the structure of the markup. If the presentation layer is removed, the content is still accessible (although now it’s a king with no clothes).

Applying layers of extra information like this is called **progressive enhancement**. Web pages that are built using progressive enhancement will almost certainly degrade gracefully.

Like CSS, all the functionality provided by JavaScript and the DOM should be treated as an extra layer of instructions. Where CSS contain information about presentation, JavaScript code contains information about behavior. Ideally, this behavior layer should be applied in the same way as the presentation layer.

CSS work best when they are contained in external files, separate from the markup. It’s entirely possible to use CSS in an inefficient manner and mix them in with the markup, like this:

```
<p style="font-weight: bold; color: red;">
  Be careful!
</p>
```

It makes more sense to keep the style information in an external file that can be called via a `<link>` tag in the head of the document:

```
.warning {
  font-weight: bold;
  color: red;
}
```

The `class` attribute can then be used as a hook to tie the style to the markup:

```
<p class="warning">
  Be careful!
</p>
```

This is far more readable. It’s also a lot easier to change the styles. Imagine you had a hundred documents with the `warning` class peppered throughout. Now suppose you wanted to change how warnings are displayed. Maybe you’d prefer them to be blue instead of red.

As long as your presentation is separated from your structure, you can change the style easily:

```
.warning {  
  font-weight: bold;  
  color: blue;  
}
```

If your style declarations were intermingled with your markup, you would have to do a lot of searching and replacing.

It's clear that CSS work best when they are unobtrusive. What works for the presentation layer will also work for the behavior layer.

Unobtrusive JavaScript

The JavaScript you've seen so far has already been separated from the markup to a certain degree. The functions that do all the work are contained in external files. The problem lies with inline event handlers.

Using an attribute like `onclick` in the markup is just as inefficient as using the `style` attribute. It would be much better if we could use a hook, like `class` or `id`, to tether the behavior to the markup without intermingling it. This is how the markup could indicate that a link should have the `popUp` function performed when it is clicked:

```
<a href="http://www.example.com/" class="popup">Example</a>
```

Fortunately, this is entirely possible. Events don't have to be handled in the markup. You can attach an event to an element in an external JavaScript file:

```
element.event = action...
```

The tricky part is figuring out which elements should have the event attached. That's where hooks like `class` and `id` come in handy.

If you want to attach an event to an element with a unique `id`, you can simply use `getElementById`:

```
getElementById(id).event = action
```

With multiple elements, you can use a combination of `getElementsByName` and `getAttribute` to attach events to elements with specific attributes.

Here's the plan in plain English:

1. Make an array of all the links in the document.
2. Loop through this array.
3. If a link has the class "popup", execute this behavior when the link is clicked:
 - A. Pass the value of the link's `href` attribute to the `popUp` function.
 - B. Cancel the default behavior so that the link isn't followed in the original window.

This is how it looks in JavaScript:

```
var links = document.getElementsByTagName("a");
for (var i=0; i<links.length; i++) {
  if (links[i].getAttribute("class") == "popup") {
    links[i].onclick = function() {
      popUp(this.getAttribute("href"));
      return false;
    }
  }
}
```

Now the connection between the links and the behavior that should occur when the links are clicked has been moved out of the markup and into the external JavaScript. This is unobtrusive JavaScript.

There's just one problem. If I put that code in my external JavaScript file, it won't work. The first line reads:

```
var links = document.getElementsByTagName("a");
```

This code will be executed as soon as the JavaScript file loads. The JavaScript file is called from a `<script>` tag in the `<head>` of my document and the JavaScript file will load before the document. Because the document is incomplete, the model of the document is also incomplete. Without a complete Document Object Model, methods like `getElementsByTagName` simply won't work.

I need to execute the code once the document has finished loading. Fortunately, the complete loading of a document is an event with a corresponding event handler.

The document loads within the browser window. The document object is a property of the window object. When the `onload` event is triggered by the window object, the document object then exists.

I'm going to wrap up my JavaScript inside a function called `prepareLinks`, and I'm going to attach this function to the `onload` event of the window object. This way I know that the Document Object Model will be working:

```
window.onload = prepareLinks;
function prepareLinks() {
  var links = document.getElementsByTagName("a");
  for (var i=0; i<links.length; i++) {
    if (links[i].getAttribute("class") == "popup") {
      links[i].onclick = function() {
        popUp(this.getAttribute("href"));
        return false;
      }
    }
  }
}
```

Don't forget to include the `popUp` function as well:

```
function popUp(winURL) {
    window.open(winURL, "popup", "width=320,height=480");
}
```

This is a very simple example, but it demonstrates how behavior can be successfully separated from structure. Later on, I'll show you more elegant ways to attach events when the document loads.

Backwards compatibility

As I keep mentioning, it's important to consider that visitors to your website might not have JavaScript enabled. However, there are degrees of JavaScript support.

Most browsers support JavaScript to some degree, and most modern browsers have excellent support for the Document Object Model. Older browsers, however, might not be able to understand DOM methods and properties.

So even if a user visits your site with a browser that supports some JavaScript, some of your scripts may not work.

The simplest solution to this problem is to quiz the browser on its level of JavaScript support. This is a bit like those signs at amusement parks that read, "You must be this tall to ride." The DOM Scripting equivalent would be, "You must understand this much JavaScript to execute these statements."

This is quite easy to accomplish. If you wrap a method in an `if` statement, the statement will evaluate to either true or false, depending on whether the method exists. This is called **object detection**. Methods, like almost everything else in JavaScript, can be treated as objects. This makes it quite easy to exclude browsers that don't support a specific DOM method:

```
if (method) {
    statements
}
```

If I have a function that uses `getElementById`, I can test whether or not that method is supported before attempting to use it. The method won't actually be executed so there's no need for the usual parentheses. I'm just testing to see whether it exists or not:

```
function myFunction() {
    if (document.getElementById) {
        statements using getElementById
    }
}
```

If a browser doesn't understand `getElementById`, it will never even get to the statements using that method.

The only disadvantage in the way I've written that function is that it adds another set of curly braces. If I do that every time I want to test for a particular method or property, then I'm going to end up with the most important statements being wrapped in layers and layers of curly braces. That won't be much fun to read.

It would be much more convenient to say, "If you don't understand this method, leave now."

To turn "if you do understand" into "if you don't understand", all that's needed is the "not" operator, represented by an exclamation point:

```
if (!method)
```

You can use `return` to achieve the "leave now" part. Seeing as the function is ending prematurely, it makes sense that the Boolean value being returned is `false`. This is how it would look in a test for `getElementById`:

```
if (!getElementById) {
  return false;
}
```

Because just one statement needs to be executed (`return false`), you can shorten the test even further by putting it on one line:

```
if (!getElementById) return false;
```

If you need to test for the existence of more than one method or property, you can join them together using the "or" logical operator, represented by two vertical pipe symbols:

```
if (!getElementById || !getElementsByName) return false;
```

If this were a sign in an amusement park, it would read, "If you don't understand `getElementById` or `getElementsByName`, you can't ride."

I can put this into practice with my page load script that attaches the `onclick` event to certain links. It uses `getElementsByName`, so I want to be sure that the browser understands that method:

```
window.onload = function() {
  if (!document.getElementsByName) return false;
  var lnks = document.getElementsByName("a");
  for (var i=0; i<lnks.length; i++) {
    if (lnks[i].getAttribute("class") == "popup") {
      lnks[i].onclick = function() {
        popUp(this.getAttribute("href"));
        return false;
      }
    }
  }
}
```

By just adding this one line, I can be sure that older browsers won't choke on my code. This is assuring backwards compatibility. Because I've used progressive enhancement to add behavior to my web page, I can be sure that the functionality will degrade gracefully in older browsers. Browsers that understand some JavaScript, but not the DOM, can still access my content.

Browser sniffing

Testing for the existence of a specific property or method that you're about to use in your code is the safest and surest way of ensuring backwards compatibility. There is another technique that was very popular during the dark days of the browser wars.

Browser sniffing involves extracting information provided by the browser vendor. In theory, browsers supply information (readable by JavaScript) about their make and model. You can attempt to achieve backwards compatibility by parsing this information, but it's a very risky technique.

For one thing, the browsers sometimes lie. For historical reasons, some browsers report themselves as being a different user-agent. Other browsers allow the user to change this information at will.

As the number of different browsers being used increases, browser-sniffing scripts become more and more complex. They need to test for all possible combinations of vendor and version number in order to ensure that they work cross-platform. This is a Sisyphean task that can result in extremely convoluted and messy code.

Many browser-sniffing scripts test for an exact match on a browser's version number. If a new version is released, these scripts will need to be updated.

Thankfully, the practice of browser sniffing is being replaced with the simpler and more robust technique of object detection.

What's next?

In this chapter, I've introduced some important concepts and practices that should be at the heart of any DOM scripting you do:

- Graceful degradation
- Unobtrusive JavaScript
- Backwards compatibility

You've seen how we can learn from other technologies like Flash and CSS to ensure that JavaScript is used wisely. A cautious, questioning attitude certainly seems to be a desirable trait when you're writing scripts.

I'd like to take that attitude and apply it to the example from the last chapter. In the next chapter, I'm going to re-examine the JavaScript image gallery, identify its flaws, and take steps to remedy them.