O'REILLY® Short Cuts

Unobtrusive Ajax

by Jesse Skinner

Copyright © 2007 O'Reilly Media

ISBN: 978-0-596-51024-4

Released: July 10, 2007

Unobtrusive Ajax is about making web applications that work for everyone all the time, even if you have JavaScript turned off, or you're using a mobile phone or a screen reader, or however you happen to be using the Web. It's about the separation of behavior (JavaScript), content (HTML), and presentation (CSS).

This short cut will focus on the practical benefits of using Ajax and JavaScript unobtrusively and show you that unobtrusive web development and progressive enhancement benefit both web developers and users of the Web. You'll get to see many simple examples of building web interfaces that are unobtrusive. You'll quickly see that it is actually very easy to make web applications that everyone can use.

When you're finished reading this book, you will be able to convince anyone why developing unobtrusively is the best way to build a site with JavaScript and Ajax.

Contents

Ajax has changed the way we think about web applications and the Web in general. It has made it possible to create web applications and interfaces that are even better than what has traditionally been done on the desktop. No longer do we have to wait for the page to refresh, plus we get access to enormous amounts of data that we never would have had on the desktop.

Unfortunately, Ajax has also made the Web a lot more inflexible. Some web sites require a fast computer, a fast Internet connection, a large monitor, and a very modern web browser with JavaScript enabled. Many sites even require the user to be able to use a keyboard and mouse and to have good vision. The web developers who make these obtrusive sites often have little hesitation requiring these things from users — after all, most web developers have fast computers, fast Internet connections, large monitors, and no physical disabilities.

The web developers or managers who make such demanding applications often argue that it's a waste of time and money to develop a version for the small minority with disabilities or without JavaScript. They argue that desktop software has always made system requirements, so they should be able to, too.

If these arguments sound familiar, or if you've found yourself making them yourself, then this book is for you. I'll show you that web applications don't need to have such requirements. I'll show you that you don't have to make two versions of your application, you can build just one version that everyone can use. I'll show you that making an accessible site doesn't just benefit your users, it will benefit your developers.

And no, this book won't tell you to stop using Ajax and JavaScript. In fact, the majority of your users won't be able to notice the difference. However, that minority of your users you've been neglecting will be extremely grateful.

I'll walk you through some examples of Ajax being used in the wild, and show you exactly how to get these Ajax techniques to work both with and without JavaScript. After a few examples, you'll see that it's actually very easy and straightforward, and you'll be able to apply the techniques to any JavaScript-based web development.

I'll also show you how Unobtrusive Ajax benefits web developers just as much as it benefits users of web applications. You'll have all the arguments you'll need to convince a skeptical boss or client why developing unobtrusively is the smart choice. So dispel your disbelief, save your questions for the end, and stay tuned as I show you what Unobtrusive Ajax is, why you would want to use it, and most importantly, how you can use it in everything you do.

What Is Unobtrusive Ajax?

Unobtrusive Ajax is a technique for developing web applications. It's not a web standard; it's a best practice for creating web applications that work in the widest number of browsers and clients by making the fewest assumptions.

With so many really cool and cutting-edge interfaces on the Web, it's easy to get really excited and carried away. Sometimes it seems that the wow-factor has become a priority, with everything else secondary.

This often means that the core functionality and content of a web site can't be accessed except with JavaScript or Flash. Usually it isn't the functionality or content itself that actually requires JavaScript or Flash, it's just the interface to the functionality and the content that makes these requirements.

A few years ago, before Ajax had a name, there were all kinds of cool web sites. These sites didn't have Ajax or Drag-and-drop, but they still worked. Sure, people had to wait for the page to refresh, and the interfaces weren't anything to brag about, but the sites got the job done.

These "old-fashioned" sites were built using basic HTML. All the content was delivered right inside each HTML document, and functionality was achieved through HTML forms communicating with server-side applications.

JavaScript adds a layer on top of HTML that makes the web page more interactive. Actually, most of what JavaScript does is add and change the HTML on the page dynamically.

Even Ajax is just talking to the web server in the same way HTML forms do, submitting variables and getting back some content. In the old days, that content would be a new web page. With Ajax, the content is usually a part of a web page or some other data.

Separating Behavior, Presentation, and Content

Unobtrusive Ajax is about separation. It's about separating JavaScript from HTML, the behavior from the content, so that if the HTML stands alone, it still works. It's also about separating CSS, the presentation, so that the JavaScript and the HTML don't get too concerned with defining what things look like.

The separation of JavaScript, CSS, and HTML can happen on both a physical and conceptual level, and I'll explain the differences and benefits of both.

In any case, this separation helps keep things more organized for developers, and makes sure that each level, especially the HTML, can stand alone. The only thing you can assume about people who use the Web is that they are capable of using

and interacting with plain HTML web pages. As you'll see, everything else can and should be optional.

Physical Separation

Physical separation means putting your JavaScript, HTML, and CSS in separate files. or at least separating them within a web page. It's about avoiding attributes like `onclick`, `onload` and `onmouseover`. It's also about avoiding the `style` attribute to change the way a single element looks.

By using these attributes, you're mixing code written in different languages. This makes it more difficult to understand and manage the code later on. If you have JavaScript and CSS scattered throughout a page, or scattered across your site, it becomes a chore to find the code to make changes to it.

One option is to put all the JavaScript on a page into a `<script>` block, and all the CSS into a `<style>` block. While these techniques are better than using HTML attributes, putting JavaScript and CSS into separate, external files, attached to the page with `<script>` and `<link>` is even better. This way, the JavaScript and CSS files can be cached by the browser and reused across multiple HTML documents. This has a number of benefits which I will discuss in a minute.

> **Does it Ever Make Sense to Use <script> and <style> Blocks?**
>
> There are times when it's more practical and easier to put some JavaScript and CSS directly on the page rather than keep everything in external files. You may want to consider this if:
>
> 1. The JavaScript or CSS is very specific to one page and the page won't be accessed very often.
> 2. You only need to use a few lines of JavaScript or CSS.
> 3. You want to pass variables to a third-party script, like web statistics.
>
> Every situation is different, and you'll want to be careful to weigh the benefits of client-side caching and easier maintenance with the difficulty in creating external files. I'm sure you'll agree it's rarely difficult to create external files.

It's also beneficial to separate the CSS out of the JavaScript. You can do this by using class names whenever possible. For example, it's better just to give something a class name like *warning* rather than turn it specifically red using JavaScript. This way, if the color scheme changes, you won't have toworry about changing your JavaScript.

The goal here is to be able to change all the CSS or all the JavaScript without having to change anything else. In reality, this isn't always possible. However, the closer we can get to this point, the easier maintenance will become.

Let's look at some common scenarios in web development, and see how physically separated code would make them easier.

Changing the design of a web site

If a site is designed without using any CSS, then you will need to throw out all the HTML and start from scratch. The HTML can define how things look, but it means you can to copy the look of things across a bunch of documents.

However, if the entire design for a site is in a few CSS files, then you have just one place to go to change the way things look. Physically separating the CSS from the rest of the site makes it very easy to find and update the design.

Reorganizing or rewriting the JavaScript for a web site or application

When you finish developing a piece of code, you sometimes hope that it will be used forever. Of course, code is usually quite temporary. This is even more true in a fast-paced environment like the Web. If you've ever had the pleasure of cleaning up JavaScript that looked for `document.layers` in order to support users of Netscape 4, then you understand how temporal web development really is.

If there is JavaScript scattered throughout a site, with event attributes like `onclick` being used liberally, then rewriting the JavaScript will involve rewriting much of the HTML. Chances are, these `onclick` attributes will call functions found in other parts of the page, or in other external files.

By keeping all related JavaScript in a single, external file, you can attach click handlers close to the same place you define your functions. When you need to rewrite your code, you'll be able to delete all the JavaScript and start over without having to worry about some button breaking somewhere in your application.

Making your code more understandable

When you're writing code, sometimes you trust that you'll be the only person who will need to work with it. Unfortunately, eventually someone will probably have to look through your code to understand how everything fits together. This person may actually be yourself — if you spend long enough away from your code, you can easily forget nearly everything.

If JavaScript and CSS are scattered throughout HTML pages, it makes it very difficult to track down code in case a bug creeps up or a feature needs to be added. Having all the CSS and JavaScript in their own files means that developers know where to go to change the behavior (JavaScript) or the design (CSS).

Having these separated in their own files also makes reading the code a lot easier, simply because they are very different languages. If you've ever tried to read something like the following, you know what I mean:

```
<form action="/item/edit" method="post" onsubmit="if (!validateForm()){
this.style.background = '#FFAAAA';
this.style.border = '1px solid red'; alert('Please complete all fields!');
return false; }">
```

It takes a lot of practice to understand a line of code like that, and I've seen much worse. Imagine what that line looks like with sever-side logic added to it.

You want to reduce bandwidth and improve loading times

JavaScript and CSS are mainly "static," in the sense that they rarely change due to the state of the server, and they are usually shared across all users. By separating all the CSS and JavaScript into separate files, the browser can cache these files, reducing the size of your HTML, thus reducing the time it takes your pages to load, and reducing your bandwidth. This also gives you the opportunity to apply compression to the JavaScript and CSS, something you can't do when they are inside the HTML.

You can also use techniques to combine all the JavaScript or CSS into a single file, again improving loading times by reducing the number of requests the browser makes to the server. None of this is possible when your JavaScript and CSS is scattered throughout your HTML.

Conceptual Separation

Separating JavaScript, CSS, and HTML into different files is a great way to organize your web site or web application, but it mostly benefits web developers more than the people who visit web sites. Conceptual Separation means separating the behavior (JavaScript), presentation (CSS), and content (HTML) so that they are as independent from each other as possible.

What does your web site look like when CSS is disabled? How does it work when JavaScript is disabled? What is it like to use your site with images turned off? If we have the content, presentation, and behavior conceptually separated, we'll know that the answers to these questions is always "just fine, thanks".

A site can have all the JavaScript separated into external files and still break horribly when JavaScript is turned off. For example, there are web sites on the Web right now with completely blank HTML documents, where the JavaScript loads everything via Ajax and creates the entire contents of the `<body>`. Of course, when someone has JavaScript turned off, they get a blank page.

A site can also have all the CSS in an external stylesheet but still rely heavily on tables and images for layout and break entirely when someone uses a text-based web browser.

There are also sites that have JavaScript scattered completely in event attributes that work wonderfully when JavaScript is disabled. Similarly, there are sites where all the presentation (CSS) is in `style` attributes that degrade nicely on text-based browsers.

Why bother using Conceptual Separation?

The benefits of using Physical Separation are clear to most web developers, because it benefits us the most. Conceptual Separation mostly benefits the users of the site, but not entirely. Some of the benefits include:

1. **The site will work when users have JavaScript disabled**.
   According to some statistics, between 5 to10 percent of web users have JavaScript disabled,possibly due to company policy, security concerns, a slow connection or browser incompatibility.
2. **The site will work when the JavaScript has an error or fails to load**.
   We don't like to admit it, but everyone makes mistakes. What happens if you upload a buggy script file right before the weekend without testing it in every browser? Sure, it might work fine in Firefox, but that extra comma in your JSON will cause it to fail in Internet Explorer.

   By ensuring you have a solid HTML base that works without JavaScript, you can rest assured that people will still be able to use the site (although without all that great drag-and-drop functionality).
3. **The site will automatically be more accessible**.
   Accessibility is about letting everyone access the site, no matter what limitations they face. By separating the presentation and behavior from the content, you make the HTML much easier to access.

   For example, if your color scheme is giving someone with color blindness a hard time, they'll have the technical option of disabling CSS or using a user stylesheet to change the colors of your site — but not if the colors are written right into the HTML using `<font>` tags and `bgcolor` attributes. If the content is separated from the presentation and behavior, you give your visitors a choice in how to use and interact with your site.

By separating the layers of your web site conceptually, you make them more independent, and you make sure that if one of them fails or can't be used for any reason, the core of your site, the content and HTML, will still work just fine.

How Many People Have JavaScript Disabled?

This is probably the biggest question that is asked around the topic of Unobtrusive Ajax and progressive enhancement.

Most statistics show that around 5 to 10 percent of people have JavaScript disabled. There are two popular web sites that collect broad Internet statistics:

http://www.thecounter.com/stats/
http://www.w3schools.com/browsers/browsers_stats.asp

Every web site is different, so you may want to collect your own statistics to see for yourself. No matter what the numbers, you can be sure there will always be someone with JavaScript turned off.

Similarities to Model-View-Controller

If you do a lot of server-side development, chances are you've heard of the *Model-view-controller* (MVC) pattern. This concept means you separate the code in applications so that the code that deals with the database (Model), the code that generates the user interface (View), and the business logic in between (Controller) are each separated into their own layers. This way, the entire user interface can be changed without necessarily changing any of the business logic code. Similarly, the database can be entirely restructured without needing to change either the business logic or the user interface.

The similarities between Unobtrusive Ajax and MVC are quite striking. If you use your imagination a bit, you could see the parallels between the content (HTML) and the model, between the presentation (CSS) and the view, and between the behavior (JavaScript) and the controller. Well, not really, since the JavaScript doesn't sit between the CSS and HTML, but that doesn't really matter. They are both about separating code with a similar purpose into their own independent layer.

The benefits of using an MVC framework take some time to realize. Sure, it may seem like needless overhead at the start of a project. It's very tempting to scatter database queries, business logic, and HTML all together in the same templates at first, and it does get the job done faster. However, a few days later when you need to make changes, you'll feel the pain as you imagine all the templates that you need to go through. You also leave yourself open to forgetting about more obscure templates and how all the templates interact with each other.

The moral of the story is: keeping similar code together makes maintenance **a lot** easier. The points of interaction are kept to a minimum, and this means you can make changes in each layer without affecting the other layers.

Summary

- Separating JavaScript and CSS into their own files reduces bandwidth and load times by allowing the browser to cache these files.
- Separating languages makes all the code easier to read.
- Sites with JavaScript and CSS in external files may still be very obtrusive.
- Structuring a web site into independent layers improves both maintenance and accessibility, and makes a web site more robust in case of failure.
- Sometimes it makes sense to use inline JavaScript and CSS, but very rarely.

Using Web Technologies Unobtrusively

Each web technology, whether HTML, CSS, JavaScript, Flash, or anything else, has its own problems and challenges for being unobtrusive. Using a technology unobtrusively means the technology tests to see if it can be applied before applying itself.

In contrast, using a technology obtrusively means shoving it in people's faces, forcing them to use it whether or not they can. Here are some ways to use HTML, CSS, JavaScript, and Flash unobtrusively.

Unobtrusive HTML

Yes, I know HTML is always pretty unobtrusive. How many of your visitors are using a browser that doesn't support HTML? Maybe a couple of years ago I'd consider supporting Gopher, but I think today we can agree that HTML is the bare minimum we can require from our visitors.

Nonetheless, there are ways to abuse HTML and make it more dependent on other technologies and browsers, so let's look at some best practices for using HTML.

Always use valid HTML or XHTML

It doesn't really matter which type of HTML you choose to use, whether HTML 4.01 Transitional or XHTML 1.1 Strict. What's important is that you use it by the book (the book, in this case, being the W3C Specifications).

Using valid HTML isn't about being a perfectionist or being able to claim superior knowledge of the HTML specification. Using valid HTML means that you're creating a HTML document that is independent of any web browser differences.

When people make web browsers, they grab a copy of the same W3C Specifications in order to understand what all the HTML elements are supposed to do, how they're supposed to fit together, and what the different attributes are supposed to mean. By following the specification and creating valid HTML, we know that any web browser that also follows the specification will be able to use our HTML in the same way.

However, if we do invalid things in our HTML, we're forcing web browsers to make guesses and assumptions. When this happens, we end up with differences across web browsers, because web browser makers can't make the same guesses all the time.

For example, what happens if you put a form around a single table row inside a table, like the following:

```
<table>
	<form id="login" action="/user/login" method="post">
		<tr>
			<td>Name: <input name="name" type="text"></td>
			<td>Password: <input name="password" type="password"></td>
		</tr>
	</form>
</table>
```

The HTML specification says that a `<form>` element isn't allowed to go directly inside a `<table>` element. If you do this, you're forcing the browser to make a decision about your mistake. It has to make a guess about what it is you were trying to do, and its guess may not be what you expect.

Here is how Firefox interprets the above HTML:

```
<table>
	<form id="login" action="/user/login" method="post"></form>
	<tbody>
		<tr>
			<td>Name: <input name="name" type="text"></td>
			<td>Password: <input name="password" type="password"></td>
		</tr>
	</tbody>
</table>
```

That's right, it ends the `<form>` element immediately after it starts. This has very real implications for DOM scripting. If you want to find all the inputs inside the login form, you might try something like this:

```
var login_form = document.getElementById('login');
var inputs = login_form.getElementsByTagName('input');

alert(inputs.length); // alerts "0"
```

Surprise! The inputs you put inside a form are no longer where you thought they were, because the browser had to make a guess about what you were doing.

Here's how Internet Explorer 7 interprets the same HTML:

```
<TABLE>
       <FORM id=login action=/user/login method=post>
       <TBODY>
              <TR>
                     <TD>Name: <INPUT name=name></TD>
                     <TD>Password: <INPUT type=password value="" name=password></TD>
              </TR>
       </FORM>
       </TBODY>
</TABLE>
```

Notice that Internet Explorer has made much different assumptions about how things fit together, and that these assumptions have totally broken the nesting of elements: the `<TBODY>` starts inside the `<FORM>` but ends outside!

I encourage you to run your own tests with the ways browsers interpret invalid HTML. You can find the results I did by creating documents and inspecting the value of `document.body.innerHTML`.

Use Semantic HTML

HTML is a document markup language with dozens of elements that define page headers, lists, forms, form fields, form field labels, paragraphs, links, and lots of elements that give a document structure and, to some degree, meaning.

That's what Semantic HTML is all about: giving a document more meaning by careful use of HTML elements.

Header elements (`<h1>`, `<h2>`, etc.) give the document the most structure by defining a nested series of sections, and giving each of those sections a meaningful title. There are tools and browsers out there that can actually use header elements to create a table of contents for a web page. For example, this allows a screen reader to give users a quick overview of the contents of a document without having to listen to the entire thing. This could also be used on mobile devices with very small screens where a visual scan of the document is more cumbersome or impossible.

Header elements are also very useful for tools which try to understand the meaning of documents, such as search engine spiders. Search engines often give more importance to the words in a header element, and use those words to give more context to the paragraphs which following.

Lists (`<ul>`, `<ol>` and `<dl>`) also give a document some structure, though arguably less than headers. Lists simply group similar items together in a very general way.

This grouping can be useful to both JavaScript and CSS for hooking in design and functionality. For example, you might use CSS to create a separator between each list item, or use JavaScript to show and hide list items dynamically.

Tables are the perfect HTML elements for tabular data, like displaying calendars, charts or spreadsheet data. They are, however, a poor choice for layout, such as giving a web page multiple columns or structuring a form. HTML tables have a number of elements and attributes which give you the opportunity for adding context and meaning to tabular data, elements like `<caption>`, `<th>`, and `<colgroup>`, and attributes like `header`, `scope`, and `axis`.

Form labels (`<label>`) give a way of associating words (the label) with an input field in a form. This gives meaning to the input field, making things easier for those using browsers and devices like screen readers, but also giving the browser the opportunity to make things easier for users. For example, when a label is used for a checkbox, often clicking the label will toggle the checkbox.

Then there are generic, meaningless elements like `<div>` and `<span>`. These are used and often abused by web developers, usually touted as a better alternative to using tables. Divs and spans have no semantic meaning. They're just empty placeholders. Indeed, you could practically build an entirely web page using just div and span tags, but there would be little structure to the page. If CSS were disabled, the browser would render the page as completely flat text, unable to tell the difference between a title and a paragraph and a list item.

By using Semantic HTML, you give browsers, devices, search engine robots, and other things the ability to take your HTML and use it by itself, without needing any CSS or JavaScript. Try commenting out the CSS on your web pages and see how they look. Is there enough semantic meaning and structure used in the page that the browser knows what to do with it to make it readable?

Unobtrusive CSS

CSS is a wonderful concept for separating presentation from content. The more attributes or elements that you can take out of the HTML and replace with some CSS, the better. This means getting rid of all the `<font>` tags and `align` attributes. The more details you have about the presentation in the CSS, the easier it will be to change the presentation for different devices and browsers. As I mentioned earlier, it will also make things very easy for you when you decide you want to change the look of your site or application.

You can easily have multiple stylesheets for the same HTML page. You might have one for the screen, one for print, one for mobile devices, etc. You might even have a special stylesheet for people with JavaScript enabled (more about that later).

For an excellent example of using multiple stylesheets with a single HTML document, check out CSS Zen Garden at **csszengarden.com** [***http://csszengarden.com***]. This site is a single HTML page that invites web designers to create a new stylesheet that completely changes the look of the page. As you go through the hundreds of different designs (986 at the time of this writing), you quickly discover just how independent the design really is from the content.

It's possible to go too far when using CSS, when you end up removing semantic meaning from the HTML. For example, it's better to use `<h1>` instead of `<div class="header">`, and `<strong>` instead of `<span class="bold">`. The semantic meaning of the page should remain in the HTML, whereas the details about what that meaning looks like should go in the stylesheet.

Unobtrusive Flash

Most Flash websites are completely obtrusive, requiring visitors to have Flash installed, whether that is feasible or not. Other web sites try to be slightly less obtrusive by giving visitors a choice between a "Flash Version" and an "HTML Version."

Truly unobtrusive Flash doesn't need to ask the visitor. Instead, it can detect the presence of the Flash player, and if it's a new enough version. If so, the Flash content will be loaded and put onto the page, replacing the default HTML content. If Flash is not supported, the HTML content remains on the page, and the web site remains usable.

This kind of dual Flash-HTML web site has a number of benefits, especially when it comes to accessibility and search engine optimization. Visitors who can't use use Flash, whether because of technical limitations (a mobile device, for instance), or physical limitations (unable to use a mouse), or because the or she is actually a search engine spider looking for text to index, will all be able to access the core content of the site.

Musicians seem to love Flash web sites, and often put their tour dates directly inside the Flash files. If you search for the musician's name and venue on the Web, you may never find the site. If you're out of the house and you want to double check the start time of the concert using your phone, again, you're out of luck.

By requiring your visitors to have Flash supported, you're taking away their ability to choose how to use the Web in exchange for some special effects and animation.

Some people would complain that it takes too much work to put the tour dates on an HTML page as well as a Flash page. If designed correctly, you don't necessarily need to have the information copied into two places. There could be, for example,

an XML file that contains all the tour information. The web site could read the XML file and create the HTML for it, and the Flash could similarly load up the XML file and display the same information. This way, only the XML file needs to be changed.

To accomplish this dual Flash-HTML technique, there is a really great JavaScript library that has become a kind of industry standard: Bobby van der Sluis' Unobtrusive Flash Objects (UFO), available at: ***http://www.bobbyvandersluis.com/ufo/***

This script lets you suggest replacing an HTML element (identified by ID) with a Flash movie. If Flash is supported, it will be replaced. Otherwise, the HTML content remains on the page.

Here is a simple example that shows how easy UFO makes this:

```
<script type="text/javascript" src="ufo.js"></script>
<script type="text/javascript">
var FO = {
      movie: "mymovie.swf",
      width: "600",
      height: "100",
      majorversion: "6",
      build: "0"
};
UFO.create(FO, "flash_banner");
</script>

<div id="flash_banner">
      Here is where the alternative HTML content would go.
</div>
```

Notice that this technique also means you don't have to mess up your HTML files with `<object>` and `<embed>` elements. It also lets you specify a Flash version, so if a visitor has an older version, she will still get your HTML content rather than a broken page. It's actually easier to use this technique to add Flash to a page than the traditional method.

Unobtrusive JavaScript

If Flash can be made unobtrusive, then JavaScript can be made even more so, and more easily than Flash.

What happens if JavaScript is unavailable? Do we need to make alternative content for those without JavaScript like we did for those without Flash? Certainly not. We can take advantage of the fact that JavaScript is built on HTML.

Nearly anything JavaScript does involves some HTML. JavaScript creates and manipulates the HTML of a page in order to create a dynamic, interactive experience.

We don't need to send data in an XML file to both the HTML and the JavaScript (though this is, of course, a possibility). Instead, we have the option of putting all the content into the HTML in the first place, and using JavaScript to simply enhance the experience of interacting with that content.

Using JavaScript unobtrusively means always keeping in mind that JavaScript is an optional part of the web site, something that you can't rely on. It means remembering how web sites were made back in Web 1.0, back before Ajax had a name, and even before mouse roll-overs gained popularity.

Anything that goes directly into the HTML should be completely independent of any JavaScript that will be added to the page. This means that every link goes to another page, and every form submits to a page that processes it. It also means that you don't assume your JavaScript form validation will prevent bad values from being submitted. In total, it means that your whole web site works without a single line of JavaScript.

Here are some examples of code that is very common but very obtrusive:

```
<!-- it says "add comment" but really means "add nothing" -->
<a href="javascript:void(0)" onclick="addComment()">add comment</a>

<!-- where does this go? the top of the page? -->
<a href="#" onclick="refreshData()">refresh</a>

<!-- this submit button doesn't submit anything.. and where is the form?? -->
<input type="submit" onclick="location.href = 'somepage.html'" value="some page">

<!-- how will anyone ever see the sub menu with JavaScript disabled? -->
<div onmouseover="document.getElementById('submenu').style.display = 'block'"
     onmouseout="document.getElementById('submenu').style.display = 'none'">

     Top Level Menu

     <div id="submenu" style="display: none">
          <a href="page1.html">Sub menu option</a>
     </div>
</div>
```

I'm sure you've seen examples like this before, and probably even used these techniques yourself (I know I'm as guilty as anyone). Well, they all work great when JavaScript is enabled, but the minute JavaScript is disabled, suddenly the site becomes completely unusable.

Probably the most obtrusive JavaScript technique is the use of JavaScript-only links and buttons, the bulk of the above examples. Here is a much better way to have links and buttons that have some JavaScript functionality attached to them:

```
<!-- let people add a comment on the add_comment.html page -->
<a href="add_comment.html" onclick="addComment(); return false">add comment</a>

<!-- refreshing a page doesn't need Ajax, it can just be a link to the same page -->
<a href="thispage.html" onclick="refreshData(); return false">refresh</a>

<!-- why bother using JavaScript at all when plain HTML will do? -->
<form action="somepage.html" method="get">
    <input type="submit" value="some page">
</form>
```

These examples are completely unobtrusive, and the JavaScript is conceptually separated from the functionality of the HTML. (True, the JavaScript isn't physically separated from the HTML, but it would make my simple examples messier, so cut me some slack.)

Achieving the dynamic navigation is a more complex example, possibly involving special CSS for only those with JavaScript. The end result would be that the navigation is fully expanded for those without JavaScript, and the submenu is only hidden when JavaScript is enabled. I'll cover more complex examples later in this book. You can skip ahead if you're impatient. I won't be offended.

I think the physical separation of JavaScript and HTML is stressed a lot because it forces you to see the HTML document as something that can stand alone. Seeing an `onclick` or `onmouseover` attribute on something seems to satisfy the question of "what is this element for." I think when you separate the JavaScript out of the HTMLthe result looks absurd without some default functionality attached to it:

```
<a href="#">Edit User</a> <!-- what? the edit page is '#'? -->

<div>Show Menu</div> <!-- how is a <div> supposed to show something? -->
```

Using JavaScript unobtrusively also means not making assumptions about the browser. Anyone can come along and make a new browser that doesn't support `innerHTML` or `getElementById`. We don't have to choose to support these browsers, but we do have to acknowledge that they could exist. JavaScript can be made more robust by using object detection to check what the browser supports. This becomes important as web browsers add new features, such as new DOM Scripting functions.

If we know the plain HTML-only version of the website works great, then we don't have to feel bad about requiring certain JavaScript functionality. Here is a simple example that will only let a script run if `getElementById` is supported:

```
function myCoolScript() {
	if (!document.getElementById) {
		// we don't want to support this browser
		return;
	}

	// do fancy JavaScript stuff
}
```

It obviously isn't practical to check every function before you run it, but checking some core functions is a convenient way of making sure a browser is up to par.

Building in some plainold-HTML functionality into the page can also improve usability. Not everyone is comfortable and familiar with using drag-and-drop to add something to their shopping cart. By adding an old school "Add to Cart" button together with drag-and-drop functionality, you keep the "wow"factor for the Internet savvy, but still build in some basic, form-submitting, link-clicking functionality for everyone else. This can also have the convenient side effect of letting people without JavaScript (or without a mouse) add something to their shopping cart. I think you'll agree that adding something to a shopping cart is functionality you don't want to break.

The simplest way to build a site that uses JavaScript unobtrusively is to enforce a rule in the early stages of development: **No JavaScript Allowed**. Wait until all the core functionality is in place before you add a line of JavaScript. Make sure that all the server-side validation is in place, all the content can be accessed without any Ajax, all the forms do what they say they will do, and all the links go where they say they will go.

Once you know a site works without any JavaScript, then you can decide how to enhance it and give it that all-important "wow"factor (hopefully without changing any of the HTML). You can hijack a link to an edit form and add the Ajax to load the edit form into a floating panel. You can make the columns of a table sortable. You can enhance all the form inputs to make them more usable. And you can rest assured that the site will work great whether or not JavaScript is loaded.

I will go into these examples in detail, and show you exactly how to add JavaScript to a functional web site. But first, let's look at some more reasons why you should consider using Ajax unobtrusively.

Summary

- Using a technology unobtrusively means making it optional and independent, and forcing it to test if it can be applied before it applies itself.
- You can always rely on HTML, as long as it's valid.
- Take advantage of the full range of HTML elements.
- CSS can and should be used to describe what a page looks like.
- Even a web site with a ton of Ajax and Flash can be built unobtrusively.

Why Use Unobtrusive Ajax?

I hope by now I've already convinced you of the practical benefits of using Ajax unobtrusively. You've seen that separating and grouping HTML, CSS, and JavaScript together makes your code cleaner and easier to maintain. You've learned how to use HTML and CSS to its full potential in order to make the use of JavaScript less crucial. I've also told you all about how your sites will be much more accessible and usable when you don't rely on JavaScript or Flash alone.

Well, in case you're not yet convinced, I'd like to give you even more reasons to see the light. If you are already convinced and I'm just preaching to the choir, read ahead to learn how to convince those around you why they should care about the minority without JavaScript.

You Don't Have to Use Unobtrusive Ajax

It's true. There's no law saying you have to use Ajax unobtrusively, and in actuality, most people don't. It's important to understand that sometimes an obtrusive web application makes some sense. However, I think you'll see that those scenarios are quite rare, quite specialized, and very obvious. In other words, chances are your web application isn't the exception.

Even if a component of your web site requires JavaScript or Flash, I don't think any web site or application would need JavaScript entirely from start to finish. There should be as much content available to everyone as possible.

There are quite a few web scenarios that require JavaScript or Flash, and sometimes this makes perfect sense. Most of the time this requirement is highly unnecessary. Let's look at a few examples where it may be necessary.

- **Sites with video and audio**
  What would YouTube be without Flash (or some other video plug in)? Sure, people could write transcripts of all the videos, but I don't think a transcript is going to capture a monkey peeing into its own mouth. Notice that even You-

Tube only requires JavaScript and Flash to watch the videos themselves, not to use the rest of the site.

- **Mashups of Ajax web services**
  If you're building a site that relies heavily on JavaScript-based web services, then I think you can get away with requiring JavaScript. You may want to give serious thought to the situation where someone doesn't have JavaScript, and think about how much you can offer these visitors. For example, maybe you can accomplish much of the same functionality by moving the bulk of the mashup to the server-side.
- **Games and other interactive sites**
  It's perfectly okay to require JavaScript in order to play a JavaScript- and Ajax-based game. Even still, it would be possible to build a checkers game that worked by using a meta refresh, but there's probably a point when you need to draw the line.

If you're doing something that is impossible to do without JavaScript or Flash, then you have yourself a good excuse. Here are some things that don't count as excuses:

- **Drag-and-Drop**
  So your web application just wouldn't be as fun and easy to use without drag-and-drop? What about users who aren't comfortable using drag-and-drop? What about users who are unable to use a mouse? You really should build in an alternative way of working with your application anyway, and you might as well make it work without JavaScript.
- **All the content is loaded with Ajax**
  If you really want the content to be loaded without refreshing the page, at least give the content a home. Make a real old-fashioned URL for the content and stick that in the `href` of the links in your navigation. When people with JavaScript click the links, you can have your Ajax load up the content and display it. When everyone else follows the link (including search engine spiders), they will still get the content on a page by itself.
- **A great interactive user experience**
  If you think that your web application has such a great user experience that it wouldn't even be worth using without Ajax and JavaScript, then you are making a decision about how your users can use your application (and, in effect, making a decision about which users can use it). If you really want to give your users a great user experience, spare them from seeing a message saying, "Sorry, but you need to have JavaScript enabled."
- **It would cost too much to support users without JavaScript**

It's not as hard as it sounds to build a JavaScript interface unobtrusively. Often, it's very easy. You won't have to make a whole new version from scratch for those without JavaScript, you'll just have to make small changes to the way you put the site together. In the end you'll have an application that is more robust, easier to maintain, and also accessible to everyone.

In the end, it will always be your decision (or the decision of your boss or client) how accessible the web site will be, who will be able to use it, and how the users will be able to use it. If you want to turn away a portion of your potential users, go ahead, but don't make this decision lightly.

Making Web Development Easier

I've already shown you how much easier it is to make drastic changes to a web site or application when JavaScript and CSS is physically and conceptually separate from the HTML. This is simply because well organized code is always easier to read and maintain.

When I talk about maintenance, I'm talking about coming back to the code after some time and making changes. These changes could be bug fixes or new feature enhancements. Either way, a web site built unobtrusively will make both of these easier.

If you find yourself with a very serious bug in your JavaScript, and you know the site works without JavaScript, you always have the option of disabling JavaScript on the effected pages while you work to resolve the bug. Sometimes this happens without you making the choice; sometimes the bug causes the JavaScript to break completely. Even with the JavaScript broken, hopefully the site will work fine without it.

Fixing a bug is also easier if you know right away where the bug lies. If you separate CSS and JavaScript you know exactly where to fix display bugs or scripting problems instead of trying to find your way around the whole document.

Adding a feature to a web application is made easier when the code is more organized to begin with. It's also a lot easier if the existing features were built in a modular, independent way, so new code and features can be added without breaking any existing functionality.

Personally, I find it more fun to work on a project with really clean, organized code. Developing unobtrusively feels like an elegant solution to the problem rather than a series of hacks and intertwined, messy logic.

Search Engine Optimization

If your site is available to the public, and if it's important to you that people can find your site when they search for the topic you cover, then Unobtrusive Ajax should be very important to you.

Search engine spiders don't have JavaScript or CSS turned on. They look for content directly in the HTML, not dynamic content loaded with Ajax. They also will only find new pages if they come across the URL in the `href` of an `<a>` tag. If your navigation is generated using JavaScript, there may not be links to all of your pages in the HTML.

If all of your content is in JavaScript and Flash, not only will the search engine spiders have a hard time finding it — your potential visitors will, too.

Most search engine spiders also give importance to well structured HTML. As I discussed earlier, Semantic HTML gives your HTML documents more structure and meaning, and by structuring your content with header elements (`<h1>`, `<h2>`, etc.) and linking your pages together with well chosen link text, you give the search engines clues about what your pages are all about.

Accessibility

Accessibility is certainly the biggest reason to use Unobtrusive Ajax, but perhaps not for the reasons you expect. I'm not talking about accessibility from a legal point of view, though that could certainly be important. I'm talking about accessibility in a very broad sense. As far as I'm concerned, accessibility is simply the ability for people to access your web site.

Often when people talk about accessibility on the Web, they narrow their view of accessibility to dealing with blind people using screen readers. Too often I've heard web development managers dismiss accessibility from this perspective;I've actually heard people say, "We don't have to worry about accessibility, I don't think there'll be any blind people using this."Well, there's a lot more to accessibility than screen readers.

Accessibility is about acknowledging that you don't know how people will access your site, and you don't know what limitations he or she may face. Some people want to use the Web from their mobile phones or other mobile devices. Some people are using the Web on very slow Internet connections. Some people prefer to disable images to save bandwidth.

Some people disable JavaScript for personal security concerns,others have JavaScript disabled as part of a company or organization security policy,and others

have it disabled against their will because they're using a browser or device that doesn't support it.

Some people can't use a mouse for physical reasons, others can't use a mouse for technical reasons, and others still simply prefer not to use a mouse if they don't have to.

Some people are color blind, others are partially blind and need to use very large fonts, and others are completely blind and need to use screen readers.

Some people are new to computers and the Internet and have a hard time understanding technical jargon (and they don't have a clue what a tag cloud is supposed to be for, or even what drag-and-drop means), and others speak a foreign language and have a very difficult time understanding your instructions.

I hope you start to see the pattern that emerges. Accessibility affects a lot more people than you may expect. I don't remember who said it, but someone once said "Everyone faces accessibility problems at one time or another."

Maybe one day you'll find yourself trying to check your Ajax-based web mail at an internet cafe with JavaScript disabled. Or maybe your mouse will break and you'll go online to try and buy a new one and find yourself faced with a drag-and-drop shopping cart. Eventually, everyone will find themselves in that "minority" that web developers are quick to dismiss.

One of the most powerful features of the Web is its platform-independence. There is nothing inherent about HTML, JavaScript, and CSS that says what software needs to be running on a computer. Any device, any operating system, any computer can take HTML and display it. By building web sites that rely only on HTML, we make no assumptions about how the web site will be used.

We can still add fancy CSS, JavaScript, and Flash to a web site without breaking the basic functionality of the HTML. This way, we get the best of both worlds. Those with fast computers, fast Internet connections, and modern browsers can get the full "wow"factor we were hoping for. Luckily for us, most people fit in this category. However, no matter how far we look in the opposite direction, there will always be someone who lies outside of our assumptions. For these users, plain, boring HTML is much better than nothing.

Summary

- Unobtrusive Ajax benefits developers by keeping code organized and robust, making it easier to find and fix bugs, or add new features.

- It also benefits users by giving everyone the ability to access web sites no matter what physical or technical limitations they face.
- Search engine spiders are visitors too, and Unobtrusive Ajax benefits them by making sites that they can traverse and understand.
- Unobtrusive Ajax isn't always feasible, but those situations are rare.
- Even when being fully unobtrusive isn't possible, you should try the best you can to offer some level of content and functionality to everyone.

How to Use Unobtrusive Ajax

Okay, enough of the benefits for using Ajax unobtrusively.Let's get to the fun stuff.

Convincing Your Bosses and Clients

Certainly the first step in doing any programming, you have to convince the people who pay the bills, right?

Well, maybe not. Let me suggest something daring: don't ask for permission.Just do it.

Developing unobtrusively is much more a technical design than a design decision. Your bosses and clients depend on you to use your education and technical know-how to make the appropriate technical decisions when you work. Do you need to ask them before you comment your code? Do you need permission to decide whether or not to program with a Model-View-Controller pattern? Unless you are doing work for someone highly technical, chances are the answer is no.

You are responsible for making technical decision because you understand the pros and cons, and you understand how difficult it is to do things. Chances are, the people who pay the bills don't. Here is a bad example of a possible conversation, sadly one I've had several times in the past:

You: "What do we do if JavaScript is turned off?"

Boss: "Well, just display a message that they need to have JavaScript turned on."

Seems like a logical decision, doesn't it? Notice there are a lot of assumptions being made without being spoken. First, it's assumed that everyone has the option of turning JavaScript on. Second, it's assumed that the only way the web site will work is with JavaScript enabled. These are big assumptions, but easy to make when you don't have all the information.

Here's a much better example of the same conversation:

You: "Do you want me to spend a few hours going through the site and making sure it works for people who can't use JavaScript?"

Boss: "Well, how many people don't have JavaScript?"

You: "It depends, but somewhere between 5 to10 percent."

Boss: "You said only a few hours? Sure, go ahead."

Here you've made it clear that supporting those without JavaScript isn't a huge amount of effort, and you've also made clear that for some users, JavaScript is not an option. Your boss will, hopefully, make the most obvious decision to spend a few hours making the site work for everyone.

However, here's my favorite version of the same conversation:

You: "I built the site with accessibility in mind, so it will work for everyone, no matter what browser they're using, even if it's a mobile phone, a blind person using a screen reader, or anyone else. You'll even be able to use the site with JavaScript turned off."

Boss: "Great work! Thanks!"

Your bosses and clients want you to produce a great product that works for everyone. If you can show that you know how to do this, that it's not a big technical challenge, that the benefits are very tangible, and that you're able to accomplish this without a noticeable effort, your bosses and clients will be impressed if nothing else.

If this doesn't work, and your bosses and clients have already decided that people without JavaScript are worthless, you can suggest some other options:

- Remind them of any web accessibility legislation in your country.
- Calculate the lost revenue from potential visitors being denied access.
- Explain that the site will be more stable if it doesn't require JavaScript to work.
- Promise that you won't have to make a separate version for people without JavaScript.
- Insist that it will be easier for you and other developers to maintain and understand the site if it's built unobtrusively.
- Tell them to read this book.

Of course, in the end, there will always be bosses and clients who aren't willing to spend a cent towards supporting the minority, no matter how hard to try to convince them otherwise. But hey, you tried your best, right?

Develop without JavaScript First

This is the most simple and the most powerful technique for developing unobtrusively. I admit, it's a little boring. This Ajax stuff is so cool and so exciting, it's usually the first stuff you want to get to, isn't it? Well, a little patience can pay off.

You should ensure that all the forms have server-side validation. If you don't build server-side validation, you could still end up with invalid data in your system, even if you require JavaScript. Some of your visitors will know how to turn off JavaScript to get around your form validation (I know I've used this technique a few times to make a form field optional that was supposed to be required). If the validation of your data is really important, then you can't trust the browser. If you leave the server-side validation until later, it can be easily forgotten, especially if everyone testing the site has JavaScript turned on. You can even store the rules of the validation in a JSON object, allowing you to reuse the logic instead of duplicating it.

With JavaScript off limits, you start off by building what search engine spiders and users on mobile devices will see. This way, you can make sure that all content and functionality is available to everyone. Later, you can add JavaScript to enhance the experience, loading content dynamically or submitting a form using Ajax to speed things up for those that can handle it.

In these early stages, you can, of course, keep the JavaScript and Ajax features you'll be adding later in mind. Abstract your content and data so that it can be reused and available with Ajax as well as in an HTML page of its own. Separate the presentation (CSS) from the content (HTML) so it will be easier to describe how things look with JavaScript turned on.

Something else that might happen when you develop without JavaScript: you may realize that you can put the site live sooner. Since everything works already, you may decide that you didn't need as much Ajax as you originally thought. If you're running late on a deadline, you'll be able to release a simple version of the site early and get people using it, then add the Ajax features at your leisure.

This is still the Web, and most people are comfortable using traditional web interfaces, clicking links, and submitting forms. You may find that the places where Ajax is most helpful is different from what you originally expected. Often, just a few simple enhancements can make a big difference, even though up front it seems like trying to copy a desktop application feeling is the only way to impress people. If you launch the site without JavaScript first, you can get feedback and user experience to help you decide where you should spend your time adding Ajax.

Use JavaScript Libraries

Nearly all JavaScript libraries make it easier to do DOM scripting, add events to elements, and generally program in JavaScript.

Rather than go into detail about all the different JavaScript libraries, explaining the differences, strengths, and weaknesses of each, I'll talk about the things they have in common.Certain functions and functionality can be found in nearly any JavaScript library, even homemade ones.

I encourage you to use all these functions, if you aren't already. If you're using a JavaScript library already, then I encourage you to find the equivalent functionality in that library and take full advantage of it. If you're not using a JavaScript library, then I encourage you to use the example functions provided in your own projects.

Ajax Function

If you're doing any Ajax, you, of course,will need to have a function that handles the Ajax request for you. I'm sure you've seen a function like this before, but here is one I like to use:

```
function httpRequest(url, callback, data) {
      var httpObj = false;
      if (typeof XMLHttpRequest != 'undefined') {
            httpObj = new XMLHttpRequest();
      } else if (window.ActiveXObject) {
            var versions =["MSXML2.XMLHttp.5.0",
                  "MSXML2.XMLHttp.4.0","MSXML2.XMLHttp.3.0",
                  "MSXML2.XMLHttp","Microsoft.XMLHttp"];
            for (var i=0; i < versions.length;i++) {
                  try {
                        httpObj = new ActiveXObject(versions[i]);
                        break;
                  } catch (e) {}
            }
       }
       if (!httpObj) {
              alert("Error: XMLHttpRequest not supported.");
              return;
       }
       httpObj.onreadystatechange = function() {
              if (httpObj.readyState == 4) {
                    if (httpObj.status != 200)
                          alert("Error connecting to server");
                    else if (callback)
                          callback(httpObj.responseText);
       };

       httpObj.open(data ? 'POST' : 'GET', url, true);
```

```
        if (data)
                httpObj.setRequestHeader('Content-type',
                        'application/x-www-form-urlencoded');


        httpObj.setRequestHeader("X-Requested-With", "XMLHttpRequest");
        httpObj.send(data);
}
```

To use it, simply pass a URL as the first parameter, and an optional callback function as the second parameter. The callback function will get the `responseText` property as the only argument (personally, I always use JSON, so I don't need `responseXML`). If you need to send some POST data, just pass it in a string as the third argument, and the function will automatically call using the POST method.

Event Handling Functions

Nearly any unobtrusive Ajax programming involves adding some event handlers to elements. The old-fashioned way of using attributes like `onclick` works, but not reliably. If you assign a second `onclick` handler to the same element, it will overwrite the first one. Using DOM event handlers, you can add and remove event handlers independently. This keeps your code more portable, because it prevents your events from breaking other scripts on the same page.

In a perfect world, every browser would use the DOM method of attaching events,which is `addEventListener()`. Unfortunately, we have to worry about older browsers, including Internet Explorer.

To deal with various browsers, many people have developed various `addEvent()` functions. There are many solutions out there, and I really recommend using a JavaScript library if only for the event handling functions. For the purposes of this book, just keep in mind that I am using a function that is something like this:

```
// call this function like:
//    addEvent(link, "click", function(e) {
//          // handle the click event
//    });

function addEvent(element, type, handler) {
      // attach function of "type" to element
      // when the event is trigger, handler() will be called
}
```

Preventing Default Event Behavior

I've talked a lot about hijacking links and forms to give new behavior. In order to do this, we need to make sure the browser doesn't execute the default behavior, such as submitting a form or following a link.

When we use a click handler attribute like `onclick`, we can simply return false at the end of the function. However, when using most `addEvent()` functions, like the one above, this won't work. We need to, instead, explicitly tell the browser to prevent the default behavior. We can do that with a function like this:

```
function preventDefault(e) {
    e = e || window.event;
    if (e.preventDefault)
        e.preventDefault();
    e.returnValue = false
}
```

This function takes care of the different ways of preventing default behavior. You would use it in an event handler function like this:

```
addEvent(form, 'submit' function(e){
    // do some stuff

    // prevent the browser from submitting the form
    preventDefault(e);
});
```

DOM Ready Event Function

This solves a different problem from the event handling functions above. Using them, the best you can do is add a `window.onload` function like so:

```
addEvent(window, 'load', function() {
       // do something when the page has loaded (including images)
});
```

However, this method will wait until the entire page has loaded, including all the images on the page. Sometimes you need this, but usually this delay can cause problems. People may try to use your JavaScript functionality before the images have loaded and find the page broken. In reality, you usually just need the HTML to be on the page before you go and manipulate the page.

The standard DOM way of doing this is to attach a DOMContentLoaded event to the document. Unfortunately, at the time of this writing, this isn't supported in Internet Explorer or Safari.

An alternative solution to this is to stick a `<script>` tag right before the end of the `<body>`, and it's the solution used by most people up until 2006. In June, 2006, Dean Edwards, John Resig (same writer of the event handler functions), and Matthias Miller got together and solved this problem. You can see the discussion and solution here at Dean Edwards' web site: ***http://dean.edwards.name/weblog/2006/06/again/***

I adapted their solution into a function I use regularly, called `addDOMLoadEvent()`. You can find this function (including a compressed version) on my web site here: ***http://www.thefutureoftheweb.com/blog/2006/6/adddomloadevent***

Here it is for your convenience:

```
function addDOMLoadEvent(func) {
   if (!window.__load_events) {
      var init = function () {
          // quit if this function has already been called
          if (arguments.callee.done) return;

          // flag this function so we don't do the same thing twice
          arguments.callee.done = true;

          // kill the timer
          if (window.__load_timer) {
              clearInterval(window.__load_timer);
              window.__load_timer = null;
          }

          // execute each function in the stack in the order they were added
          for (var i=0;i < window.__load_events.length;i++) {
              window.__load_events[i]();
          }
          window.__load_events = null;
      };

      // for Mozilla/Opera9
      if (document.addEventListener) {
         document.addEventListener("DOMContentLoaded", init, false);
      }

      // for Internet Explorer
      /*@cc_on @*/
      /*@if (@_win32)
          document.write("<scr"
             + "ipt id=__ie_onload defer src=//0><\/scr"+"ipt>");
          var script = document.getElementById("__ie_onload");
          script.onreadystatechange = function() {
              if (this.readyState == "complete") {
                  init(); // call the onload handler
              }
          };
      /*@end @*/

      // for Safari
      if (/WebKit/i.test(navigator.userAgent)) { // sniff
          window.__load_timer = setInterval(function() {
              if (/loaded|complete/.test(document.readyState)) {
```

```
                init(); // call the onload handler
            }
        }, 10);
    }

    // for other browsers
    window.onload = init;

    // create event function stack
    window.__load_events = [];
  }

  // add function to event stack
  window.__load_events.push(func);
}
```

You use it simply by calling it as many times as necessary, passing it through a function each time. All the functions will be executed as soon as the DOM is ready.

This function or an equivalent is available in nearly every major JavaScript library as well, usually called a DOM Ready or DOM Load function.

Adding, Removing, and Checking for Class Names

If we are going to properly separate behavior (JavaScript) from presentation (CSS), we will have to use class names extensively. We can then easily use class names like "active" or "disabled" to affect the user interface without dealing with any colors or borders in JavaScript.

To accomplish this, we generally need three functions which allow us to add, remove, and check for class names. Since an element can have more than one class added to it, the code can get a bit messy, and we have to use regular expressions to remove and check for classes. That's why functions like these are handy to have around.

These functions typically look something like this:

```
function addClassName(obj, name) {
    if (obj.className != '') obj.className += ' ';
    obj.className += name;
}

function removeClassName(obj, name) {
    obj.className =
        obj.className.replace(new RegExp("(^|\\s)"+name+"(\\s|$)","g"),' ')
}

function hasClassName(obj, name) {
    return new RegExp("(^|\\s)"+name+"(\\s|$)").test(obj.className)
}
```

Selecting Elements by Class Name (nd More)

When working with the DOM, most of the time, you try to collect a set of elements (like all the list items in a navigation), and then work on that set of elements (like attaching event handlers). Doing this with just `getElementById` and `getElementsByTagName` can get rather frustrating.

Many times, you simply need to select a set of elements based on their class name. Using class names lets you add hooks to a bunch of elements on the page, and then you can later use JavaScript to add special functionality to all those elements.

For example, you might add a class name of "tree" to some lists on the page. When JavaScript is enabled, it can go and make the tree an interactive, collapsible tree navigation.

There are many functions on the Web that can find elements based on a class name, and there will be a really fast one built into Firefox 3. There are more advanced functions which can find elements based on XPath queries or CSS selectors, such as those found in the jQuery and Prototype JavaScript frameworks. These are much too complex for me to include here, but if you are using such a framework, I recommend you get familiar with their syntax to really speed up your JavaScript development. If you're not using such a framework, I recommend you have a look at them.

Browser Tools

If you're going to be doing any serious amount of JavaScript development, I highly recommend you use all the web developer tools available. Currently, this includes a few gems:

- **Firebug**. Firebug is the best thing to happen to web development. It lets you manipulate everything about a web page, including the CSS, HTML and JavaScript, on the fly. It is currently available only for Firefox and other Mozilla-based browsers. You can download it here: ***http://www.getfirebug.com/***
- **Firefox Web Developer Extension**. Also for Firefox (or any Mozilla-based browser), this is another extension that every web developer needs to have. The ability to disable and enable JavaScript, CSS, and images is absolutely essential for programming unobtrusively. The ability to validate HTML and CSS easily also comes in handy. You can install this toolbar here: ***http://chrispederick.com/work/webdeveloper/***
- **Internet Explorer Developer Toolbar**. Like the Firefox Web Developer Extension, this toolbar lets you turn on and off JavaScript, CSS, and images easily, allowing you to see your page as others might. You can download a copy here:

http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en

There are lots of other great tools out there to help with programming web sites unobtrusively, but these are just a few I use on a regular basis.

Importance of Links and Forms

When building web sites and applications, links and forms are your best friends. Links are how people can go through your site to reach content, and forms are how people can send data to your server to interact with it. These are the essential building blocks of the Web. They're often the first things to break when relying on JavaScript.

Luckily, they're also the easiest things to fix to make a site work without JavaScript. Let's revisit a common example, a link that triggers a JavaScript function:

```
<!-- this goes nowhere -->
<a href="javascript:void(0)" onclick="fancyAjaxStuff()">wow me</a>
```

Whenever you see an `href` abused like this, you should see your chance to make an unobtrusive fix. All you need to do is specify a real web page in that `href`:

```
<!-- now we have somewhere to go if JavaScript is disabled -->
<a href="old_school.html" onclick="fancyAjaxStuff(); return false">wow me</a>
```

If JavaScript is disabled, the link will just go to our page old_school.html which should provide the same functionality or content as the Ajax function would have. If JavaScript is enabled, the Ajax function will still be called, but the default behavior of the link (going to another page) will be canceled, thanks to the `return false` at the end of the click handler.

Similarly, keep an eye out for forms without a real `action` attribute, like this:

```
<!-- a JavaScript-only form? -->
<form onsubmit="doAjaxSubmit(); return false">
```

Again, this form could easily be made unobtrusive just by specifying a real URL that a non-JavaScript user could submit to:

```
<!-- JavaScript-only no more! -->
<form action="take_my_data.php" onsubmit="doAjaxSubmit(); return false">
```

The `return false` once again stops the form from being submitted for our JavaScript-enabled users, but the `action` attribute gives a backup for everyone else.

I realize that the work doesn't stop at adding URLs to links and forms. You actually have to make the destination page exist, and this does take some more work. It doesn't always have to take a lot of extra work, though. If done correctly, you can probably reuse a lot of the same code. Here are some tips for reusing:

- If you're submitting a form with Ajax, then you already need to make a page that handles the form variables. Make the `action` of the form point to the same page you call via Ajax.
- When submitting a form using Ajax, pass an extra parameter like '&ajax=1'. Your form processing page can then detect whether or not Ajax was used. If Ajax is used, send back some HTML, JSON, or XML data, and if not, do a redirect to another URL, or display a "Thank You" page (whatever makes sense).
- If there is a link that loads some HTML content using Ajax, again, pass a parameter like '&ajax=1' that tells the page to return only the core content as HTML, JSON, XML, or whatever. If the Ajax parameter is missing, the page can display the same content wrapped nicely in your site template.

Let's say, for example, that you have a listing of items, and beside each item there is a button that deletes the item. Start off by using a simple HTML form that goes to a delete page, like so:

```
<table id="itemlist">
    <tr>
        <td>Item 1</td>
        <td>
            <form action="confirm_delete/1/" method="post">
                <input type="submit" value="Delete"/>
            </form>
        </td>
    </tr>
</table>
```

Let's assume that "confirm_delete/1/" is a template that asks the user if they are sure they want to delete Item 1. If they say yes, they are redirected to a template "do_delete/1/" that actually deletes Item 1 from the database, and redirects the user back to the listing page.

Now, it may be annoying for users to have to go to another page, then reload the listing page every time they want to delete an item, and have to scroll back down to where they were in the list, and so on. Thanks to JavaScript and Ajax, we can hijack the forms and delete buttons and do something fancy instead.

To quickly review the behavior we will implement, we want the delete button to simply remove the table row containing the item, but at the same time, use Ajax to tell the server to delete the item. Asking for confirmation is still important, so we will pop up a confirmation to make sure the user wants to delete the item. Here is the JavaScript we can add to the page to do this:

```
// run this function when the page loads
addDOMLoadEvent(function(){
    // find the itemlist table
    var itemlist = document.getElementById('itemlist');

    // test if the itemlist is on the page
    if (!itemlist) return;

    // find all the forms in the table
    var forms = itemlist.getElementsByTagName('form');

    // loop over the forms
    for (var i=0;i < forms.length;++i) {
        // get each form element
        var form = forms[i];

        // make sure the action of the form contains 'confirm_delete.php'
        if (form.action.indexOf('confirm_delete') != -1) {

            // replace 'confirm_delete' in the action with 'do_delete'
            form.action = form.action.replace('confirm_delete', 'do_delete');

            // attach submit handler to the form
            addEvent(form, 'submit', function(e) {

                // pop up a confirmation
                if (confirm("Are you sure you want to delete this item?")) {
                    // if 'yes', we will do some Ajax and delete the row

                    // call the do_delete.php URL in the action attribute using Ajax
                    // this will actually delete the item in the background
                    httpRequest(this.action);

                    // find the row (parent of the parent of the form)
                    var row = this.parentNode.parentNode;

                    // remove row from its parent (a <tbody> in actuality)
                    row.parentNode.removeChild(row);
                }

                // prevent the actual submitting of the form
                // we do this whether or not the deleting was confirmed
                preventDefault(e);
            });
        }
    }
}
```

Add HTML Elements Using JavaScript

There are times when you really want to have JavaScript-only links and forms on a page, and don't feel the need to make an equivalent for when JavaScript is disabled. For example, you might want to add a link that toggles between hiding and showing a section. This kind of functionality is often just cosmetic and frivolous, and someone without JavaScript wouldn't miss it.

Rather than leave a nonfunctional hide link on the page, we can just add it using JavaScript. This way, when JavaScript is disabled, there simply is no hide link.

Adding a link with JavaScript is rather simple. Let's say this we want to end up with HTML like this:

```
<div id="content">
       <p>Some content that not everyone will want to see, blah, blah.</p>
       <a href="#" onclick="hideContent(); return false">Hide Content</a>
</div>
```

We can instead start off with HTML like this:

```
<div id="content">
       <p>Some content that not everyone will want to see, blah, blah.</p>
</div>
```

Then we add the link using DOM scripting, like this:

```
// create the link
var link = document.createElement('a');

// add parameters to the link
link.setAttribute('href', '#');

addEvent(link, 'click', function(e){
       hideContent();
       preventDefault(e);
});

// add text inside link
link.innerHTML = 'Hide Content';

// add link to the page
var content = document.getElementById('content');
content.appendChild(link);
```

That's it. Now, of course, when JavaScript is unavailable, there will, simply, be no link on the page.

You can do the same with forms, buttons, or anything else that can be found on a page. I won't go into depth with DOM Scripting, but there are excellent books and

resources available on the topic. I recommend you get familiar with the basics of DOM Scripting if you are working with JavaScript at all.

This brings up an important philosophical issue: which functionality should be supported when JavaScript is disabled, and which is frivolous?

When Is It OK to Have JavaScript-only Functionality?

This is really a personal judgment call, and every situation is different. Nonetheless, here is the philosophy I try to employ.

I think frivolous functionality is anything that is simply there to make a site look nicer, be more fun, or be easier to use. Essentially, anything that a user wouldn't necessarily notice is lacking or unavailable is something that can require JavaScript.

I'm thinking here about drag-and-drop shopping carts, the ability to change the design or layout of a page on the fly, hiding or showing form fields dynamically depending on what answers people provide, and so on. This could also include having tabs instead of one long page, or having a tree navigation that can be expanded and collapsed instead of one that is permanently expanded.

I would define the core content and functionality of a site as being anything that the user would severely miss, anything that when removed would severely limit the user's ability to interact with the site.

When I talk about content, I'm basically referring to any words or information on a site. Unless the content is relevant only to a user interface (instructions on how to drag-and-drop, for instance), then everyone needs to have access to this whether or not they have JavaScript. However, I wouldn't consider advertisements or other typically annoying things as content.

As for functionality, this would probably include adding comments on a blog, using a shopping cart to purchase something on an e-commerce site, or managing a list of items in a content management system.

It basically comes down to the user experience. Is this something people will need to see or interact with? Or is it something that is pure extra that people can live without? Ask yourself this and the answer should be relatively clear.

Write Special CSS for Users with JavaScript

Sometimes you'll need to change the way things look when JavaScript is enabled or disabled. Something that is draggable might get a move cursor when JavaScript

is enabled, but shouldn't have one when it is disabled. A table column header might look clickable when JavaScript is enabled, but simply be bold and plain when JavaScript is unavailable. Dynamic content that is hidden and shown using JavaScript has to be shown all the time when JavaScript is unavailable.

Be careful, though, not to simply hide JavaScript-only links, buttons, and forms using CSS alone. CSS can also be disabled, and some browsers and clients will end up displaying the JavaScript-only elements inappropriately.

There are quite a few techniques for using different CSS when JavaScript is enabled or disabled. They're all basically equivalent, so I'll go over a few so you can choose your favorite or come up with a new one.

Using document.write()

With this technique, you simply put a `<script>` block in the `<head>` of the page, and use `document.write()` to output a CSS `<link>` tag. It goes a little something like this:

```
<!-- CSS that is common to all, and some for those without JavaScript -->
<link rel="stylesheet" href="normal.css"/>
<script type="text/javascript"><!--
    // add a CSS document to the page that makes changes for JavaScript only
    document.write('<link rel="stylesheet" href="javascript.css"/>');
// -->
</script>
```

Here, normal.css will contain the bulk of the CSS, but also define how things look when JavaScript is disabled. javascript.css will only have a few rules that change things, and perhaps undo some of the styles in normal.css.

This method is quite easy, but not everyone likes the use of inline JavaScript and `document.write()`. You could easily put the JavaScript in an external file as well.

Using DOM Scripting to Change a `<link>`

This technique allows you to have a single, common stylesheet, and then have specialized stylesheets for both browsers with and without JavaScript. To do that, we start off with two `<link>` tags on the page, one for the common styles, the other for the non-JavaScript styles. Then we change the `href` of the second `<link>` tag to that of a JavaScript-only stylesheet.

Here's a simple example of this:

```
<!-- CSS that is common to all -->
<link rel="stylesheet" href="common.css"/>

<!-- CSS only for those without JavaScript -->
<link id="javascript_css" rel="stylesheet" href="no_javascript.css"/>
```

```
<script type="text/javascript">
    // first, check if the link tag is on the page
    var js_css = document.getElementById('javascript_css');

    // change the href of the link tag to point to JavaScript-only CSS
    if (js_css) js_css.setAttribute('href', 'javascript.css');
</script>
```

Again, quite simple. Like the last example, you could easily put this one line of JavaScript into an external file. You may also want to put it into a DOM Ready or onload function, but as long as the JavaScript is placed after the second `<link>` tag, this isn't necessary.

Using this method, you could also get rid of the first `<link>` tag pointing at common.css, and instead put the following at the top of both no_javascript.css and javascript.css:

```
@import "common.css";
```

Your choice will depend on how you like to organize your code.

Make Content Available with and without Ajax

Earlier, I alluded to keeping content reusable so that it can be easily accessed by your JavaScript and Ajax, as well as available through plain HTML. This makes it really easy to use Ajax to dynamically load content without breaking your existing site.

If all you do is load a chunk of HTML, then all you need to find out is whether it was accessed via Ajax or normal HTTP. If Ajax is being used, just show the content, and don't show the template (header and footer) of the page. If Ajax is not being used, show the entire page including the template.

To identify the use of Ajax, you may want to use a URL parameter, like 'ajax=1'. There are other techniques available, like setting an HTTP header in the request. This technique is used by many JavaScript libraries, including my `httpRequest()` function; they set a X-Requested-With header to 'XMLHttpRequest'. This header can then be detected by the server to determine if Ajax is being used.

In PHP, this might look something like this:

```
// detect if this request was called using Ajax
$IS_AJAX = (isset($_SERVER['HTTP_X_REQUESTED_WITH'])
              && $_SERVER ['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest');

// if Ajax is not being used, include the header file
if (!$IS_AJAX) {
    include "header.php";
```

```
}

echo "<p>Here is my content.</p>";

// if Ajax isn't being used, output the footer
if (!$IS_AJAX) {
    include "footer.php";
}
```

In other languages, this would certainly be different, and you may even want to use a single template file instead of using a header and footer (that's what I usually do, but this method made for a simpler example). In Ruby on Rails, you can simply check for this Ajax header with the built-in function `xhr?`.

Separate Presentation from Content with CSS

This is a big topic, and has been the basis of many books out there on web design using CSS, so I won't go into any depth here. However, I thought I'd list out a few ways you can separate the way a web page looks from the web page itself using CSS.

- Rather than use tables for layout, use a combination of floats, positioning, margins, and padding to place elements on the page.
- Stay away from deprecated or invalid HTML elements and attributes, like `<center>`, `<font>`, `<u>`, `bgcolor`, and `border`. Use the CSS equivalents instead.
- Instead of using ` ` and `<br/>` to add space around things, use CSS margins and padding.
- Use `border-top` or `border-bottom` instead of `<hr/>`.
- Whenever possible, remove decorative images from the web page, and put them into the CSS using `background-image`.
- Avoid having text that is only available as an image. This is a bad practice for accessibility, usability, and search engine optimization. If you want to use an image containing some text, you can put the text into an element and use image replacement like this:

```
<!-- HTML -->

<!-- We want to replace this <h1> with an image containing the title -->
<h1>My Great Title</h1>

/* CSS */

h1 {
       /* width of the replacement image */
```

```
            width: 200px;

            /* height of the replacement image */
            padding-top: 100px;

            /* push the text down (with the padding) and hide it under the overflow */
            overflow: hidden;
            height: 0;

            /* bring in the image to fill the box we've created */
            background: url(title_image.gif) no-repeat;
    }
```

- Use `overflow: auto` instead of iframes and framesets when you just want to get an area of the page to scroll.
- Avoid using class names that say too much about what the element is supposed to look like. A class name like "red-border" restricts you from being able to change the color scheme of your site. You'll either end up with "red-border" giving elements a green border (confusing), or you'll have to go through all the HTML and rename it to "green-border" (painful). Consider using semantic class names like "warning," "note," and "side-panel" instead.
- Put CSS in a style block or in an external file rather than in `style` attributes.

There are actually a few situations where some description of what things look like is appropriate to leave in the HTML, mainly when they serve a semantic purpose, or at least give a hint to the browser how to make things look when CSS is disabled or unavailable. These include:

- Use `size` attributes on `<input>` tags.
- Use `cols` and `rows` on `<textarea>` elements.
- Use `width` and `height` on `<img>` tags.
- Use `<strong>` and `<em>` when appropriate.

For the most part, though, stick with using CSS as often as possible.

Create a Separate Web Site for Those without JavaScript

This is the worst solution, yet it's often the first solution many people think of when faced with the problem of supporting those both with and without JavaScript.

Having two sites is a kind of brute force solution to the problem. Certainly having two separate sites with unique requirements would work, but it would also take a lot of work to accomplish.

When regular software developers develop an application to work on multiple platforms (Windows, Mac, and Linux, for example), they rarely rewrite the entire application from scratch. Instead, they reuse as much as possible, and isolate the differences between platforms to the smallest piece of code.

With JavaScript and non-JavaScript, there is an even greater opportunity to reuse functionality than when developing a desktop application for multiple platforms. There's much we can do to take advantage of the HTML base that any JavaScript-enhanced site is based on, and reuse as much as possible between a site enhanced with JavaScript and Ajax, and one without such enhancements.

Nonetheless, creating two separate sites remains a last resort solution. I would never take this path, and would rather not use JavaScript at all than maintain two versions of the same site. I just felt I had to mention it, but only because I hear it mentioned so often.

In summary, there has always got to be a better way to support non-JavaScript users than creating an entire separate site from scratch.

Summary

- You probably don't need to convince anyone to start building sites unobtrusively — as a developer, it's your responsibility to decide this.
- Developing unobtrusively is easiest if you start off without using any JavaScript, CSS, or Flash, and add these later.
- Links and forms are the building blocks of the Web and should be respected.
- Elements like links, buttons, and forms that only make sense when JavaScript is enabled should be added to a page using JavaScript.
- You can have different CSS for those with and without JavaScript.
- You should reuse server-side logic between your basic and Ajax-enabled web site components.
- You don't need to create separate web sites for those with and without JavaScript.
- Take advantage of the JavaScript libraries, functions, and techniques others have developed to be able to program more unobtrusively.

It's been a long chapter, but by now you should have the tools in your hand to go out and start developing unobtrusively. Next, I will be looking at some more complex examples, showing how to apply the techniques above to real life situations.

Examples

It's time we really got into some more complex examples of unobtrusive development. Here, I'll go over a few typical web development challenges, complete with some fairly heavy code examples. I think it will be worth it, though, so you can see how the practice of unobtrusive development can be applied to very typical development scenarios.

These examples will be written in PHP (because it's rather common, and it's what I'm comfortable in, and I don't think anyone would benefit from trying to understand my pseudocode), but I won't be doing anything very PHP-specific, so you should be able to read along and apply the concepts to any language, even if you haven't seen PHP before in your life.

These examples will also take advantage of the JavaScript functions discussed in the previous chapter. Feel free to replace them with the equivalents in your JavaScript library.

Dynamic Ajax Tabs

Thanks to Ajax, we can make a web page look and work similar to a desktop application, complete with tabs.

Tabs were possible in the Web before Ajax, too. They just acted different. When you clicked a tab, the page reloaded, the tab was highlighted, and the appropriate content was displayed. With Ajax, the tab highlight can change with the content without reloading the page. In this example, we will get both working on the same page using Unobtrusive Ajax.

We'll start with our data—in this case, the pages themselves. A page just consists of a title and a body. Typically this will come from the database, but for our simple example, we'll just make the data two static PHP arrays:

```
// web page content data
$tabs = array("Home", "Work", "School", "Garden", "Pub");
$tab_content = array(
    "There's no place like Home.",
    "All work and no play makes me a dull boy.",
    "School's out for summer!",
    "InAGaddaDaVida!",
    "Cheers to the beers!"
);
```

We will use the first array for the labels of the tabs. The second array will be the content displayed on the page.

Next, we need to set up the default content for the page. Let's just use the first tab —in this case,"Home." We will make a few variables that point to the active tab title and content, and initialize them to the defaults:// default values

```
$active_tab = $tabs[0];
$content = $tab_content[0];
```

Next, we have to think about what the URLs will look like. If the template of the page is index.php, then we can just pass a URL parameter to this page containing the ID of the tab we want. This way, index.php?tab=2 will bring up the "School" tab.

```
// if other tab is requested, set content values
if (isset($_GET['tab'])) {
    // get tab index
    $index = $_GET['tab'];

    // find tab content
    $active_tab = $tabs[$index];

       // set active content
    $content = $tab_content[$index];
}
```

At this point, $active_tab will contain the name of the tab (defaulting to the first tab), and $content will contain the content of the tab (also defaulting to the first tab). If the URL parameter "tab" is given, that tab will be loaded instead. Now we can display the tabs and the content on the page:

```
<!-- Use an HTML list for the tab navigation -->
<ul id="navigation">
<? foreach ($tabs as $i => $tab): ?>
       <!-- if this tab is active, add the 'active' class -->
       <li
              <? if ($tabs[$i] == $active_tab): ?>
                     class="active"
              <? endif; ?>
       >
              <!-- wrap the tab label inside a link to the tab page -->
              <a href="index.php?tab=<?= $i ?>"><?= $tabs[$i] ?></a>
       </li>
<? endforeach; ?>
</ul>

<!-- print out the content -->
<p id="content"><?= $content ?></p>
```

Now, everything looks great! Well, it doesn't look great, the code looks great, but the HTML page looks like a list followed by a paragraph. But nonetheless, the site

works. You can click the links and the content will change. We need to add some CSS so this actually looks like tabs:

```
/* styles for the UL tab navigation */
#navigation {
    padding: 0;
    list-style: none;

    /* push down 1 pixel so the border of the tabs overlap the content border */
    position: relative;
    top: 1px;
}

/* styles for the LI tabs */
#navigation li {
    float: left;
    padding: 5px 10px;
    border: 1px solid black;
    margin: 0 5px 0 0;
}

/* styles for the active tab */
#navigation li.active {
    /* make the bottom border white so the tab looks attached to the content */
    border-bottom: 1px solid white;
}

/* styles for the P content */
p {
    clear: both;
    width: 300px;
    height: 100px;
    border: 1px solid black;
    padding: 50px;
}
```

Great. Now everything looks and works perfectly (**Figure 1**), without a single curly bracket of JavaScript. If we wanted to, we could stop here, and we wouldn't hear a single complaint from our visitors (but let's wow them with some Ajax anyway).

Our Ajax will need to have access to the content in the tabs. Let's reuse the functionality on our page, and just add a little extra for the Ajax. How about we pass another URL parameter, 'ajax=1', which tells the page to simply dump out the content instead of the entire HTML page. This is rather simple:

```
// if the ajax parameter is given, echo the content and stop processing the page
if (isset($_GET['ajax'])) {
    echo $content;
    die;
}
```

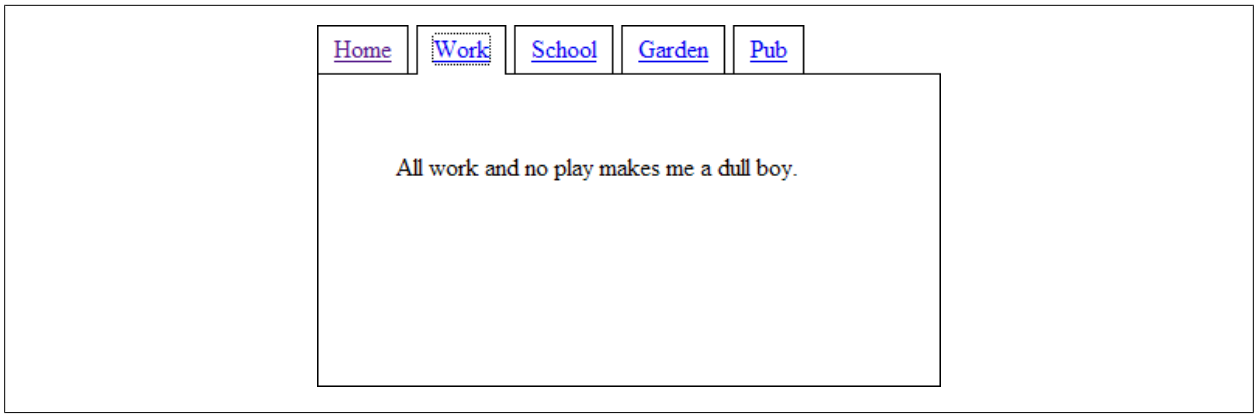

Figure 1. The tab navigation looks good and works without any Ajax added yet.

Next, of course, we need some JavaScript. We will find the navigation tabs, and override the behavior of the links so they load the content with Ajax, change the active tab class, and update the content on the page:

```
// change the text inside the content area
function setText(text) {
    // find content area - if it doesn't exist, stop function
    var content = document.getElementById('content');
    if (!content) return;

    content.innerHTML = text;
}

// this function is called when a tab link is clicked
function linkClickAction(e) {
    // find surrounding ancestor elements
    var li = this.parentNode;
    var navigation = li.parentNode;

    // reset any currently active tab
    var tabs = navigation.getElementsByTagName('li');
    for (var i=0;i < tabs.length;i++) {
        // remove the 'active' class from all tabs
        removeClassName(tabs[i], 'active');
    }

    // add the 'active' class to the clicked tab
    addClassName(li, 'active');

    // set temporary loading message
    setText('Loading...');

    // request content from server, adding the ajax=1 URL parameter
    httpRequest(this.href + '&ajax=1', setText);

    // stop browser from following link
```

```
    preventDefault(e);
}

// run this function when the DOM loads
addDOMLoadEvent(function() {
    // find navigation - if it doesn't exist, stop function
    var navigation = document.getElementById('navigation');
    if (!navigation) return;

    // attach event to every link in the navigation
    var links = navigation.getElementsByTagName('a');
    for (var i=0;i < links.length;i++) {
        // add the click handler to each link
        addEvent(links[i], 'click', linkClickAction);
    }
});
```

That's a lot of code to swallow in one gulp, but let me try to describe the flow. At the end of the code, we register a DOM Load function with `addDOMLoadEvent()`. When the page loads, this function is called, finds the navigation, and adds a click event to each link in the tabs.

When a tab is clicked, the 'active' class is removed from all the tabs in the navigation, then added to the tab that was clicked. The content area is changed to 'Loading...' (using a simple `setText()` function), and the content is loaded via Ajax, using the URL of the link, but adding 'ajax=1' to the end. `setText()` is passed as a callback function, so when the content is passed back, it immediately goes into the content area. Lastly, we prevent the browser from following the link with the `preventDefault()` function.

Once this script is added to our page, the tabs will be loaded using Ajax without needing to reload each time. If JavaScript is disabled, or if our script breaks, the old functionality will still be there and work fine. Search engine spiders will also be able to find each page the old way, so someone searching on 'School' may be able to link directly to the complete 'School' page.

Also worth noting are the checks for elements on the page. The script doesn't assume that there is a 'navigation' or 'content' element on the page. It checks first. This way, if we remove these elements, or attach the script to a page without the tabs, it won't break, it just won't do anything.

That's it for the first example! Not so bad, was it? We didn't have to make two separate sites, we only had to write a few lines of PHP extra and provide real values for the link `href`s. The amount of JavaScript we had to write was the exact same as we would have to when no JavaScript was enabled. Plus, we get all the benefits of accessibility and unobtrusive development!

Hiding and Showing Page Sections Dynamically

This example is quite similar to tabs, except for an important difference: all the content is on the page, and the JavaScript is just used to show and hide sections without loading anything extra from the server. You may have seen this done as an Accordion interface, or a number of other user interfaces. In this example, I will use an interface with a small table of contents which displays the different sections.

At this point, it helps to think about how the page will work with JavaScript enabled or disabled. When JavaScript is enabled, we only want to see one section at a time. Clicking the section links will hide a section and display another.

With JavaScript disabled we can't do the same functionality without reloading the page. This is an option, but not really necessary. Since all the content is already on the page, we can just show all the sections at once and use the section navigation as a table of contents, jumping down to the section using link anchors. We should also then include a "Back to top" link that jumps back to the navigation. This should be perfectly acceptable interface to anyone using the site, even if it doesn't have the fancy showing/hiding that JavaScript provides.

As usual, we'll start off building the non-JavaScript version first. Let's start off with some simple HTML containing our sections:

```
<ul id="nav">
       <li><a href="#section1">Section 1</a></li>
       <li><a href="#section2">Section 2</a></li>
       <li><a href="#section3">Section 3</a></li>
       <li><a href="#section4">Section 4</a></li>
       <li><a href="#section5">Section 5</a></li>
</ul>

<ul id="sections">

<li id="section1">
       <h2>Section 1</h2>
       <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</p>
       <p class="backtotop"><a href="#nav">Back to top</a></p>
</li>

<li id="section2">
       <h2>Section 2</h2>
       <p>Aenean pharetra lacus. In vel quam eu odio sodales lobortis.</p>
       <p class="backtotop"><a href="#nav">Back to top</a></p>
</li>

<!-- Sections 3-5 go here -->

</ul>
```

I've shortened the content in each section, and only included two sections for brevity, but it should be obvious how this would keep going.

At this point, the page is perfectly usable. It doesn't look so great, but at least the content in each section is available, and the section links and "Back to top" links both work fine.

Let's make this look a bit nicer with some simple CSS:

```
/* the section navigation / table of contents */
#nav {
       /* put the navigation on the left */
       float: left;

       /* frame it as a box of links */
       border: 1px solid black;
       list-style: none;
       padding: 10px;
       margin: 20px 0;
}

/* the sections container */
#sections {
       /* put beside the navigation */
       float: left;
       list-style: none;
       width: 75%;
       padding: 0;
       margin: 20px;
}

/* frame each section item as a box */
#sections li {
       border: 1px solid black;
       padding: 20px;
       margin: 0 0 20px 0;
}

/* give the "Back to top" links some space, and make them smaller */
#sections .backtotop {
       font-size: 0.8em;
       margin-top: 20px;
}
```

Now things look a bit nicer, and the site is very much usable (**Figure 2**). Of course, our boss asked for that fancy Ajax stuff, so we'll have to add it. Plus, we probably want to impress our users.

The JavaScript will need to hijack the links in the table of contents. Clicking each link will hide all the sections and show the relevant section. We will be able to

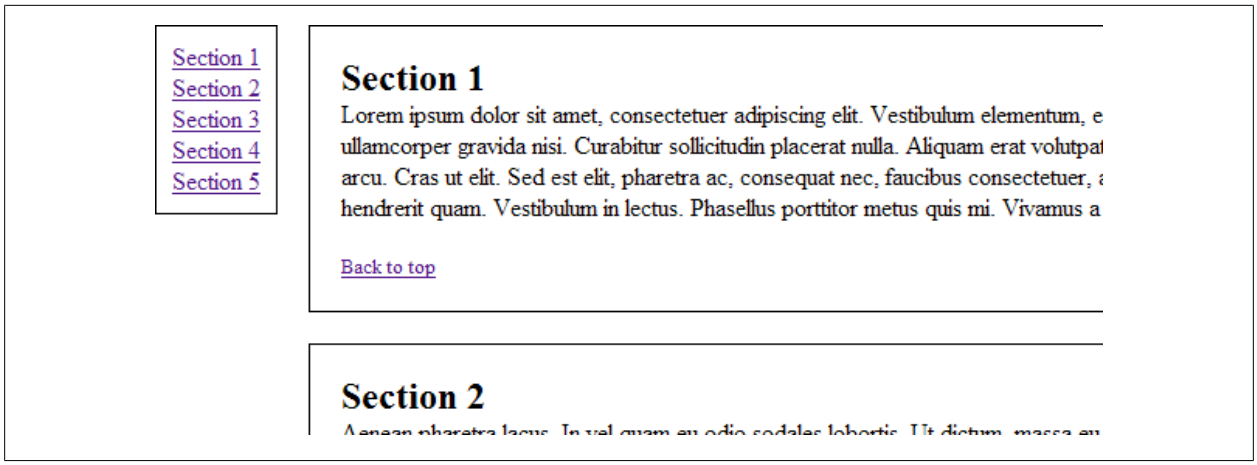

Figure 2. The sections are all visible, and the navigation functional, without JavaScript.

deduce which section to show from the `href` of each link (it has an anchor which contains the `id` of the section). We will put the whole thing in a DOM Load event handler so it only executes when the page has loaded.

But wait, which section do we show first? We could default to the first one, but what if someone has bookmarked our URL with an anchor, like "index.html#section2"? We can get clever here and have a look at the current URL to see if an active section anchor is there. If so, we will use it as a default, and if not, we will default to the first section:

```
addDOMLoadEvent(function() {
	// find nav and sections, if they don't exist, return
	var nav = document.getElementById('nav');
	var sections = document.getElementById('sections');
	if (!nav || !sections) return;

	// find actual sections
	var lis = sections.getElementsByTagName('li');
	if (!lis.length) return;

	// see if page url suggest a section should be active
	var activeSection;
	if (window.location.hash == '') {
		// no section selected, make first one active
		activeSection = lis[0];
	} else {
		var id = window.location.hash.substring(1);
		activeSection = document.getElementById(id);
	}
	addClassName(activeSection, 'active');
	var links = nav.getElementsByTagName('a');
	for (var i=0;i < links.length;i++) {
		var link = links.item(i);
		addEvent(link, 'click', function(e) {
```

```
                    // remove the 'active' class from all sections
                    for (var j=0;j < lis.length;j++) {
                          var li = lis.item(j);
                          removeClassName(li, 'active');
                    }

                    // add the 'active' class to the section linked to by this link
                    var id = this.href.substring(this.href.indexOf('#') + 1);
                    addClassName(document.getElementById(id), 'active');
              });
       }
});
```

Notice that I didn't use `preventDefault()` at the end of the click handler. This way, the URL of the page will change to include the anchor, like "index.html#section4."This way, people can bookmark a particular section, and our fancy JavaScript will display that section automatically. A browser without JavaScript visiting the same link will simply jump down to that section using traditional HTML anchors.

I've also introduced an 'active' class that we don't have in the CSS yet. It didn't make sense in the CSS until now, because with JavaScript disabled, every section is 'active' in a sense. The 'active' class will indicate which section to show, and the rest of the sections will be hidden automatically. It looks like we will need to have some special CSS for when JavaScript is enabled.

While we're at it, we can hide the "Back to top" links, because when JavaScript is enabled, users will always be at the top already.

We can link in our JavaScript-only CSS by using a simple `document.write()` somewhere in our JavaScript file (but **not** inside the `addDOMLoadEvent()` function):

```
document.write('<link rel="stylesheet" href="javascript.css"/>');
```

Inside javascript.css, we can put our JavaScript-only CSS changes:

```
/* hide all sections by default */
#sections li {
       display: none;
}

/* display only the active section */
#sections li.active {
       display: block;
}

/* hide all the "Back to top" links */
.backtotop {
```

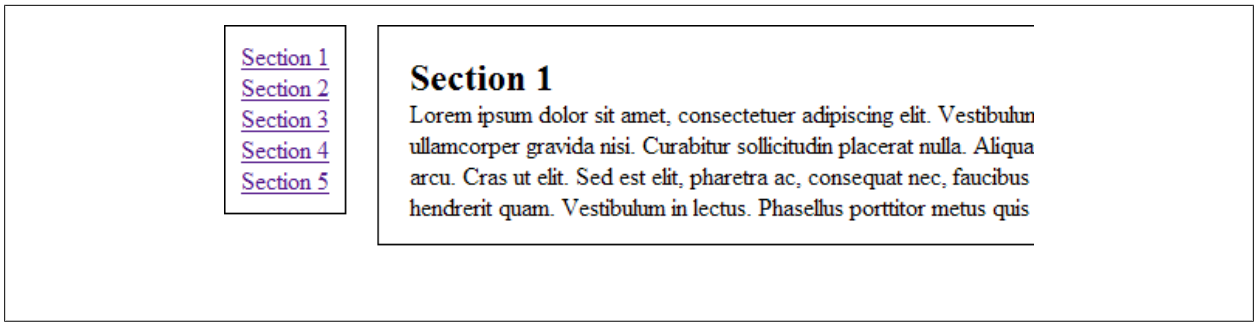

Figure 3. One section displayed at a time, thanks to the power of JavaScript.

```
        display: none;
}
```

Great. Now only one section is shown at a time when JavaScript is enabled, but all the content is still available when JavaScript is disabled. Even the URLs with anchors work in both scenarios. Everyone wins, and we barely had to write any extra code to get it to work for everyone. All we had to do was choose how we pieced things together, and be careful to keep both scenarios in mind the whole time (**Figure 3**).

Dynamic Select Boxes

Another common use of JavaScript is to have two select boxes linked—changingthe first select box changes the contents of the second select box. You often see this with relationships like country and province/state.

It's not really possible to get this to work without JavaScript, at least not the dynamic changing of select boxes. You could show just the first select box, allow the user to submit the form, and only then display the second box with a message that they should use it.

Instead of doing this, you could also let users without JavaScript use the form by putting all the values into a single, huge select box. This only works if the amount of data is small enough that you're okay with putting everything on the page. It may not be reasonable to put every country and province/state on a page, but for smaller amounts of data, this may be a very reasonable option.

We'll start off with a long, single select box like this:

```
<select id="subcategory" name="subcategory">
        <option value="1">Animals - Birds</option>
        <option value="2">Animals - Fish</option>
        <option value="3">Animals - Mammals</option>
        <option value="4">Animals - Other</option>
        <option value="5">Colors - Blue</option>
        <option value="6">Colors - Red</option>
        <option value="7">Colors - Purple</option>
```

```
        <option value="8">Colors - Yellow</option>
        <option value="9">Colors - Other</option>
        <option value="10">Food - Meat</option>
        <option value="11">Food - Fruit</option>
        <option value="12">Food - Vegetables</option>
        <option value="13">Food - Junk Food</option>
        <option value="14">Food – Other</option>
</select>
```

We can make the interface easier for people with JavaScript by splitting this up into two select boxes, so they don't have to scroll down a long, single select box (okay, this isn't very long, but pretend it is).

All we really need is the value of the second select box. The first select box will only contain the categories, and we don't need to invent a numeric value for "Colors,"for example.

To do this, we will loop over the options, parse the text inside each of one, and split it up by the " - ". We will build a JavaScript data model inside an array, and use that to populate the second select box each time the first changes. We will create a new select box containing only the categories, and use this as the first select box. We will use the original select box as the second one.

We will also be clever enough to check if an option has already been selected. If it is, we'll need to make it selected between the two select boxes as well, and this means prepopulating the second select box accordingly:

```
// store the option data model in here
var category_data = [];

// run this function when the page has loaded
addDOMLoadEvent(function() {
        // find subcategory select
        var subcategory = document.getElementById('subcategory');
        if (!subcategory) return;

        // check for an existing value
        var old_value = subcategory.options[subcategory.selectedIndex].value;

        // populate data model and new select box based on old one
        var category = document.createElement('select');
        category.id = 'category';
        var last = '';
        for (var i=0;i < subcategory.options.length;i++) {
                // split up label into category and subcategory
                var label = subcategory.options[i].text.split(' - ');

                // add data to array
                category_data.push({
```

```
                'id': subcategory.options[i].value,
                'main': label[0],
                'sub': label[1]
        });

        // if this is a new category, create a new option in the first select
        if (last != label[0]) {
                last = label[0];
                var option = document.createElement('option');
                option.text = label[0];
                category.add(option, null);
        }

        // if selected subcategory belongs to this category, select it
        if (subcategory.selectedIndex == i)
                category.options[category.options.length - 1].selected = true;
    }

    // add category select to page as first select box
    subcategory.parentNode.insertBefore(category, subcategory);

    // update second select box to match contents of first
    updateSelects();

    // add event to update second select box anytime first changes
    addEvent(category, 'change', updateSelects);

    // reset the selected 2nd-level select box item, if necessary
    for (var i=0;i < subcategory.options.length;i++) {
        var option = subcategory.options[i];
        if (option.value == old_value) {
            option.selected = true;
            break;
        }
    }
})

// this function populates the second select box based on the value of the first
function updateSelects() {
    // find both select boxes
    var category = $('category');
    var subcategory = $('subcategory');
    if (!category || !subcategory) return;

    // clear out select box
    subcategory.options.length = 0;

    // populate with sub-categories for this category
    for (var i=0;i < category_data.length;i++) {
        var item = category_data[i];
```

```
            if (item.main == category.options[category.selectedIndex].text) {
                var option = document.createElement('option');
                option.text = item.sub;
                option.value = item.id;
                subcategory.add(option, null);
            }
        }
}
```

That was a big chunk of code, but it was worth it. We now have a great interface with a select box that changes a second select box dynamically, yet the form is still perfectly functional when the JavaScript breaks or is unavailable.

This example might not be the most practical, but it demonstrates how there is nearly always a non-JavaScript alternative for any JavaScript interface, and the two can often easily coexist.

If we were to build this interface as relying on JavaScript, we would still need to pass the data model to the JavaScript somehow. It isn't really more work to do this in the select options themselves. We would still need to deal with default values, so really the only new code we had to write was the parsing of the options, building a data model from it, and putting a second select box on the page. This is a small amount of work to make the site work for everyone, and a far cry from building a separate web site in order to be accessible.

Other Real-Life Examples

I hope by now you are starting to get the hang of the attitude and logic behind building interfaces unobtrusively. Rather than walk you through any more code examples, I thought I would just discuss briefly some common real life interfaces and functionality, and suggest a way you would design these unobtrusively.

Search Suggestions

Google Suggest was probably the first famous example of using Ajax to enhance a traditional interface. It's also the easiest technique to make unobtrusive. With JavaScript enabled, you get search suggestions on the fly. When JavaScript is unavailable, you simply get a plain search box without suggestions.

Click to Vote/Rate

I don't know where this first appeared, but it seemed to appear everywhere at once. Being able to click one of five stars to rate an item is a great use of Ajax, because it doesn't need to reload the page in order to record the vote.

To keep this unobtrusive (because you don't want to lose votes unnecessarily), we can still start off building it as five links. Clicking each link goes to a page which

records the vote, then forwards back to the original page. However, with JavaScript enabled, clicking the link can call the same page using Ajax and dynamically color the stars to show that the vote has been recorded.

Maps

Certainly Google Maps was another very early example of a great use of Ajax, and definitely the first one that made everyone scratch their head and rethink the way web interfaces were supposed to work.

If you ever find yourself building a map interface, try to combine the two interfaces into one. Clicking a button to zoom in, with JavaScript disabled, would reload the page zoomed in. Clicking the same button with JavaScript enabled would redraw the map dynamically using Ajax. The same for all the controls on the map. No extra logic or special interface needed.

Drag-and-Drop Sorting

How did we used to do sorting on web pages before drag-and-drop? With boring old "Move up" and "Move down" buttons, of course. Yes, there's nothing more painful than reordering a list of items when the page refreshes between each click. But this doesn't mean you can't build this functionality in as a backup, and add the fancy drag-and-drop stuff on top.

Don't hide these buttons unnecessarily, either. Someone might be using your web site that can't use a mouse, and these buttons will be the only way they can sort the list. Other users might not be familiar with drag-and-drop on the Web. For these users, you could hijack the buttons so they resort the item using Ajax instead. This way, everyone gets the best experience possible.

Form Validation

I've mentioned this already, but it's worth repeating. If you need to validate the values of a form, do it on the server. Otherwise, you won't be able to trust the values you get. You can still add client-side JavaScript validation to improve the user experience, but you can't assume that JavaScript will always be enabled. As I mentioned, I often disable JavaScript when using forms so that I can get around validation, and therefore don't have to answer questions that would otherwise be mandatory.

Pop-up Windows and Layers

First, let me say, I hope to never see a pop-up window ever again.

Having said that, if you really need to have pop-up windows or layers for some obscure reason, you can still put `target="_blank"` on a link. True, you won't be

able to control the size of the window, but at least it will work. Use JavaScript to hijack the link and make it launch a pop-up window or layer instead.

Sortable Table Columns

JavaScript can make for some great table interfaces by letting you click the column headers to sort the table. What happens if JavaScript is disabled?

Depending on how important sorting is, you could either display an unsortable table to those without JavaScript. Or, you could make the column headers clickable, but clicking them would reload the page with the table resorted. Then, you can hijack these links and add in the fancy dynamic table sorting with JavaScript and Ajax.

Tree Navigation

If you have a tree navigation with collapsible and expandable levels, you could just have it completely expanded by default, and use JavaScript to collapse the tree and make the tree nodes clickable. You could also have it collapsed, but reload the page on each click to expand a tree node. However, I think this would be a painful user experience, and I'd rather have to scroll a lot than wait for the page to refresh a dozen times.

Ajax Form Submitting

This one is easy. If you want to have a form submit in the background with Ajax, simply make sure that it can submit somewhere regularly when JavaScript is disabled. Hijack the form and add your fancy Ajax and JavaScript to update the page dynamically every time the form is submitted.

Summary

- Before building an interface that uses JavaScript, think about how it would be built without JavaScript.
- Build it without JavaScript first, keeping in mind the fancy functionality you will add to it later.
- Nearly every Ajax and JavaScript interface has some kind of non-JavaScript equivalent, and the two can nearly always coexist on the same page.

Conclusion

Well, we've finally come to the end of the book, and the beginning of the world of Unobtrusive Ajax development. There will always be challenges for every web developer to maintain the principals of unobtrusive development, but remember that there is always a solution available, and often easier than it seems.

Armed with the techniques in this book, start programming unobtrusively as often as you can. Your bosses and clients probably won't even notice the difference (unless they have JavaScript disabled), but a lot of other people will.

Ajax interfaces will become more common in the future, and it will be harder to argue against making JavaScript a requirement for every web site and application. It will also be more important for web developers to take a stand and ensure that the Web remains a place that everyone can use, no matter what technical or physical limitations they face.

I'd like to thank you very much for reading along. Just showing an interest in Unobtrusive Ajax is really the biggest step towards making the Web a better place for everyone. Nobody's perfect, so just try your best, and more importantly, have fun!

--Jesse Skinner