jQUERY *for* DESIGNERS

MEAP

Remy Sharp

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1

Start using jQuery, without knowing anything!

Welcome to *jQuery for Designers*. If you've picked up this book it's likely you're a front end web designer, interaction designer or some designery/"front end" type that works on the web and wants to get going with jQuery pretty quickly without necessarily having to understand what's going on behind the scenes. It's a bit like when I buy a new computer: sure, I want it to have lots of memory and hard drive space, but I don't really care how it works under the hood, I just want to work, and look good when it works. If that's what you want out of jQuery, then this book is for you.

The great thing about jQuery is that without much knowledge of JavaScript, you can create impressive effects to wow your boss or clients, and it's relatively simple to add new and interesting ways to interact with your site visitors, often with very little code. All this *and* you can still creative standards based web sites that are "*unobtrusive*" (which I'll explain in chapter 2).

Throughout this book I'll try to show you real world problems and show you how they can be solved using jQuery. I won't try to explain how jQuery is achieving these techniques under the hood, but you will have all the knowledge you need to be able to create some impressive interactions and effects in the browser.

I've met a number of individuals that have asked me how to create a simple effect for the browser, but they absolutely did not want to write any code. To create the effect they needed some code, even if just a little. If we used jQuery to solve their problem, we actually needed very little code, something like one or two lines that I'll show you in this chapter.

jQuery does a lot of the heavy lifting for you and doing something as simple (as you might think) as adding new options to a select drop down element is tricky without jQuery, because of differences between how `option` elements are inserted by browsers. jQuery

handles all these tricky areas for you and reduces the complexity of solving (what should be) simple problems. You will need to code *something* but the effort required is very little for those simple tasks – exactly as it should be!

By the end of this chapter we will have dived in to some code to show you how easy it is to add simple effects to a web page, but don't worry if code puts you off, I'm going to walk through each stage of the process so you understand what goes where and how it fits together.

Introducing jQuery

jQuery is a JavaScript library. It's not another language, though on initial review it might look like it is. jQuery is a *DOM scripting* library, designed primarily to help you manipulate web pages without the hassle that typically comes with DOM scripting.

The DOM

DOM scripting is what we developers do when we're manipulating web pages using JavaScript in typically (or hopefully) an unobtrusive fashion (a term and concept I'll explain in Chapter 2).

The Document Object Model or DOM for short, is what browser sees your web page as after it's processed the markup you give it. Once loaded in to the browser, we, the developer, are able to manipulate and change the DOM using JavaScript, which could change what we see in the browser window.

It's important to remember that the DOM represents your markup *after* the browser has processed it, and this means it doesn't always match the raw HTML exactly.

When we refer to the DOM, we're referring to the document that can be seen in the browser window.

Out of the box jQuery provides the following features:

- Easy selection and targeting of elements on the page. jQuery uses CSS to navigate around the DOM to let you target a specific area of the document
- Manipulation of the page and its elements. jQuery makes it easy to modify the DOM, insert new markup, modify element classes and a great deal more
- Handling user interaction. jQuery allows you to run a specific action when a specific event occurs on a specific element, such as a user clicking (the event) on a link (the element)
- Animations. jQuery provides simple animation effects to show and hide elements as well as the ability to create your own custom animation whereby you have complete control over all of the animating properties
- Ajax. jQuery makes Ajax incredibly simple if you just want to inject content from

another web page, or include content from your favourite social media web site (such as Twitter), or if you want to perform a more complicated task exchanging data between your web site and the server, jQuery offers a way to take complete control of Ajax requests

- Cross browser - jQuery provides A-grade cross browser support, so you don't need to worry about how IE6 may handle DOM manipulation differently to Firefox

A-Grade Browser Support

Yahoo! provides a table of "A-grade" browsers, see in the figure below, on different platforms (such as Windows XP or Mac OS X) which is regularly reviewed. A-grade browsers are generally ones that are either majority share holders in the browser market or are take full advantage of modern standards.

	Win XP	Win Vista	Mac 10.5.†	Mac 10.6.†
Firefox 3.0.†	A-grade			
Firefox 3.5.†	A-grade	A-grade		A-grade
Opera 10.0.†	A-grade			
IE 8.0	A-grade	A-grade		
IE 7.0	A-grade	A-grade		
IE 6.0	A-grade			
Safari 4.0.†			A-grade	A-grade

The figure above should be part of the sidebar

The latest table of support can be seen online: http://developer.yahoo.com/yui/articles/gbs/

jQuery (at time of writing) supports the list of browsers that follow, and each release of jQuery is automatically tested on 50 different browsers and 11 different platforms (Windows XP, Windows Vista, etc). jQuery officially supports the following versions of browsers *and all successive browsers*: Internet Explorer 6.0, Firefox 2, Safari 3, Opera 9 and Chrome.

Back in the early days of my web development career there weren't any frameworks or libraries I could use to help make my job of contending with the DOM and doing fancy JavaScript tricks easier. Certainly no libraries that had the benefits and success that jQuery

has today. There's a number of factors that make jQuery a good choice and successful JavaScrpit library:

1. By wrapping a lot of the donkey work up in a library, DOM manipulation, the cruft that comes with effects, how Ajax works and so on, it makes web development simpler and quicker.
2. Since jQuery has such good cross browser support, hiding the issues particularly with older non-standards based browsers like IE6 it takes away the headache we used to have when dealing with issues ranging from the simplest of things like setting opacity to much more subtle and complicated browser specific bugs.
3. jQuery doesn't discriminate against front end designer or developer: because it uses the CSS syntax as its method of working with the document. CSS should be our working day bread and butter, so it doesn't mean we have to learn some new method to navigate and target specific areas in the DOM.
4. jQuery has a particularly easy method of extending the library. This means that there are literally thousands of jQuery plugins available to solve any number of different tasks. The jQuery foundation provides its own library of plugins provide custom UI element, such as datepickers and accordions.
5. The community around jQuery. There's a huge number of blogs and tutorials, screencast, conference sessions and workshops, twitter accounts, chat channels and jQuery even runs it's own dedicated conference. The more the community puts information out on the web about jQuery, the easier it is to learn and problem solve using jQuery.

If you or your manager or boss needed more convincing that jQuery was a good choice for your web site, there's also proof in the pudding. According to http://backendbattles.com, who has gone through the Alexa top 10,000 web sites and catalogued a number of statistics, including any JavaScript library included, jQuery is included in over 21%[1] of the top 10,000 web sites, and the next JavaScript library is less than 5% of the top 10,00 web sites. Another web site that tracks how web sites are built, http://builtwith.com has recorded that jQuery is in over 35% of all their catalogued web sites, including: whitehouse.gov, twitter.com, ibm.com, espn.com, wordpress.com and time.com to name but a few.

Getting help beyond this book

Following list should be part of the sidebar

[1] http://www.backendbattles.com/JavaScript_Libraries

1. The jQuery forums (previously hosted on Google Groups, now a dedicated forum): http://forums.jquery.com which has categories including Using jQuery, Using jQuery Plugins and jQuery for Designers to name a few.
2. #jquery on irc.freenode.net IRC channel. Often 300-400 users attending, with many friendly and experienced developers ready to help answer all types of questions relating to jQuery. If you're not familiar with IRC, it's simply a chat room. You can download specific software to access the chat rooms or you can access it via the web using the following: http://webchat.freenode.net/?randomnick=1&channels=jquery
3. The jQuery documentation is available in several different forms. Most of these documentation sites include practical and live examples of how each function works – which is perfect for experimentation:
    a. Primary documentation: http://api.jquery.com
    b. Visual categorisation: http://visualjquery.com
    c. Online app and offline Adobe Air browser: http://api.jquery.com/browser
    d. And several other types, including even an iPhone documentation browser!
4. @jquery, @jqueryui and @jquerysites are the official Twitter accounts that announcing tutorials, tips and new resources to learn about jQuery.
5. Finally http://jqueryfordesigners.com that has a growing collection of free screencasts for beginners hosted by yours truly.

Creating your first jQuery effect

Later on in this book I'll walk you through specific problems and explain how we're solving it. In particular, we'll be looking for a common pattern in the problem so that you'll recognise how to solve similar problems in the future.

However, for this first chapter, I'm going to let you dip your toe in the water without getting you too wet. In fact, I'm not even going to explain what's going on. You'll be able to copy and paste this code and solve what's quickly becoming known as the "Hello World" of jQuery: showing and hiding content.

Showing and hiding is an interactive effect that without jQuery could be relatively complicated, certainly if you're not comfortable with JavaScript. As I said, you'll be able to copy and paste this solution, so I won't explain the code blow-by-blow, but I will explain in plain English what we're trying to solve and how.

For the example I'm going to take you through a little application I've built that helps me search Twitter. There's certain parts that are only accessible if the user has logged in to Twitter, so I needed a way to say: "this part of the app is only available if you're logged in". I chose to solve this by showing an overlay, which is what we'll look at next.

The effect of showing an overlay isn't, I'm afraid to say, going to blow your minds. It's not that impressive as an effect. On the other hand, it is a fundamental effect and extremely common on web sites today. Moreover the example is designed to show you how incredibly

easy it is to execute a simple effect in a matter of three lines of code. It's this simplicity that makes jQuery mind blowingly simple!

Tip: When problem solving, try to forget code for a moment

When I'm trying to solve how some cool effect works on a web site I've just seen, I'll talk through the problem, out loud, and in *plain* English – i.e. not mentioning code at any point.

If you can explain the problem in plain language and without referring to "slideDown" or "the dollar function" then it's highly likely that you've got a firm grasp on the problem, and also the solution, even if you don't know exactly what bit of code you specifically need.

I've even heard of some offices having a teddy bear in the corner, that if a developer gets stuck, they had to first explain their problem to the teddy before they're allowed to request help from their line manager or peers. I've lost count as to the number of times I've started explaining a problem to a colleague, and half way through the explanation had the eureka moment.

So when you want to solve a problem, forget code for a moment, and solve it using plain English first.

Real world example: Snap Bird – showing the redirect warning

Snap Bird, http://snapbird.org, is a real web app that helps search Twitter for particular tweets (a status update from a user) by particular users that I built to scratch my own itch, seen in figure 1.1. By default, you don't need to sign in or register to make use of the application. However, if as a user, I want to search my friends' tweets (those people I follow and listen to) I need to explicitly allow Snap Bird to access that information from Twitter. Rather than send the user straight off to Twitter's authentication page just because they selected a different type of search, we're going to show them a warning message, that tells the user what's about to happen, and why they need to jump off to the Twitter web site for a minute before continuing.

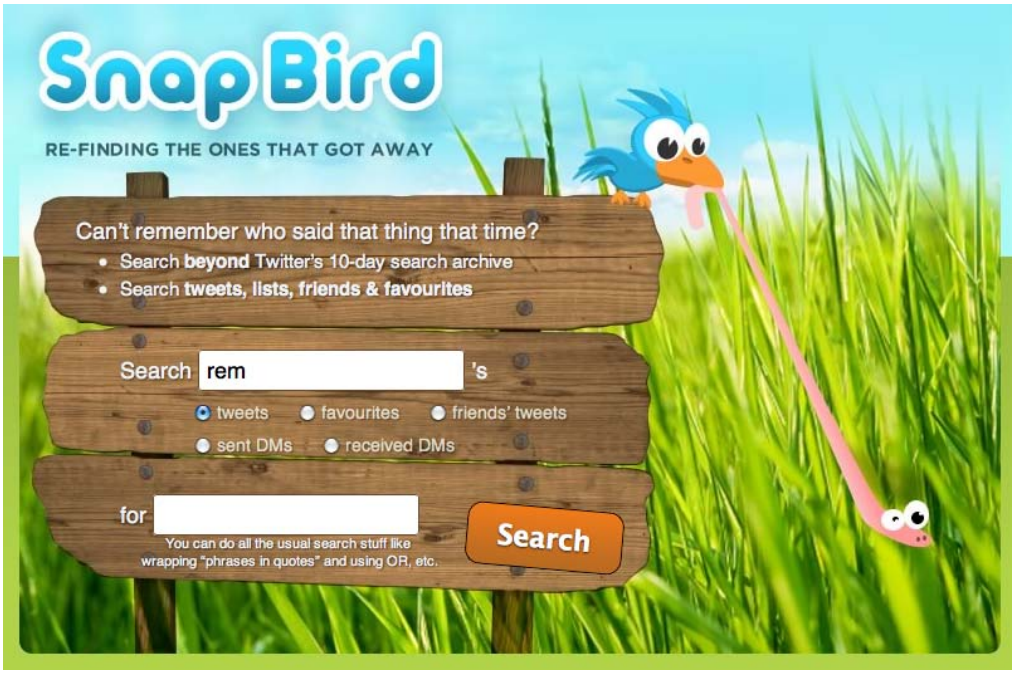


Figure 1.1 – Snap Bird starting page on http://snapbird.org

Our problem is that we want to show the overlay telling the user we have to send them to twitter.com before they can continue when they click on the "friends' tweets" radio button shown in figure 1.2, we want to show an overlay panel with some new information shown in figure 1.3.

All of this content is designed without jQuery and without the functionality applied. It's important that you don't try to style this using JavaScript, as CSS in style sheets is perfect for this job. Sometimes you may want to generate the actual elements using JavaScript, but in most cases you won't. You'll want to include the hidden markup in the overall web page, so that it can later be brought into visibility.

Figure 1.2 – The radio button that triggers the overlay in figure 1.3

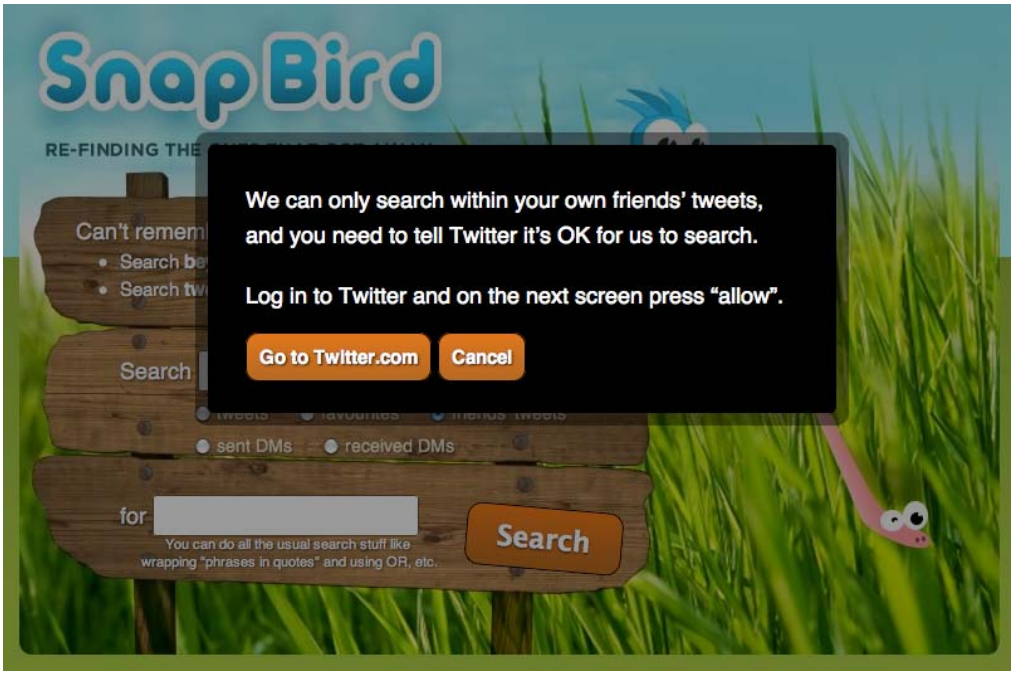


Figure 1.3 – User clicked on "friends' tweets" which requires a log in from Twitter.com

Showing content when we click

The solution in the case of Snap Bird is: *when the radio button is clicked, we want to show the overlay*.

The markup for figure 1.3 has been prepared so that the overlay is sitting in the HTML ready to be shown, a snippet of the markup can be seen in listing 1.1. Then using CSS, we'll hide the overlay initially, as seen in listing 1.2.

Listing 1.1 – Overlay markup for figure 1.3

```
<div id="auth">
  <div id="overlay"></div>
  <!-- nasty triple nesting, but necessary for the rounded border types
we're doing -->
  <div id="login">
    <div>
      <div>
        <p>We can only search within your own friends&rsquo; tweets, and
you need to tell Twitter it&rsquo;s OK for us to search.</p>
        <p>Log in to Twitter and on the next screen press
&ldquo;allow&rdquo;.</p>
        <p><a class="button"
href="http://twitter.com/oauth/authorize?oauth_token=">Go to
Twitter.com</a> <a class="button cancel" href="#">Cancel</a></p>
      </div>
    </div>
  </div>
</div>
```

Listing 1.2 – Hidden authorisation panel

```
#auth {
  display: none;
}
```

Now that our initial display is ready, i.e. when the user visits our page there's no visible overlay, we need to add the jQuery library to the page. Once the library has been included, we can do our magic with jQuery.

For the time being, we'll add jQuery and our JavaScript to the <head> element. There are a number of options for where we put the jQuery library, but we'll go in to those choices in chapter 3.

Listing 1.3 shows what our new <head> markup looks like with jQuery included. Our jQuery code is going to be written directly after the jQuery library is included. Note that listing 1.3 assumes that the jQuery library is in the same directory as *this* file (index.html or what you might have called the file). It's also worth noting that I've included the style sheets *before* the jQuery library. This is generally good practice for technical reasons beyond the scope of this book (but are explained here: http://api.jquery.com/ready/).

Listing 1.3 – Snap Bird with jQuery included

```
<head>
  <title>Snap Bird</title>
  <meta charset="utf-8" />
  <link rel="stylesheet" href="snapbird.css" type="text/css" />
  <script src="jquery.js"></script>
  <script>
    // Our code will go here
  </script>
</head>
```

Now comes the juicy step of including the code that will show the overlay when we click on the 'friends' radio button.

We've given the radio button the id of `withfriends` (not show in a listing as yet), but this is important because it's our anchor point. It's the element that's *clicked* that makes our overlay *show*.

Jumping in with the jQuery code

Listing 1.4 shows the <head> element with the completed jQuery code to create the interactive effect.

Listing 1.4 – Completed <head> with jQuery to show overlay

```
<head>
  <title>Snap Bird</title>
  <meta charset="utf-8" />
  <link rel="stylesheet" href="snapbird.css" type="text/css" />
  <script src="jquery.js"></script>
  <script>
    $(document).ready(function () {
      $('#withfriends').click(function () {
        $('#auth').show();
      });
    });
  </script>
</head>
```

http://jsbin.com/eyiyi/edit#html

Typesetter note: the link above will be styled as the interactive version of this code the reader can view and change and play with online.

For now, just try to ignore the $ symbol and all the curly braces and brackets. We'll come on to those in later chapters, and frankly if you've not seen this before, it's a little frightening.

So with the $ symbol set aside, we've really only got three lines of code here, and I'll explain what of them mean. To explain this, I'm going to work backwards through the code. We'll start with the final result, and we'll step back through the code to understand what's going on here.

SHOWING THE OVERLAY

The final result is that the overlay is shown. The overlay markup from listing 1.1 has a group of elements all wrapped in a single `<div>` with the id of `#auth`. This is the element we use jQuery to find on the page, and then tell jQuery to *show* that element:

```
$('#auth').show();
```

Notice that we're using the CSS id of the element to find the overlay. This is an important concept in jQuery that we'll constantly use throughout this book, and I'll explain in detail what your options are in chapter 3.

HANDLING THE USER CLICKING

The overlay is shown by the user clicking, so we need to capture when the user clicked on the radio button with the id of `#withfriends` as seen in figure 1.2. Again we're using CSS selectors to find the radio button that we want to capture this click event.

Once we've found the radio button, we give it a function without a name. For now you'll have to trust me that this works and it's an extremely common way of coding when working with jQuery. Again, I'll explain these odd looking functions in chapter 3:

```
$('#withfriends').click(function () {
  $('#auth').show();
});
```

This is not going to show the overlay until the element is clicked. When the `#withfriends` element is clicked, all the code inside the `function ()` code will run.

Finally we need our code to run when the document is ready for attaching interactive behaviours to. If we didn't only run when the document was *ready*, our code wouldn't work.

RUNNING WHEN THE DOCUMENT IS READY

jQuery provides a way to say "only run this code when the document is ready". If you plan to find an element on the page, you can only try to find it once the browser knows about it, i.e. the markup has been parsed into the DOM. What do that mean to you? If the document isn't ready, none of the code in listing 1.4 will work. It won't break, but it won't work. It's common to find your code isn't working but isn't causing any JavaScript errors because the code is trying to run before the document is ready.

To avoid this, we wrap all of the code in the jQuery `ready` method[2] as we see below:

```
$(document).ready(function () {
  $('#withfriends').click(function () {
    $('#auth').show();
  });
});
```

Now that the code is wrapped in the ready function our code runs at exactly the right time in the browser's list of jobs to do, and allows our jQuery code to correctly find the elements we want to interact with.

Simple change to create extra whizz bang

In only three lines of code you've got a dynamic effect. No doubt your mind is ready to explode with the possibilities of what you can do with jQuery, or you're keen to move on to see what recipes I've got in store for you. However, before you leave this chapter, I'll show you how you can add extra bling to this effect with the tiniest of changes.

---

[2] There are different ways to achieve this, and we'll look at them in chapter 3.

We'll change `.show()` to `.fadeIn()` and now you've got a cool little fading in effect. Not only do we now have a super smooth fade effect, but it also works in all browsers. Yep, that's right: it works in IE6 first time around. You can see the effect here: http://jsbin.com/eyiyi/2/edit#html

Summary

In this chapter you should have an understanding of the benefits of using jQuery on your web pages. You also have a list of resources that you ask questions if you ever get stuck or need help with a particular problem, or if you need more detail about a specific function.

Although little code has been used here, and if you knew nothing about jQuery to start with, you now have the skills required to create simple but effective interactivity on your web sites today. You know how to include jQuery, you know that jQuery revolves heavily around CSS to target parts of the document, and you know how to create simple effects.

What I'm showing you is that jQuery makes it very easy to add some cheap whizz bang. By cheap I don't mean 'cheap and nasty' like the bad old days of Geocities and animated GIFs. I mean that there's a cost to doing everything. If the cost of building some effect outweighs the benefit of having the effect, I'm less likely to do the work or the client is less likely to want the work. However, jQuery can switch this around for me, and allow my imagination to run wild. I can have an idea that typically would have taken me hours to code up without a JavaScript library, but with jQuery, it only takes me a few minutes, and now it's an easy decision.

The rest of part one of this book is dedicated to getting a handle on the background required to writing good jQuery code. In chapter 2 I'll talk about best practices and specifically progressive enhancement.

2

jQuery & Progressive Enhancement: A Perfect Match

Now that you've cut our teeth on jQuery and created our first effect with a tiny dash of whizz-bang, I want to take a step back and look at the bigger picture and take you through some of the principles in designing our jQuery code and look at some of the issues we could encounter with each approach.

This chapter is here because it's really easy just to copy and paste the solutions from the web, or even the book and not understand what's good code and what's bad code. jQuery for Designers hopes to help you, and give you solutions that you can paste straight in to your web sites, but if we're going to build a better web, and Do Things Right, then it's important that you understand the right way of doing things.

In this chapter I'm going to discuss unobtrusive design in JavaScript and explain what is and what isn't good practices. Often I find it's easier to understand why something is good if you've seen the wrong way of doing things (i.e. writing code) – so I'm going to walk you through what the bad practices look like first, then show you what the right practice code looks like.

Approaches to Unobtrusive JavaScript

Unobtrusive design goes beyond HTML and CSS. It goes on to include JavaScript and how JavaScript is applied to the page. jQuery makes this process of unobtrusive design much easier, but we still need to understand the best practices and how to approach them.

Unobtrusive JavaScript is a term used to describe how JavaScript has been written on a page that still works if JavaScript isn't available. For those of you that like tongue-twisters and buzz words: you're in luck, because unobtrusive JavaScript splits in to two approaches that both aim to achieve the same thing, and in my experience, web developers, the folk that are supposed to know what they're doing, get these two approaches mixed up far too

often. This chapter will discuss these two approaches and explain to go about writing your code so that you're aware of how to achieve unobtrusive JavaScript.

Before I show you how to *write* unobtrusive JavaScript, I need to introduce you to the two ways in which we can achieve unobtrusive JavaScript, this is because even though graceful degradation and progressive enhancement may be different in their approach, they *both* aim for the same outcome.

Graceful Degradation

The first of these approaches is called *Graceful Degradation*. This is where the web page is designed to have all kinds of bells and whistles, and handles exception cases when JavaScript isn't available. There are two common examples of this that I often come across. First of all there's the "here's a completely different web site if you have JavaScript turned off" example. This is actually quite a valiant approach to graceful degradation since the amount of work is often increased greatly to maintain two sites, but if it means the user can still have a full experience it's worth the work. Google's Gmail is a perfect example of this. If I visit Gmail with JavaScript enabled, as seen in figure 2.1, the experience is a rich and interactive one. If I visit Gmail with JavaScript *disabled*, as seen in figure 2.2, I can still navigate my email, read and send emails. Perfect. Everything still works for all users.

The second example of this is if you visit a web site with JavaScript turned off, and you might see "In order to experience this web site, you'll need to enable JavaScript". Although common, this in my humble opinion, isn't particularly *graceful*, since the entire experience is broken without JavaScript. Figure 2.3 shows visiting http://me.com with JavaScript disabled.

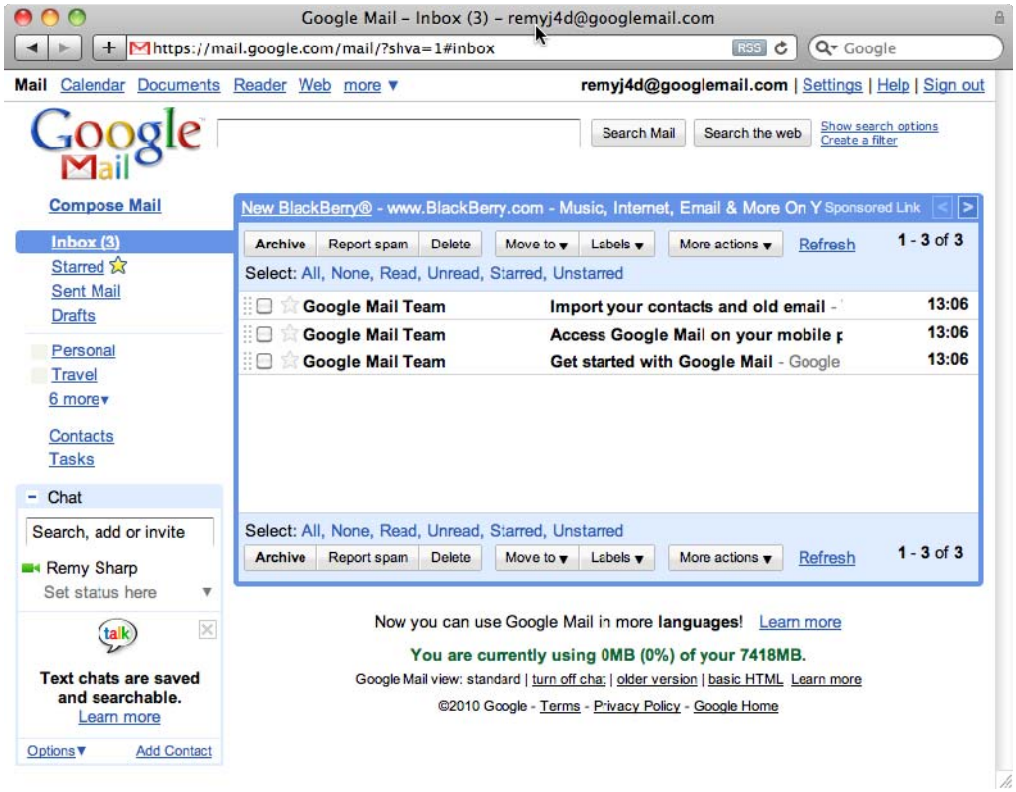


Figure 2.1 – Gmail with JavaScript enabled, notice even the browser title bar is different to figure 2.2.

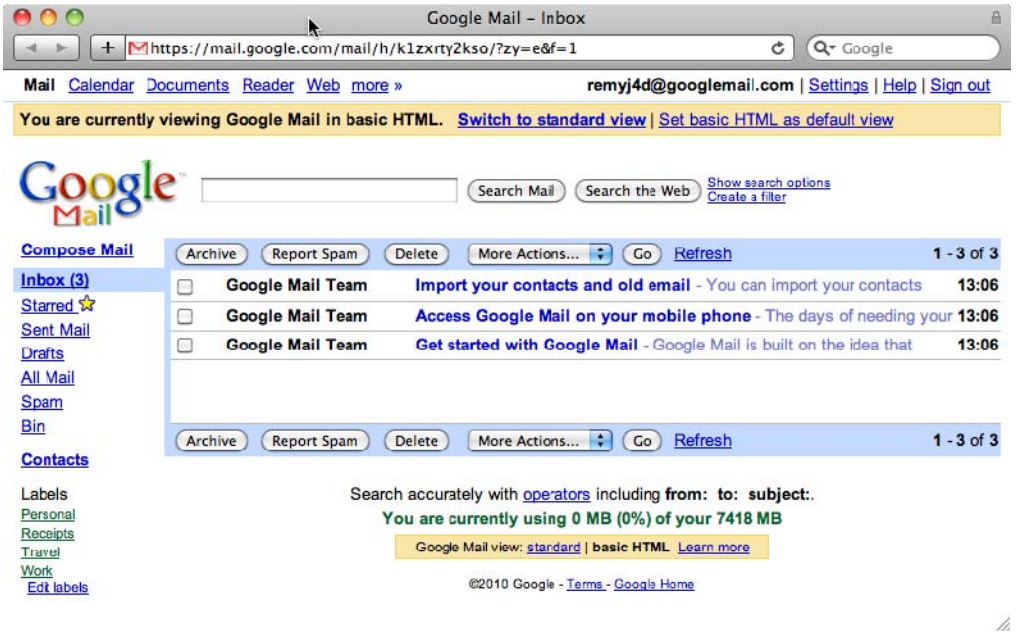


Figure 2.2 – Gmail with JavaScript disabled, a separately maintained web site

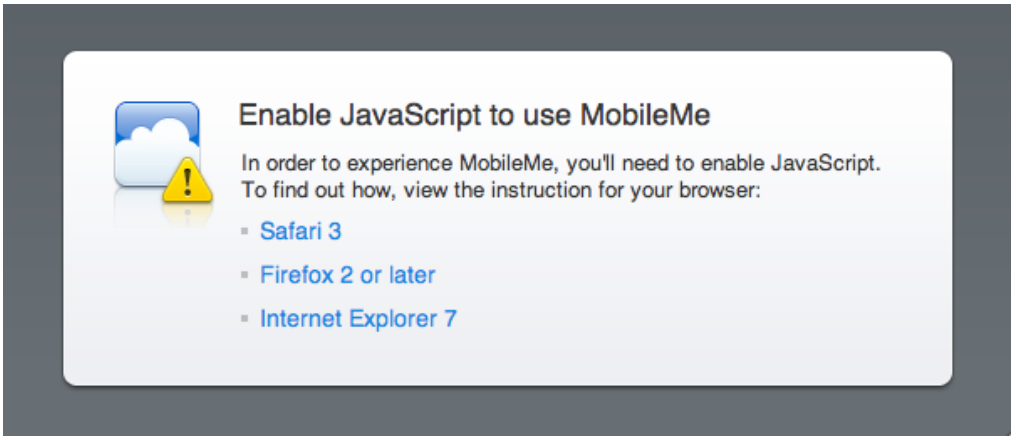


Figure 2.3 – MobileMe with JavaScript disabled, a bad example of “gracefully” degrading

Progressive Enhancement

The second approach is *Progressive Enhancement*. This approach, as it's ingeniously named, progressively enhances the page when it finds that the user has JavaScript turned on. There

are lots of good examples of this throughout the web ranging from galleries to menu navigation. If we use Delicious, Yahoo's bookmarking service for example. If I visit the site with JavaScript disabled and click on the *bookmarks* menu item, the page reloads to take me to my bookmarks and also offers me links to submit more bookmarks, view the popular bookmarks and see recent bookmarks.

If I visit Delicious with JavaScript enabled, the menu is progressively enhanced and the JavaScript has added a down arrow in each of the menu items, which when I click on the down arrow in the *bookmarks* menu item it exposes a submenu with the links I described above. If I click on the actual bookmark text in the menu, the functionality matches the experience as if I didn't have JavaScript. We can see the differences in the menu in figure 2.4 the menu without JavaScript and figure 2.5 shows the menu *with* JavaScript.

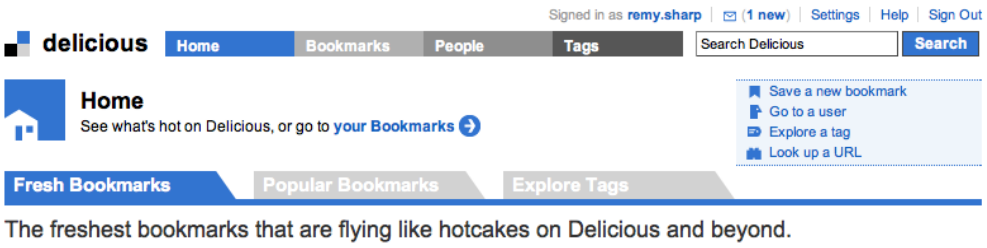


Figure 2.4 – The Delicious home page with JavaScript disabled, everything works as expected

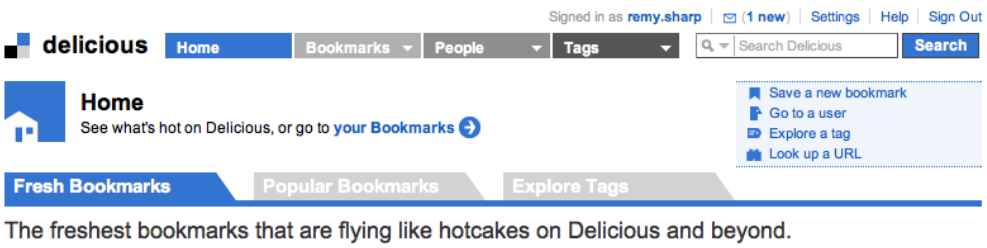


Figure 2.5 – The same Delicious home page as figure 2.4, but notice that JavaScript enhances the menu navigation by adding a drop down arrow

Next we'll look at what actually constitutes unobtrusive JavaScript by first looking at examples of what kind of code fails to be unobtrusive.

Unobtrusive JavaScript

Unobtrusive JavaScript is the starting point to a well designed web site. This means the web site can still work to its full capacity without the need for JavaScript. As the name suggests, unobtrusive JavaScript is code that has been written in a way that doesn't interfere with the

page itself. The idea is that the page will work, rather than break, if JavaScript isn't available.

Taking the unobtrusive approach to building your site means that your site will physically accessible by more devices and platforms. The most common uninformed response to the idea of unobtrusive JavaScript is:

"Surely doesn't everyone have JavaScript turned on?"

This is the point you can leap on your soapbox and put them right. No, not everyone has JavaScript enabled, but it doesn't always boil down to enabling or disabling JavaScript. It may be that the platform is slower and won't execute the JavaScript by the time the user has interacted (i.e. the page may appear unresponsive). It may be that the user is visiting from a mobile device that modifies the JavaScript; the Opera Mini (mobile device) browser removes certain types of JavaScript (for security and performance reasons) and disables timers after the first second the page is loaded – which means animations won't work on this device. With these considerations and reported stats that between 5-10% of users with JavaScript disabled means when I build a web site I'm thinking about how to make the JavaScript unobtrusive.

Don't assume JavaScript is turned on

You may initially think that JavaScript is always turned on because your browser at home has it on by default, but this isn't the case. Large corporations may have company wide policies to block JavaScript. The NoScript Firefox plugin is very popular which disables JavaScript site by site, and the user can manual decide which sites should run with JavaScript.

NoScript

NoScript is a plugin for Firefox that blocks JavaScript from running by default. The user can optionally turn on each web site's permission to running JavaScript.

There are also alternative devices: you won't be surprised to learn that the web isn't just visited on a desktop browser! There are assistive technologies, such as screen readers that

have varying degrees of support for JavaScript. Unfortunately not all mobile phones are the iPhones with Safari built in - some mobile phone browsers have limited or no JavaScript support.

There's also our friend Google that will help visitors find our content. As Google indexes our pages, it doesn't run with JavaScript - so any functionality or content depending on JavaScript will be lost on Google.

It's always important to think about how your site will be accessed, and by what kind of demographic as this will help you decide how to approach how you design your solutions.

Most of us have probably been guilty of writing JavaScript that isn't unobtrusive at one time or another, but it's worth keeping in mind what this looks like so we know what to avoid.

What's bad JavaScript look like?

Although there's no real term for the opposite of unobtrusive JavaScript, I want to walk you through a number of examples of how simple JavaScript can mess up the experience if JavaScript isn't available. The examples I'll show you aren't arcane examples of code, they're snippets that we still see in the wild. If you're new to writing JavaScript, you'd be forgiven for thinking these examples are okay, but only forgiven once! It takes very little extra thought to ensure that the solution you're working on actually works unobtrusively.

For our example, we are going to create a new popup window to the help section of our web site we want use JavaScript to provide that functionality to create a clean new window which we could control the height and width of.

There are a number of different ways to achieve this. Some bad ways, and a few good ways. Let's start with the worse way first.

JAVASCRIPT PSEUDO PROTOCOL

It's *possible* to write the link as:

```
<a href="javascript:window.open('help.html')">Help</a>
```

It's *possible*, but we shouldn't. This is called the JavaScript pseudo protocol. It's called a pseudo protocol because it's being referenced like a URL, but it's not a *real* protocol like http:// is, it just happens that most browsers support it if they find it.

The simple problem here is if the browser *doesn't* support the pseudo protocol then it will navigate to a non-existent URL. Your application has now broken.

The best example of this being a problem is when Google's search robots come in and index your site. As they don't currently follow the JavaScript pseudo protocol this page won't be never be found and therefore never indexed. This means less Google juice for your site, less SEO goodness and could mean less new visitors for your web site.

EMPTY LINKS AND ONCLICK

It's *possible* to write the link as:

```
<a href="#" onclick="javascript:window.open('help.html')">Help</a>
```

Again, this is *possible*, but we shouldn't do it. There's no semantic meaning to this link, and of course if the visitor has JavaScript disabled (remember Google again), then the link does nothing.

WORKING LINKS AND ONCLICK

It's *possible* to write the link as:

```
<a href="help.html" onclick="window.open('help.html'); return
false;">Help</a>
```

This is much better than the previous two methods. If JavaScript is disabled, the visitor can still get to the help page. If JavaScript is enabled, then they get a popup.

We add the return false to the `onclick` so that the JavaScript enabled visitor doesn't have both the popup and the browser navigate to the help page.

Even though this is a much better approach, this is intruding on our markup. Just as you wouldn't style all of your document using inline styles, you don't want to add your JavaScript inline.

As unobtrusive JavaScript goes, what's good for CSS is good for JavaScript, so let's look at how to use unobtrusive JavaScript for the link.

Writing unobtrusive code

Continuing with the same example as before of creating a popup help window, we'll look at how the markup should be written, and I'll explain how and why it's better your visitor.

UNOBTRUSIVE WORKING LINK

We *should* write our link as:

```
<a href="help.html" class="help">Help</a>
```

The first thing you notice is that there's no JavaScript. That's because we are loading the JavaScript in a separate file somewhere else in the markup. The code, when it runs, would search for the anchor element with the class of 'help' and tell the browser that when it's clicked, it should show a popup that the link would have originally linked to. This is the right solution for a number of reasons, including:

1. If JavaScript isn't present, the link still works. For examples, this means that Google can spider the link and site
2. We can use DOM scripting to attach our click handling event, which means, if written defensively, we could change the markup and the JavaScript will still work
3. Our JavaScript is separated from our markup, which means we can modify it without having to modify the markup which may well be handled by another developer in the team

Making your pages Progressively Enhance

As we've seen in the examples previously, progressive enhancement is the practice of first providing the solution to work in all the possible environments and platforms, such as the different browsers (IE, Safari, Opera, and so on), mobile phones, printers, Google and so on.

This version of the site with JavaScript turned off will work by default. Then by taking advantage of the environment's features we can progressively enhance the experience for the user.

Although the situation is getting much better, the current state of many web sites is to use graceful degradation, and in a lot of cases the sites outright fail with JavaScript turned off. Another example of JavaScript fail was a version of Twitter during early 2009. You couldn't submit your status update if JavaScript was turned off – this meant that the core feature of the web site didn't work if JavaScript was disabled. I'd suggest this was fairly serious. The reason why Twitter's update button didn't work without JavaScript, was because it relied on Ajax to submit the status. If they had used progressive enhancement, Twitter would have created a "normal" form, with a submit button, then enhance using JavaScript, making it an Ajaxified form so that the page didn't have to reload. Twitter has since updated their site and included in the updates was the ability to use the site successfully without requiring JavaScript.

Next I want to walk you through two important concepts you should keep in mind when taking a progressive enhancement approach, and then I'll show you how you would use jQuery to progressively enhance – because the code is a lot shorter and a lot easier to whip together.

The layers of progressive enhancement

Progressive enhancement builds on creating a solid foundation, or core, to work from. By starting at the core of the problem and building outwards and upwards we are ensuring that core product, without and fancy styles or fancy interaction, works fully. From there we can layer on the CSS and JavaScript. The layers of progressive enhancement can be visualise in a number of different ways – I've seen a visualisation using cakes that is a surefire way to make the idea stick with most developers! In figure 2.6 I've likened progressive enhancement to Maslow's hierarchy of needs. I like the idea that to be truly enlightened web developers and designers, we've hit all the basic needs of as many different platforms and rising up to excellence using a careful massaging of JavaScript! Regardless of how it might be visualised, progressive enhancement breaks in to the following three layers:

1. Content: Deliver the basic markup to allow for a complete experience without the prerequisite of JavaScript
2. Design: Externally link CSS so that styles aren't embedded in the markup, and browser that don't support particular CSS functionality can be targeted appropriately
3. Behaviour: Enhance the experience using externally linked JavaScript

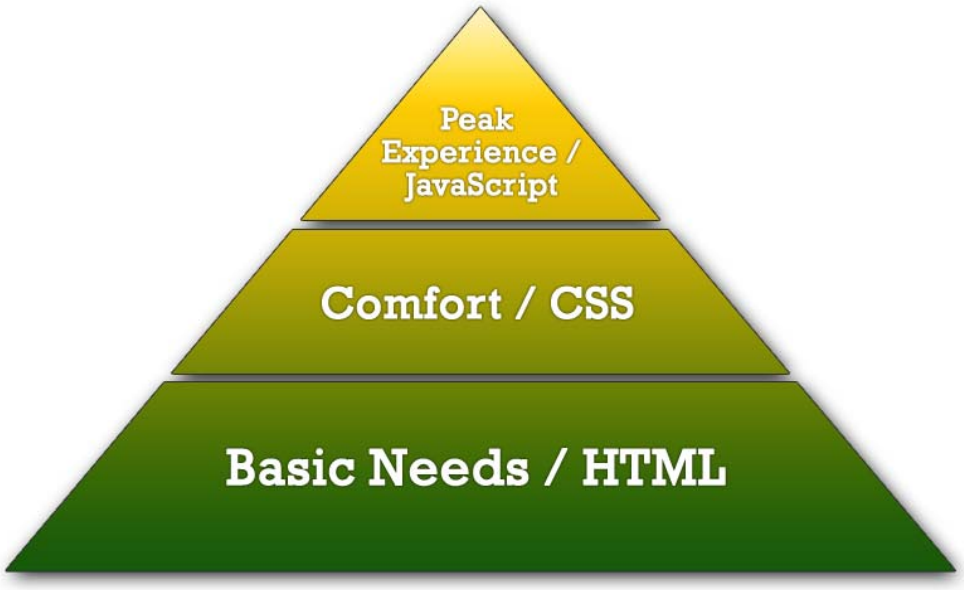


Figure 2.6 – Imagining the progressive enhancement stack as the Maslow's hierarchy of needs, but for web development.

Can production please create some sexier version of the hierarchy of needs

I won't be addressing the first two layers of progressive enhancement as this books focus is jQuery and therefore for progressive enhancement: the JavaScript layer.

Working with Browser Capabilities, and don't Browser Sniffing

You may already be familiar with the technique of "browser sniffing" in CSS, either using CSS hacks (generally bad) or conditional includes (generally better), but in JavaScript browser sniffing can be JavaScript's kryptonite. If you sniff for a browser to deliver specific functionality, and you get the browser wrong, you've turned away your visitor for no good reason.

In JavaScript, browser sniffing is achieved by looking at the User Agent and based on this and what we think the browser supports we will perform some functionality. This is where it can go all horribly wrong.

Identifying User Agent

The user agent is a unique string that identifies the particular browser, by the particular vendor, it's version, the render engine and the platform it's being viewed on. To see the user agent, try popping the following JavaScript in a page and you'll be presented with the user agent string:

```
<script>alert(navigator.userAgent);</script>
```

A cautionary tale: The browser Opera was the first browser to get to double digits, i.e. the first browser to get to version 10. During some of the early development testing they found that some web sites web checking the user agent and looking for the version number (this is a fairly typical technique in browser sniffing). However, the web sites were saying (something like): "if this browser is less than version 4, then tell the visitor their browser isn't capable". The problem was the way the web site was doing the version comparison, the result was that "version 10" was appearing to be *less than* "version four". This is clearly wrong to a human, but turned out to be perfectly correct for the computer. This is why Opera, version 10 and above, now announces itself as version 9.80, so it can work around the badly written browser sniffing code that exists on the web.

This is obviously an extreme example of browser sniffing gone wrong, but an important lesson none the less. If the web site had actually tried to *detect* the functionality they wanted to make use of they would have been fine. Think of it like this if you will: don't judge a book by its cover: a browser may have qualities you didn't think they had!

Working with what you've got: feature detection

Feature detection is the way of saying we won't be browser sniffing, we'll check first to see if can do some magic task, then get the browser doing that magic task. For example, we know that IE6 doesn't support CSS2.1 opacity (it uses Microsoft custom filters), but what about other browsers? The right way to solve this problem is check whether the browser can support opacity, and if not, use the work around.

Old versions of the jQuery library used browser sniffing to fix problems in different browsers. From version 1.3.2 onwards (release early 2009) dropped all of its browser sniffing code in favour of feature detection. Now if you ask jQuery to create an opacity animation, it will have detected if native opacity works, and if not, handle the edge case appropriately behind the scenes for you.

jQuery also exposes all of this information for you too, so that you can check if some feature is supported and act as appropriate.

FEATURE DETECTING EVERYTHING! MODERNIZR

jQuery only detects what it needs to work properly in all the browsers it supports. There are a feature other features it detects, such as box model support, but it doesn't aim to detect every feature under the sun. Modernizr http://modernizr.com, on the hand, does aim to detect every feature under the sun. Written by Faruk Ates and supported by Paul

Irish (a jQuery team member), Modernizr will add class names to your document indicating whether a CSS or HTML feature is supported, such as RGBa or CSS transitions.

Unobtrusive JavaScript without jQuery: The Issues

Using the example from chapter 1, where the user could click on the radio button to show the log in overlay, listing 2.1 shows you what progressively enhancing the page would look like if we used *vanilla DOM scripting*.

This code solves our problem and assuming the markup on the page supports progressive enhancement, it does the job as you'd expect. Except there's some problems that don't immediately show themselves, and jQuery can be solve without really having to pay these problems much thought.

Listing 2.1 DOM Scripting without jQuery

```
window.onload = function () {
  var inputs = document.getElementsByTagName('input');
  for (var i = 0; i < inputs.length; i++) {
    if (inputs[i].type == 'radio' && inputs[i].name == 'withfriends') {
      inputs[i].onclick = function () {
        document.getElementById('auth').style.display = 'block';
      };
    }
  }
};
```

If you've dabbled with JavaScript then you might have an idea of what this code is doing, if you're not happy dabbling with pure JavaScript then you may not be interested at all. I'll show you next how this code is easier to write using jQuery and actually better. Before I do, I want to briefly outline what's going on in listing 2.1:

1. When the window has finished loading (all the DOM, all the CSS and all the images), it runs all the code
2. Find all the input elements
3. Loop through those input elements looking for the input element that is a radio button and also has the name of "withfriends"
4. Attach an action when the "withfriends" radio button is clicked
5. When clicked, find the element with the id of "auth" and set the display style to block (which if it's currently set to `display: none` it will appear for the user)

Really the code boils down to: when the document has finished loading, find some elements and do *stuff* to them.

Aside from being very JavaScripty, there's a couple of problems with this code that jQuery can solve for us without any work from us. The first problem is that this code only allows for one single function to run when the window is loaded. Adding another function like this would replace the existing onload function.

The bigger problem is the time in which the code runs. Currently it's when the window has fully loaded. This is when *everything* has loaded: the DOM, the styles, any other JavaScript and the images.

Out of that list we, as the author, need the DOM and the styles in place before our code runs, because without the DOM, we wouldn't be able to attach the click functions, etc. JavaScript running first is something we can't do anything about, JavaScript runs sequentially and when one script is running, no other JavaScript can run (in fact the whole browser waits until the script has finished running before it carries on, drawing or building the DOM, etc). The images however, we don't need. In fact if there's a lot of images, this could take up precious time. What if the user clicked before our code has run because the images took too long to load?

jQuery runs the JavaScript at the earliest possible time, and that's *before* the images have loaded. We *could* code this ourselves without jQuery, but the code gets a lot more complicated, especially when we start handling how different browsers get to the *ready* state.

In addition, the same jQuery method allows us to set lots of onload functions if we want, I'll show you this in chapter 3. Let's see how this code is easily achieved using jQuery.

Unobtrusive JavaScript Made Easy with jQuery

jQuery does a great job of hiding away a lot of the problems discussed above. We can easily rewrite the DOM scripting code in listing 2.1 in just a few lines of code, as shown in listing 2.2.

Due to the way we write our jQuery code to solve a problem, we also typically inadvertently write code that progressively enhances the page. It's certainly possible to write bad jQuery code, but you'd have to be ignoring some pretty simple rules of thumb. So long as you keep your JavaScript *out of* the markup, and your site works without JavaScript first, then you're making good progress towards progressively enhanced goodness.

Listing 2.2 DOM Script with jQuery

```
$(document).ready(function () {
  $('input[type=radio][name=withfriends]').click(function () {
    $('#auth').show();
  });
});
```

Our requirements are to:

1. Run the code as early as possible, but when the DOM is ready
2. Find the radio button whose name is "withfriends"
3. …and show the element whose id is "auth" when the radio button is clicked

Now looking at our code we can see three lines, two of which have CSS selectors. Let's break down what's going on here:

First we tell jQuery to run all of the following code once the DOM is ready, but before images have loaded:

```
$(document).ready
```

Next we use a valid CSS2.1 selector and pass it to jQuery to find the radio button we want to attach the click action to:

```
$('input[type=radio][name=withfriends]')
```

Once jQuery has found our radio button, we tell it to run the following code if it's clicked (by adding the `.click` after we find the radio button), which again using a CSS selector, searches for the element whose id is "auth" and once wrapped in jQuery we can use the show method to make it appear:

```
$('#auth').show();
```

That's it. Three lines of functional code. In fact the most complicated looking part of that code is the CSS selector to find the radio button, but since you're already using CSS day to day, it shouldn't be a big deal. You could change the code to us an id on the radio button (which is what we used in the first chapter example), but both are perfectly acceptable.

$(document).ready() tip

The $(document).ready() functions fire in the order they are found by the browser, for instance first come, first serve.

Summary

A jQuery motto amongst its developers is: *find something, do something to it.* Using this approach to designing code mirrors the way progressive enhancement works, with no assumptions about the environment that you're working within. The default way of working with jQuery is to run all the code once the document has finished loading, then using CSS selectors to target the elements and add some behaviour to them: find something, do something to it.

We've looked at how to achieve unobtrusive JavaScript, and how progressive enhancement is right route to take over graceful degradation, and how jQuery makes this very easy to do. If you build your site using progressive enhancement, it will be available on more devices and work fine within different browser versions, and continue to work even if some functionality isn't available within the browser.

jQuery allows you achieve all this without you having to jump through any hoops or having to learn anything particular to the progressive enhancement approach. Before we move on to Part 2 and the individual recipes using jQuery, Chapter 3 will take you through some of the Tools of the Trade, covering some of the basics of jQuery and some high level topics in CSS, JavaScript and the available debugging tools.

3

Tools of the Trade

So far I've been fairly gentle on you, and I hope to keep it softly softly for the rest of the book, but for this chapter, I'm afraid, I'm going to have to go techie on you. As much as I'd love for developers and designers click their heels and the code magically appears in a perfect state, it's just not possible without getting your hands a *little* dirty with code. This chapter is here to help prime you with some of the techie bits you'll need to know, so you don't run away screaming when you see an army of curly brackets or some such JavaScript nonsense. Rest assured though, I'm not going to go in to huge detail on each of these topics, there's many a book dedicated to each subject separately, but we need to have an understanding of the tools (or code) that we'll be using if this is going to gel in your head.

Along with basic jQuery notation, the infamous $ (dollar) function and style of coding with jQuery, we'll look at JavaScript and some of the gotchas in the language and perhaps you'll also be able to fake your way in being a JavaScripter.

I don't plan to explain JavaScript in huge detail here, and I certainly don't plan to explain what jQuery is doing under the hood. What you will learn is what jQuery looks like, what the dollar function does and what a "chain" is. What I will show you, for instance, is how to add jQuery to any page, and start messing around with it right away.

In a later section, we'll also look at the de-facto debugging tool called Firebug for Mozilla Firefox, and how it can be a very powerful tool for understanding where a problem might exist in the code.

I'll also introduce you to the jQuery UI library, a complementary library to jQuery that provides a number of predefined widgets, such as datepickers and accordions to name a couple. Out of the box these widgets can be used in production sites, can be themed (i.e. the style of the widget) and extended (though extension of a widget is beyond the scope of this book).

Finally I'll touch on the JavaScript syntax, explain some of the subtleties that you may not know about and show you JSON (JavaScript Object Notation), a powerful, but incredibly

simple and popular syntax for transferring data from the server side to the client side (the browser). For example, JSON is used in APIs provided by Flickr, Twitter and Google.

Detailed Look at the $ and jQuery function

In all the examples you would have seen in this book so far, we've used jQuery using the $ function – although on first appearances it may not be completely obvious it *is* in fact a function. When I include the jQuery library in my page, all the <script> tags below that point can now make use of the jQuery $ function. Along with the $ function, we also have the *jQuery* function. They do the same thing. They **are** the same thing. The $ function is actually just a shortcut to the jQuery function.

So let's say I'm feeling reckless and I want make all the links on my Twitter stream page *red*. If it's important to me, I could do it, and I could do it using jQuery really easily using either the $ function or the jQuery function as you can see here:

```
$('a')
/* or */
jQuery('a')
```

This tiny piece of code is grabbing all of the links on the page. Later in this chapter, I'm going to show you Firebug, a debugging tool that I can use to run the code above on a live web page and actually manipulate the page live there and then.

Again, so that we're completely clear: I can use either the $ function or the jQuery function and they perform the **exact** same task. The jQuery function and the $ function are the exact same thing. So you can pick and choose which one you want to use. For the remainder of this book, I'll use the $ function. You can pick either one, but I'd recommend to sticking to one and not switching back and forth through your code – that's just confusing!

Going back to our aim to splash my Twitter stream with red paint, a very simple line of jQuery code can be seen in listing 3.1, which will find all the links on our page and change the background colour to red, dark red. Starting with a simple page, such as the Twitter home page (or rather *my* Twitter stream), everything appears normal, as seen in figure 3.1.

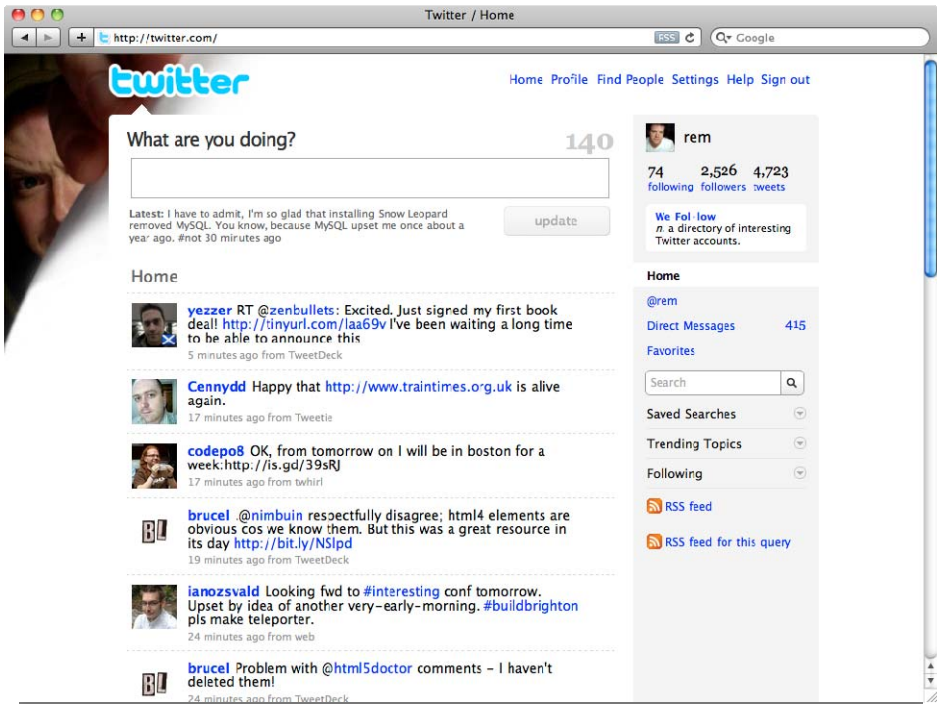


Figure 3.1 – A Twitter stream before our jQuery is applied

Then we'll add the jQuery to make the links have a red background, by first including the jQuery library, and then in a subsequent script element to run our jQuery code. In listing 3.1 I've included a link that you view in your browser where I've captured the source code for my Twitter stream at that time, and added the jQuery code from listing 3.1 so you can see, first hand what the page feels like after I've splashed it with my jQuery link painting script!

Listing 3.1 - simple use of jQuery starting with a dollar symbol

```
<script src="jquery.js"></script>
<script>
$(document).ready(function () {
  $('a').css('background', '#c00');
});
</script>
```

http://jsbin.com/onajo3/3

Figure 3.2 – the result of running listing 3.1

The result is seen in figure 3.2. Possibly the most important part of this code is to recognise the $ symbol at the start of the code. Although this is just one of the ways to use jQuery, a jQuery statement will always start with a dollar (or the jQuery alias), and we'll see why next.

Everything starts with a dollar

As I mentioned before, all jQuery statements start with the dollar function.

The dollar symbol is a function. It's a function just like `alert` is a function. That's why the dollar symbol always has a left bracket, something going in to the function, and a right bracket, because it's a function.

Using jQuery with other libraries

The main reason why there are two ways to access jQuery, using the `jQuery` function and the $ function, is for compatibility. If you're using the Prototype JavaScript library, you will find that *both* jQuery and Prototype use a dollar function. So what happens when

they're both trying to use the dollar function? One loses out. However, jQuery has a trick up its sleeve. If you include jQuery after Prototype when loading in the libraries, you can give the dollar function back to Prototype, and then you can use the `jQuery` function for all your jQuery needs (and there's ways to use the dollar function with jQuery too which we'll look at later).

If you're using another library that uses the dollar function, you need include jQuery last, remember "good guys come last", and use this code:

```
jQuery.noConflict();
```

This gives the dollar function back to the other library, and now you can use jQuery. If you still want to use the dollar function with jQuery, you can use the following wrapping trick:

```
jQuery.noConflict();
jQuery(document).ready(function ($) {
  $('a').addClass('highlight');
});
```

Note that the above code should be part of the sidebar style and the next paragraph continues as part of the box.

Notice how we're passing in the dollar variable in to the function, this is actually a copy of the jQuery function. Now we can safely use jQuery via the dollar function and Prototype's dollar is still intact.

You've seen from previous chapters that we can pass CSS selectors to jQuery, but you can do more than just CSS. There are four fundamental ways of using jQuery aka the dollar function.

1. QUERY THE DOM FOR A SPECIFIC SELECTOR

This is probably the most typical use of jQuery. We are passing a CSS selector (or a custom jQuery selector which we'll see in later chapters) to the dollar function to search the DOM for so that we can manipulate the result. For example, to find all the a elements that are direct decedents of a div element with the class "foo", we use the following code:

```
$('div.foo > a')
```

As you see this is normal CSS syntax to target a specific selection of elements, just as you would using a CSS style sheet. To test whether your selector actually found anything, jQuery provides a length property that you can test for. So if there's five links that are direct decedents of the `div.foo` element, the length property is going to show five:

```
alert( $('div.foo > a').length );
```

It's really important not to be caught out when trying to test a selector. jQuery silently fails when a selector isn't found, i.e. it doesn't break your code. This is a good thing and I'll come on to why, but because of this feature, the following code is wrong:

```
// this code is WRONG!
if ( $('div.foo > a') ) {
```

```
  // then do something
}
```

It's wrong because the result of `$('div.foo a')` does actually have a result. Although there are no elements found, there is a result. So the correct way to write the code above is to test the length property:

```
// this code is RIGHT way to test a selector
if ( $('div.foo > a').length ) {
  // then do something
}
```

Later on in this section we'll look at chaining, and you'll see how you can always use the length property through each phase of the chain if you need to.

TIP: JQUERY SILENTLY FAILS

jQuery silently fails when running a selector that doesn't match anything. You may think this is a bug or perhaps something should happen when it doesn't find anything. This isn't a bug. Think of it in respect to CSS. If you had some CSS style that didn't match any thing on the page, would you wouldn't expect the page to explode or cause an error. jQuery has the same approach. Though there's a completely different technical reason, if you keep thinking jQuery is using CSS selectors, and if you ensure your CSS selector matches, you'll be golden.

2. DOCUMENT READY

Probably the second most common use for the dollar function: the ready function. As you would have seen in Chapter 1, you would typically wrap all your jQuery code with the following statement:

```
$(document).ready(function () {
  // my code goes here
});
```

In this example instead of passing the dollar function a CSS selector, we're giving it the `document` object. Once jQuery has the document, we can use the ready method and give it a function that tells jQuery: run my code when the browser has finished building the page. Once the document is loaded, jQuery fires the "ready" function, which happens after the markup is loaded, after external scripts have loaded, but *before* images are loaded. This way your code is running at the earliest possible point in time whereby you'll have full access to the entire DOM. Otherwise, if your code ran before the DOM was ready, your selectors would fail to find the elements in question. For example, if you ran your code in the head element without a ready function, it wouldn't find any of the document elements since none of the body elements would have been loaded by the browser yet.

There is another common way to represent the `$(document).ready()` method:

```
$(function () {
  // my code goes here
});
```

This has exactly the same meaning as before, but it's just a shorthand way of setting "ready" functions. It's useful to recognise this method of setting the ready function, as you may find it in a lot of online examples and tutorials.

3. JQUERYIFY EXISTING ELEMENTS

By jQueryify, I mean that we're taking an existing element, perhaps a DOM element we created earlier using JavaScript that we haven't inserted in to the document, and now we want to use jQuery to manipulate that element. For example, if I wanted to add the class "foo" to an existing element, I could use the jQuery `addClass` method to do it, but first I'd need to wrap that element in jQuery, as you'll see in the following code:

```
var elm = document.createElement('a');
elm.href = "http://google.com";
$(elm).addClass('foo');
```

By wrapping elm with jQuery, via the dollar function, I'm now able to use all the jQuery methods against that element.

4. CREATE NEW SNIPPETS OF HTML AND JQUERYIFY

Just as we did before, by wrapping jQuery around a snippet of HTML, we are given access to all of jQuery's methods. For example, if I wanted to create a new div element with a class of "foo" and use jQuery to append to the body element, we would pass he raw HTML to jQuery, and then use the appendTo method as seen in the following code:

```
$('<div class="foo"></div>').appendTo('body');
```

This can also be represented as XHTML with a self closing tag for empty elements:

```
$('<div class="foo" />').appendTo('body');
```

jQuery is creating all elements in your HTML on the fly, and then wrapping it with all it's functionality. This is useful for instance, when you need to create some element on the page that can only be accessed using JavaScript.

What is Chaining

jQuery syntax, although not unique, does have a particular visual style to it. For example, the following code isn't unlikely:

```
$('img').addClass('foo').click(fn).attr('src', 'someimage.jpg');
```

The "chaining" aspect is where the methods are chained one after another. So the first method is `addClass`, then click and so on. They're joined together using dots and they act as one long command. Equally, you can see a chained call on multiple lines because the JavaScript syntax allows you to run a command over multiple lines in particular situations. For example the above code can be written on multiple lines so it reads:

```
$('img')
  .addClass('foo')
  .click(fn)
  .attr('src', 'someimage.jpg');
```

Using multiple lines may make the code easier to read, rather than having one very, very long single line. Equally I would argue that one very, very long chain of commands can be confusing to other people trying to use your code, so I'd recommending using chaining sensibly.

The way the chaining works is that the result of the previous function is returned which allows us to call another jQuery method on the result. This only works where the result of the method is a jQuery object. For example, running the find method searches the current context of the DOM for the selector you pass the method. The result is those found elements wrapped in jQuery. As such, we can then call another jQuery method, such as addClass. If the method returns a string, such as if we were getting the href attribute from a link, we can't chain this using jQuery, because the result was a string.

Most jQuery methods return a jQuery result: either the current selection of elements or the refined selection (from find or filter for instance), which means the result will chain. If the result doesn't chain, the browser will complain that the "Object doesn't support the specified method" (or to similar effect).

Chaining is a common stylistic aspect to jQuery which you'll see throughout this book and many examples throughout the web. Understanding that the latest results from the selection is passed along the chain is the key to being able to read the chain of code.

Testing jQuery in Any Page

I strongly encourage you to install and try out the jQuery bookmarklet by Karl Swedberg as seen in figure 3.3 after it's been run. This is a bookmark that you can add to your browser, and by clicking on it, the bookmark injects jQuery in to the web page, allowing your to play with jQuery against the web page's markup. By "play" I mean test jQuery selectors via a console. Firebug is one type of console, which we'll look at in the next section, but Safari and IE8 also include consoles that you can run commands directly in to the browser.

Installation is very easy:

1. Navigate to http://www.learningjquery.com/2009/04/better-stronger-safer-jquerify-bookmarklet
2. Drag the "jQueryify" link to your bookmark bar. If your bookmark bar isn't visible by default, in Firefox click "View"
3. Now by clicking the jQuery bookmark and the page you're currently on will have jQuery loaded in to the page

This means that regardless of whether or not the page you're currently looking at contains the jQuery library, once you click the bookmark, you can start trying out jQuery selectors and functions to play around with jQuery. This is the perfect tool for starting out and testing jQuery selectors or testing the effect of modifying the DOM and how to navigate around. My preferred environment for playing with the jQuery bookmarklet is Firebug, the Firefox plugin, which we'll look at how to install and use next.

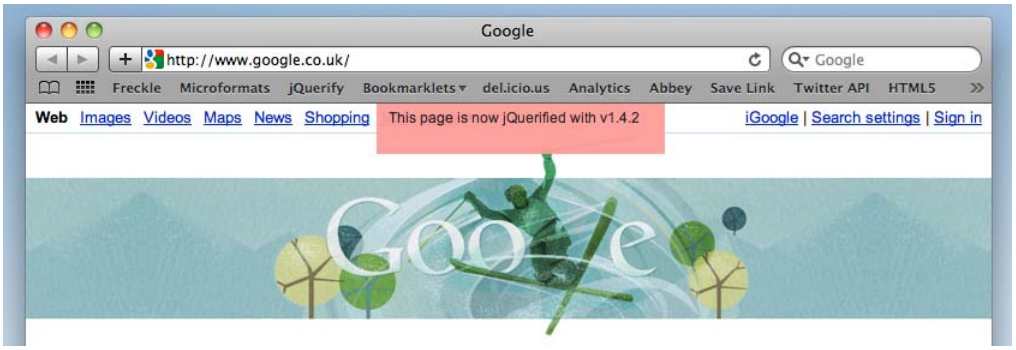


Figure 3.3 – The Google homepage jQuerified using Karl Swedberg's bookmarklet

Debugging with Firebug

Firebug is possibly one of the most useful tools for a developer out there, we've come a long way since using alert boxes to work out what's wrong in our code. Firebug runs as a plugin for Mozilla's Firefox (available for free from http://www.mozilla.com), and also available for free from http://getfirebug.com as seen in figure 3.4.

It's quite possible you're already familiar with Firebug to help you debug CSS and styling a web page. Throughout developing a solution using jQuery, I would use Firebug for a number of different tasks: testing selectors, testing distinct blocks of code, debugging existing code and testing Ajax requests.

Firebug also offers detailed JavaScript debugging features such as stacktraces, breakpoints and introspectors - but that's more for the hardened JavaScript developer and won't be covered in this book. Were you will spend more of your time is the console where we can test code and see the direct result of those tests.

Figure 3.4 – getfirebug.com homepage, changing the way we debug and develop our web pages

Can't use Firefox?

Fear not: you can also use a simplified version of Firebug for Safari, Opera and Internet Explorer using either a bookmarklet, allowing you to open Firebug on any site, or using a JavaScript include.

Head to http://getfirebug.com/lite.html and following the installation instructions.

Using the console

The console is the playground for testing out your JavaScript or jQuery. You need to open Firebug by clicking on the bug icon at the bottom of the browser, and enable the console, as seen in figure 3.5.

For a better debugging environment, expand the area you can type in to, the "command line", so that it's a multiline input box. This can be done by hitting the up arrow at the end of the command line.

Now we can test out blocks of code and check the output in the console (the area to the left).

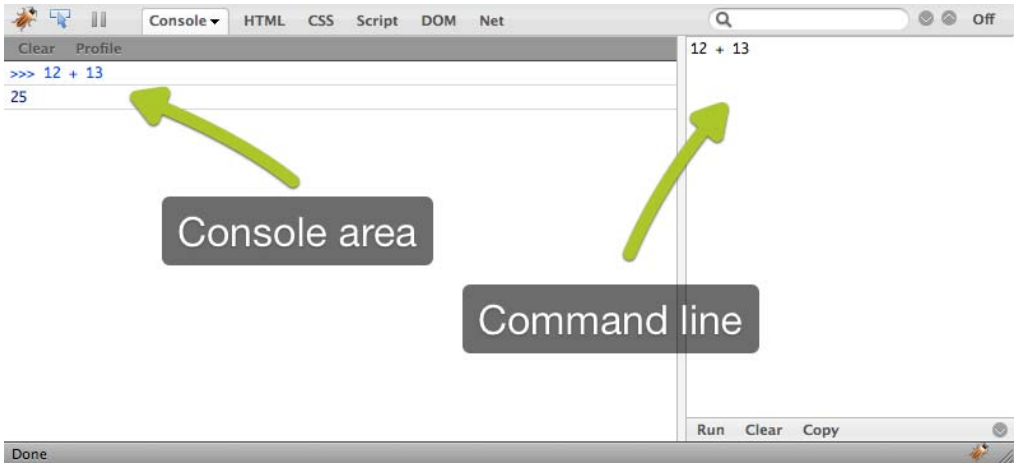


Figure 3.5 - the console and the command line

If we want to log something to this console whilst we're testing our code, we include the following code:

```
console.log( 'my debug message' );
```

Equally we can put variables in the console or results of jQuery calls. For instance to test if a jQuery selector successfully matched something on the page, I could pass this to console:

```
console.log( $('div > a').length )
```

Inspecting Elements

By inspecting elements, I mean that when we can run a jQuery selector in the command line and the result is shown in the console area. These results can look like the markup for the element or may just be a list of the element types. When we move our mouse pointer over the results, it will highlight the element on the page as seen in figure 3.6.

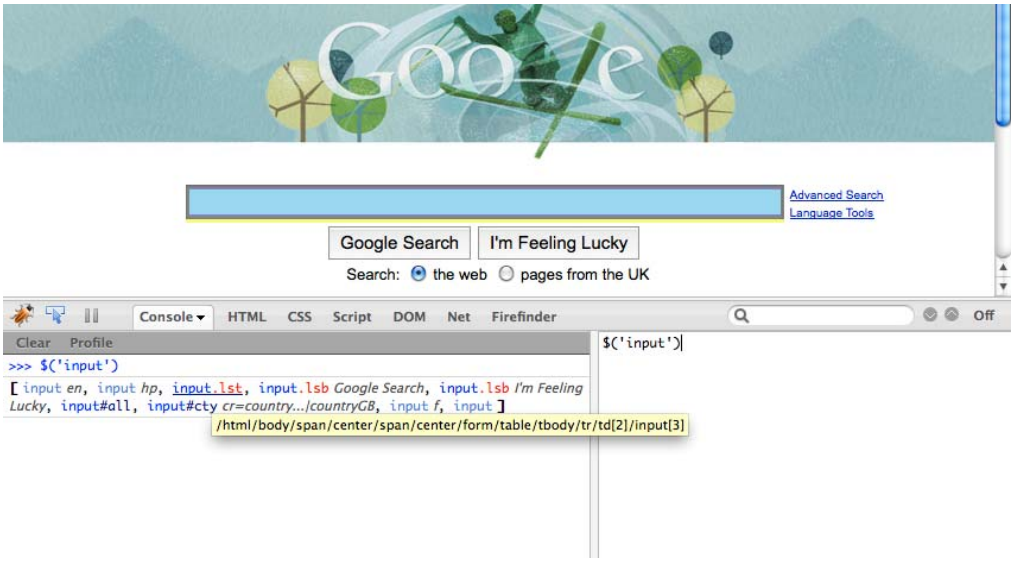


Figure 3.6 – Using Firebug to find all the input elements, and hovering over the console item to highlight the element on the web page

As I mentioned earlier, you may have used Firebug to debug CSS, so in addition to seeing where the element appears on the page, you can also see the margin (in yellow) and the padding (in purple). Also, if we click on one of the result elements in the console, Firebug will switch to the HTML tab and show us exactly where that particular element appears in the DOM tree.

When trying to debug some code I've written or someone else's code, I will always start by testing the selectors in Firebug. If they return results, I'll then double check by hovering and clicking the elements to make sure they're exactly the right element I had expected. This is especially useful if you're creating a jQuery selector on the fly, for example you're constructing the selector based on some variable. In this case, I would use the `console.log` function to output the final result of my dynamic selector, and then test it manually in Firebug to be completely sure I'm working with the part of the DOM I thought I was.

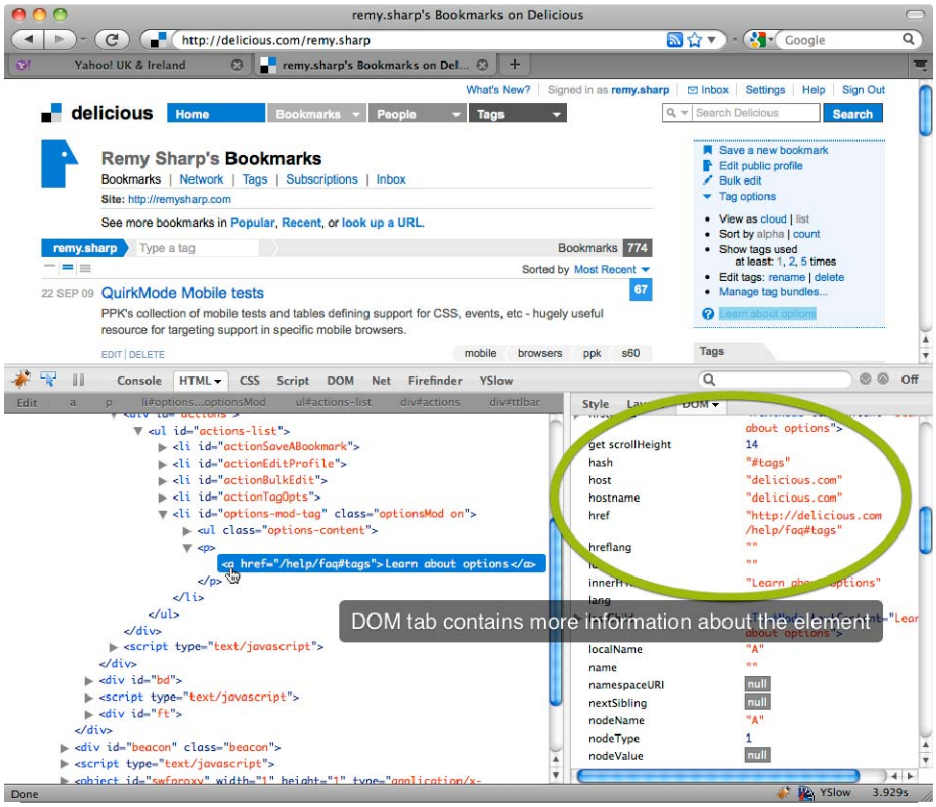


Figure 3.7 - The Firebug DOM tab show more information about the currently selected element

Whilst still on the HTML tab in Firebug, we have the three tabs on the right and of particular interest is the DOM tab, as seen highlighted in figure 3.7. Scrolling through these properties we can find out some interesting information. For example the a elements have DOM properties including href, which is the complete url of the element, but also has hash which is the fragment in the url from the # (hash) symbol, in the example from figure 3.7, the url is http://delicious.com/help/faq#tags and the hash is #tags, which if the element had been clicked on and we caught that event with jQuery, we could access the #tags value using `this.hash` - something we'll explore further and show you how you can exploit in Chapter 4.

Along with jQuery selectors and inspecting elements, the command line and console is the perfect environment to try out your experimental code, anything from functions to arrays to objects can be trialed in the command line. In the next section, we'll look at the a few of

the basic bits of JavaScript that you'll keep coming across in jQuery so that you can happily fake your way around arrays, objects and the special keyword "this".

Basics of JavaScript syntax

Although I expect you've already come across JavaScript in some shape or form, we're going to be covering a lot of JavaScript in this book and some conventions that you'll need to know.

There's lots of books dedicated to the subject of JavaScript that can help you get a great knowledge of the language, but this section of the book aims only to give you a very high level overview of bits you'll need to know for the upcoming chapters. In a way, we're want you to be able to fake your way around JavaScript so that you have enough to understand and expand on the recipes we'll see in Part 2 of jQuery for Designers.

Syntax style

Douglas Crockford, one of the leading individuals on the topic of JavaScript and has written up a set of code conventions for the style of JavaScript. It's what I'll be using throughout all the code examples, and I strongly recommend that you have a look and whether you're working on your own or in a team, start using code conventions because it makes working with code much easier to read and maintain as time goes on:

http://javascript.crockford.com/code.html

The basic variable types are strings (anything inside of single or double quotes), numbers and booleans (true or false). Beyond this we'll look at different arrays and objects, how they can be mixed I'll explain variable scope as it can cause some pain if it's not fully understood. I'll also explain what the keyword this is and how it works along with what an event is since a lot of jQuery works around events triggering.

In the previous section I showed you how to debug with Firebug, and how it will become an invaluable tool for trying out your code.

Arrays and Objects

When you need to use a variable of any type, it should be prefixed with the var keyword. This will tell JavaScript that a new variable should exist. You can, and should, also set the type of variable in the same line.

For example, the old way to create a new array would be to use the following line of code:

```
var items = new Array();
```

However, modern coding techniques don't use this syntax, and in fact the best and right way to write this is as such:

```
var items = [];
```

Square brackets means array and you'll see this method in lots of online examples and other books.

An array is just an ordered list of items that is "zero indexed". This means that if we want to access the first item, it's index is zero. For example, we can put the following code in a script tag and it will popup an alert box with the name "green" as show in figure 3.8:

```
<script type="text/javascript">
  var items = ['red', 'green', 'blue'];
  alert( items[1] );
</script>
```

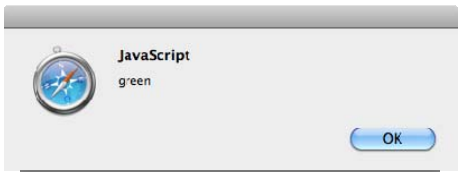


Figure 3.8 – result of alerting an item in an array

Figure 3.8 should be centre aligned

If you've ever dabbled in PHP, then you'll know that you can use a label for the index of an array. JavaScript calls these types of variables "objects". An object is like an array, except that you can use named keys to access the values. So instead of using zero or one, you can use the word "height".

The old way to create a new object follows:

```
var item = new Object();
```

As with the array creation, object is also better written in a different way:

```
var item = {};
```

Here we've used curly brackets. Whereas with an array, we could comma separate the values, with an object, the keys are joined to the value by a colon, and each of these pairs are separated by commas. For example, the following code will popup an alert box with the value associated with the height key as seen in figure 3.9.

```
<script type="text/javascript">
  var item = { height : '60px', width: '100px' };
  alert( item['height'] );
</script>
```

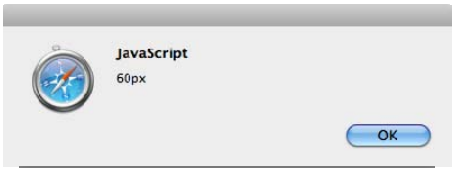


Figure 3.9 – a simple object property being show in an alert box

The object values can also be accessed using a dot, it has the same result in the code, it's just a different way to write the code. So instead of using items['height'], we can also write the same code using:

```
var item = { height : '60px', width : '100px' };
alert( item.height );
```

This is exactly the same syntax and as you find more online examples and plugins you'll see both ways being used. The only time that you have to use quotes around the key is when the key is a reserved word.

Once you have a basic understanding of these two structures you can combine them to create more complicated variables.

For example, you can have an array of more than one object:

```
var items = [
  { name : 'first', height : '60px', width : '100px' },
  { name : 'second', height : '120px', width : '75px' },
  { name : 'third', height : '180px', width : '50px' }
];
alert( items[1].name );
```

The Keyword this

The keyword `this` can be a confusing point for newcomers to both JavaScript, but also jQuery. `this` will appear extensively throughout jQuery tutorials, online examples and throughout this book, so it's useful to either know what it means, or know how to find out what it means.

The keyword `this` is a special variable that is read only, i.e. you can't set the value yourself as the developer. The easiest way to understand what `this` is, is to log it out to a debugger. We've looked at Firebug in detail in the previous sections, but for now, when we want to know what `this` contains, we'll send it to the `console.log` (assuming you have Firebug installed). This can be done in your code using `console.log(this)` or directly using Firebug as seen in figure 3.10:

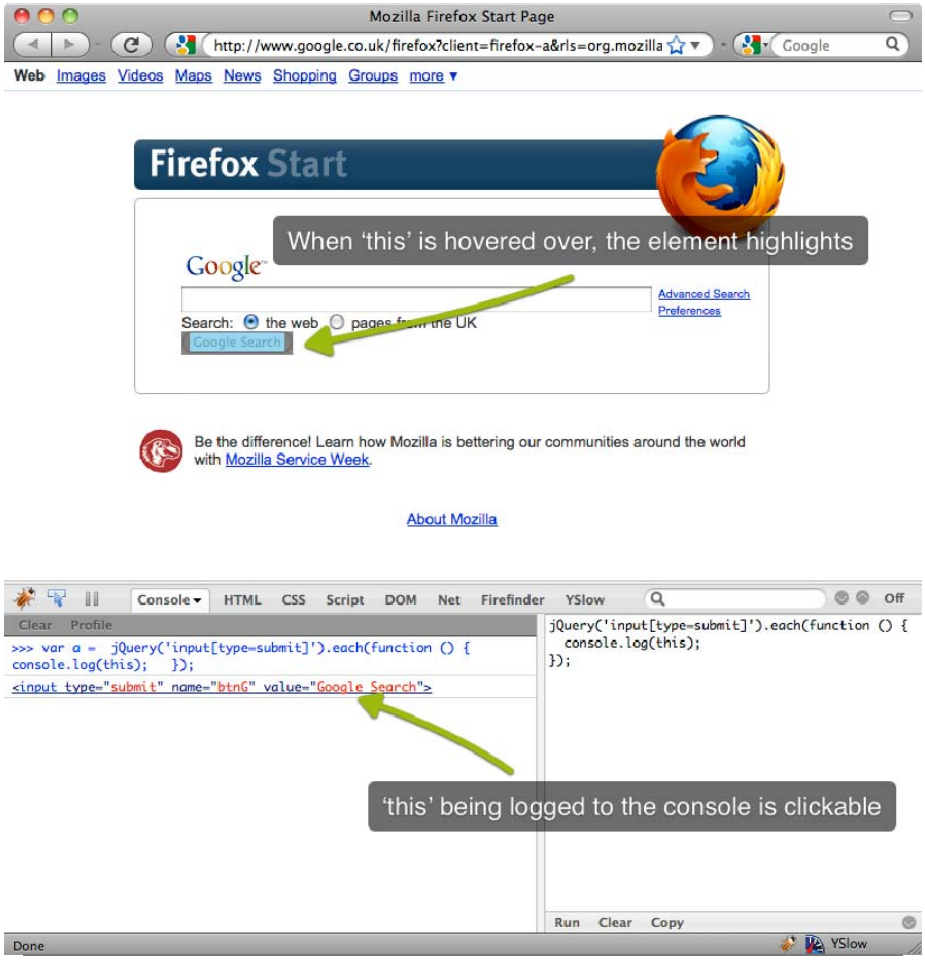


Figure 3.10 - Example of using Firebug to examine the keyword "this"

In most cases with jQuery, the keyword this will refer to an actual DOM element. A simple example is if you attach a function to run when a link is clicked, this will refer to the actual link that was clicked.

Important: remember that the keyword this is just a DOM element. It doesn't have any of the jQuery functions, such as adding a class or changing the CSS. To use jQuery against the current element, this, you must first wrap it in jQuery, then call the function, as you can see below

```
$('a').click(function () {
  $(this).css('background', '#c00');
});
```

Equally, if you want to access properties from that DOM element, you can go right ahead and access them from this, such as `this.href` would be the href to an a element.

If you're not sure, or your code is producing unexpected results when you're working with this, then the safest thing to do is to log it out to Firebug and make sure you're working with exactly the right element you thought.

Summary

Now you're versed with what jQuery looks like and the all powerful dollar function. We've looked through the JavaScript syntax that we'll see lots more of throughout this book tucked away under lashings of jQuery. The JavaScript language is a funny little thing that's taken many developers some time to wrangle, so don't worry if you don't get it completely. We'll be using the same patterns of code again and again throughout this book.

You've also been introduced to the Ninja's Swiss army knife: Firebug. Make sure you keep trying out jQuery inside the console and try visiting your favourite web sites and loading the jQuerify bookmarklet and play with jQuery in Firebug on those sites.

Finally I've introduced you to `this`. The `this` keyword is possibly one of the most confusing attributes of jQuery to someone coming from a non-coding background. So if it all clicked first time around – great work. It took me a while to get a good handle of what `this` meant at different points in the code, so always make use of the `console.log` functionality to discover what `this` is throughout your code. We'll be seeing a lot more of `this` so you'll soon be a master.

This chapter concludes part one of jQuery for Designers. You have all the background knowledge you need to start cracking on and solving typical web interaction problems – which part 2 is bursting with. Moreover we'll be taking common problems and learning how to recognise how they work, their solutions and how to twist and shape the solution into something that's bespoke for your own clients and customers using jQuery.

4

Tabbing Systems

Showing and hiding is one of the fundamental techniques for creating interactions on a web page. The most common design pattern is the tabbing system, which I'll show you in practise in section 4.2.

In fact nearly all the show and hide solutions I've seen and worked with are really, behind the scenes, tabbing systems. Tabbing systems just show and hide on steroids, but often people don't realise this.

A lot of the requests I get running the jQuery for Designers blog, is for tutorials explaining some cool effect where content is revealed in some fancy fashion. Upon some investigation, I find the effect is nearly always a tabbing system - it's just that people don't immediately realise, or understand that tabs can be presented in different ways as seen in figure 4.1. A tab usually visually is represented in the same way a folder of paper would have tabs. Something that sticks out at the top, allowing me to jump to a particular spot in content. You know what a tab looks like. But there's also tabs that have different looks. The don't have to be poking up. For example, figure 4.1 shows three examples of tabbing systems. The first has thumbnail screenshots that once clicked show a larger screen in the centre of the page. Under it's skin, this is a tabbing system. The second show tabs along the bottom of the page and the third example from Apple show how the content can be cross faded in to view – these are all *some* different ways to present tabbing systems.

Figure 4.1 Example of tabbing systems where it may not be immediately obvious the underlying design are tabs

We'll be looking at traditional tabbing system, then adding features such as 'back button support', permanent links (permalinks) and then how jQuery UI can solve this problem.

I'm also going to show you how an accordion effect still shows and hides content, but is ultimately still a tabbing system, just with a twist.

By the end of the chapter you will have an understanding of how to create the underlying tabbing systems, how to implement different show and hide methods, how to support more advanced interface features (such as the back button!) and how to present this in different ways visually.

All of this is done using semantic markup and relatively little jQuery since we'll just be manipulating the DOM and binding what's essentially show and hide functionality.

First of all, we're going to look at the simplest show and hide, and how we can use jQuery's built in animation methods to easily create what would typically be a complicated transition running two sliding animations at once.

Different ways to reveal content

Our example is a competition entry form where we want to include the terms and conditions on the page, but we don't want to initially clutter the page with the terms. This is what you would call a tabbing system, but in this example, we'll outline the fundamentals of a tabbing system, showing and hiding content when a link is clicked. These are the building blocks to tabbing systems, so it's useful to understand how and why this works.

To do this, we want to hide the terms and conditions from the user, and only when they click on a link, do we reveal the terms. In addition, we only want the link to be clickable once, so we want to hide the original link the user clicked on.

Using what we know from Chapter 2 and progressive enhancement, we will deliver the page with the terms visible, so that if there's no JavaScript enabled, the user can still navigate to the terms. If JavaScript is present we only want the terms to be visible if the user clicked the link. Figure 4.2 shows the states of animation the user will see when the click on the terms link.

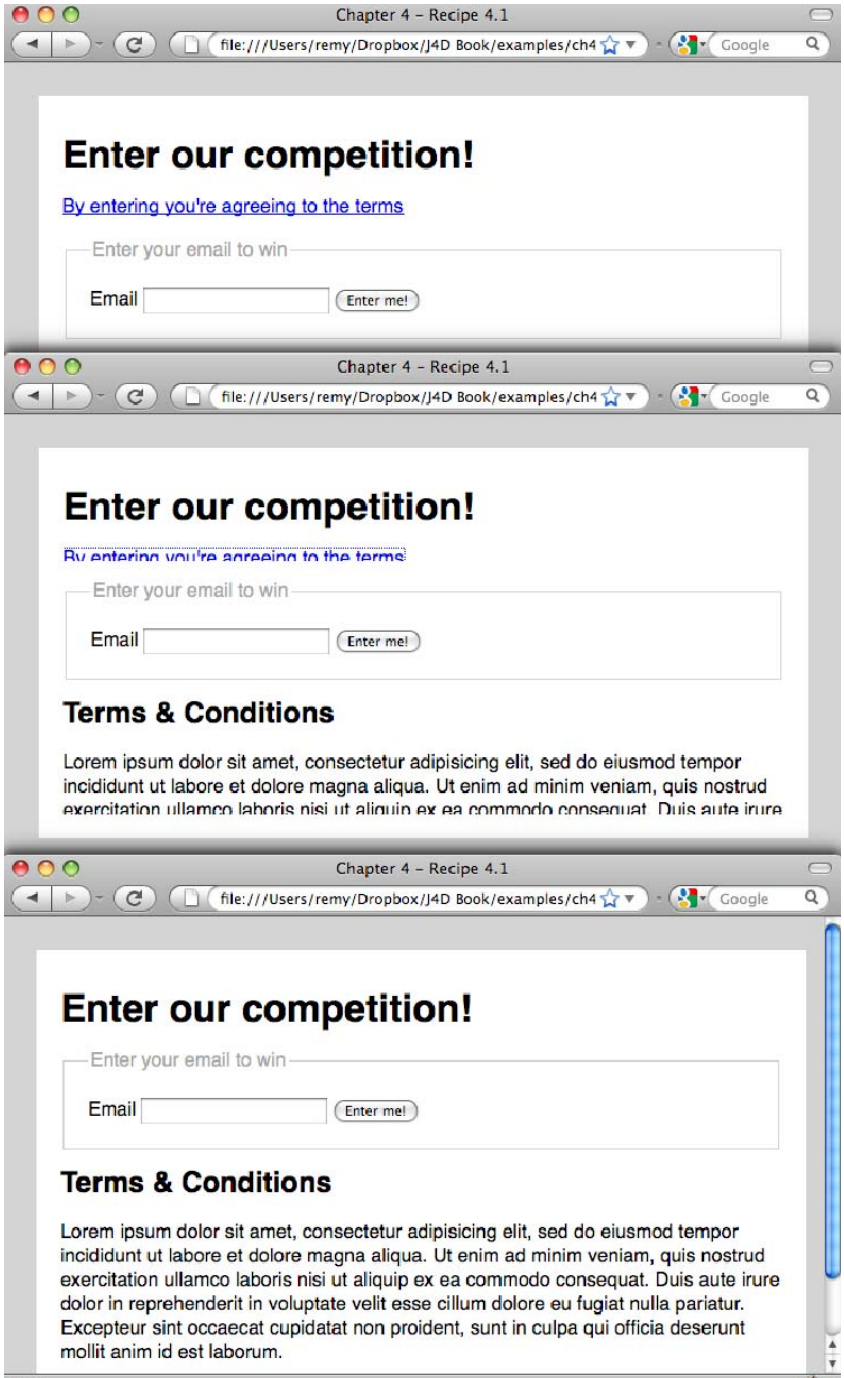


Figure 4.2 - The different states of the page as the user reveals the terms and conditions

Solution

The solution in listing 4.1 solves the problem by achieving the following effect:

1. We hide the terms using JavaScript rather than using CSS
2. When the terms link is clicked - 2.1. Reveal the terms and conditions 2.2. Hide the link we just clicked on

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Chapter 4 - Recipe 4.1</title>
<link rel="stylesheet" href="style.css" type="text/css">
<script src="jquery.js"></script>
<script>
$(document).ready(function () {
  $('#terms').hide();
  $('a.showTerms').click(function () {
    $('#terms').slideDown();
    $(this).parent().hide('default');
    return false;
  });
});
</script>
</head>
<body>
  <div id="content">
    <h1>Enter our competition!</h1>
    <p><a class="showTerms" href="#terms">By entering you're agreeing to
the terms</a></p>
    <fieldset>
      <legend>Enter your email to win</legend>
      <label for="email">Email</label> <input type="text" name="email"
value="" id="email">
      <input type="submit" value="Enter me!">
    </fieldset>

    <div id="terms">
      <h2>Terms &amp; Conditions</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum.</p>
    </div>
  </div>
</body>
</html>
```

Discussion

We've taken a very specific approach to solving this problem, and one key thing to note is that we've hidden the terms and conditions when the page is viewed in a browser supporting JavaScript.

Hiding on load if JavaScript is present

There are a few different methods to hiding content if JavaScript is enabled. One alternative is to add a class to the body tag, such as "js", and use CSS classes to determine which elements should be hidden upon loading. If does mean we need to run a simple bit of JavaScript as the first thing in the body tag, e.g.

```
<body>
<script>
$('body').addClass('js');
</script>
<!-- page continues -->
```

The jQuery could be substituted for the following pure JavaScript solution:

```
document.body.className += ' js';
```

The above two blocks of code should be part of the sidebar

As with most (if not all) our recipes, our jQuery runs once the document has finished loading. Once the document has loaded, and before any images are loaded, we first hide the element with the id #terms. This is instant and with no transition:

```
$(document).ready(function () {
  $('#terms').hide();
...
```

Next we attach a click handler to the anchor element with the class showTerms. When the link is clicked, the function is executed. The function is an anonymous function that's created and passed to the click handler. For more explanations of passing functions to event handlers (in this example a click handler), see Chapter 3 - Tools of the Trade.

```
$('a.showTerms').click(function () {
    ...
});
```

When the link is clicked we do three things:

1. Reveal the terms and conditions
2. Hide the link the user clicked on
3. Cancel the browser's default action for link clicking

To reveal the terms we use the slideDown effect. The first argument of the slideDown method is the duration which the effect should run. If this is omitted it uses the "default" speed, which is 400 milliseconds. By omitting the duration, the effects run at the same speed so that it feels like the effects are directly related (which they are):

```
$('#terms').slideDown();
```

Speeds

jQuery animations and effects take milliseconds as a argument. You can also use "slow": 600 milliseconds and "fast": 200 milliseconds.

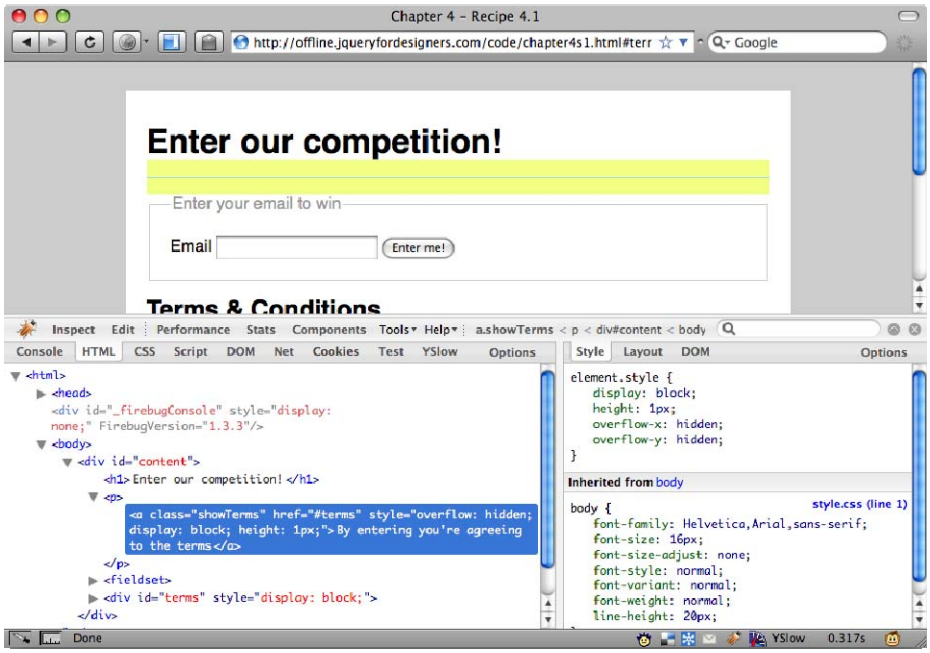
Next we hide the link clicked. You will notice that we've used the keyword this - which refers to the current element (in this case - that was clicked) and that we don't run the effect on the anchor element, but the parent:

```
$(this).parent().hide('default');
```

By using this it ensures that the effect only applies to the element we just clicked rather than any other element on the page.

You will also see that we're passing in the default speed to animate the hide effect. By passing a duration to the hide (and show) methods they don't just flip back and forth between hidden and visible. They animate the height, width, margins and opacity all at once. This means the link fades and slides away out of view when it's clicked.

If we called the hide method without calling parent first, the effect would hide the anchor. This is fine as it animates the height of the element, but when it goes from zero height (via the style attribute on the element: `style="height: 0;"`) it will then jump to hidden via `style="display: none;"`. The effect of this causes the page to visibly jump. This is because of the margins from the surrounding paragraph element having an effect on the flow of the document as seen in figure X-XX.



So to fix this, we apply the effect on the parent element (the paragraph) so that the hide method also animates the margins to zero, and thus creating a smooth hiding effect.

Finally, we cancel the browser's default action to change the URL to the href of the anchor and to make the page jump to the particular element (because we know in our design the height is fine):

```
return false;
```

All of these combined result in a smooth transition revealing the terms and conditions, and the link that was clicked, fades and slides away out of view as it's no longer required, as seen in figure 4.2 above.

Creating a simple tabbing system

A typical tabbing system consists of the following aspects:

1. A list of links to access the content
2. The links jump to the content or reload the page if JavaScript is disabled
3. Block elements with the content, whose id attribute may be referred to in the links from (1) - we'll call these tab panels
4. With JavaScript enabled, all the tab panels are hidden, then the selected panel is shown

This content can be pre-loaded in the markup, or could be dynamic via an Ajax request, it doesn't really matter. The system is hide all tab panels, then show the relevant panel. As we're designing a simple system, we're going to use the hash part of the href in the link to find the appropriate content block by the id attribute.

URL Hashes

The hash is everything from the hash (or pound) symbol in the URL, including that symbol:

http://jqueryfordesigners.com**/#this-is-the-hash**

The task is to bind click handlers to each of the tab links. To collect all the possible tab panels and to be able to hide and show those panels.

Solution

I'm using the same template as used in listing 4.1, just replacing the main markup and all the JavaScript. Obviously we'll still need to include the jQuery library (as is true for all our examples).

MARKUP

Listing 4.2: Basic markup for a tabbing system

```
<ul class="tabs">
  <li><a href="#copyright">Copyright notice</a></li>
  <li><a href="#terms">Terms and conditions</a></li>
  <li><a href="#accessibility">Accessibility statement</a></li>
</ul>
<div id="copyright">
  <h2>Copyright</h2>
  <p>...</p>
</div>
<div id="terms">
  <h2>Terms and conditions</h2>
  <p>...</p>
</div>
<div id="accessibility">
  <h2>Accessibility statement</h2>
  <p>...</p>
</div>
```

JQUERY

Listing 4.3: Simple jQuery to create a tabbing system

```
$(document).ready(function () {
  var $tabs = $('ul.tabs a');
  var panelIds = [];

  $tabs.each(function () {
    panelIds.push(this.hash);
```

```
    });

    var $panels = $(panelIds.join(','));

    $tabs.click(function () {
      $tabs.removeClass('selected');
      $(this).addClass('selected');
      $panels.hide();
      $panels.filter(this.hash).show();
      return false;
    }).filter(':first').click();
  });
```

Discussion

The very first thing we will see from the markup is that without the jQuery code, it still makes semantic sense, and the functionality of the page still works, i.e. the links jump to the content.

To explain the jQuery in (fairly) plain English, we are:

1. Collecting all the tab links
2. Creating an empty collection where we'll store panel id attributes
3. Loop round each tab link, collecting the hash and storing it
4. Collect all the panels
5. When any tab link is clicked:
    a. Remove the 'selected' class from the tab links
    b. Add the 'selected' class to the current
    c. Hide all the panels
    d. Find the panel that is related to this link and show it again
    e. Cancel the default browser action
    f. Let's look at each line of code, during which I'll explain what we expect the code to be doing.

Our code will only start to run when the document has fully loaded the DOM, which means we have access to all the elements on the page to bind to or manipulate as we want:

```
$(document).ready(function () {
```

Next we initialise two variables. The first is using a CSS selector: `ul.tabs a` which returns each of the tab links in our tabbing system.

The second creates a new empty array for us (remember that `x = []` this is shorthand notation for `x = new Array()`):

```
var $tabs = $('ul.tabs a');
var panelIds = [];
```

Typically I would want to declare all my variables at the top, and this is the best practice. However, for this example, for readability's sake, I've decided to collect the panelIds before creating the new variable `$panels`.

I could reorder the code so that all the declarations happen at the top, but we have to keep in mind that our code may be read and debugged in years to come, and possibly by a junior developer who has recently joined our team.

With that in mind, the next step is to loop through each tab link collected the hash. The really great thing about hashes on links is that they work as ID CSS selectors.

Using the `$tabs.each()` method loops round each item in our jQuery collection. Within the function we pass in to each, this refers to the specific, current DOM element.

```
$tabs.each(function () {
  panelIds.push(this.hash);
});
```

After the each loop completes, we have the panelIds array populated to look like this:

```
['#copyright', '#terms', '#accessibility']
```

We now need to convert this to a string to create a valid CSS selector, so it looks like this:

```
'#copyright, #terms, #accessibility'
```

Which is achieved using the join array method. Since we don't need this again, I'm passing the CSS selector string straight in to jQuery and capturing the output:

```
var $panels = $(panelIds.join(','));
```

Finally, we bind the click handlers to the tab links so that when a link is clicked, all the panels we discovered are firstly hidden, then we find the panel we wanted to show, and display it.

The last thing we do in our click handler (and possibly in a lot of cases with click handlers), is to return false so that the default browser action is cancelled, in this case, to jump to particular element. Note that this breaks the back button, but we'll come on to solving that problem in section 4.6.

```
$tabs.click(function () {
  $panels.hide();
  $panels.filter(this.hash).show();
  return false;
...
```

The one line I want to re-examine is the one where we show the correct panel. We're using chaining to say: from the `$panel` jQuery collection of DOM elements, filter down to only those that match the selector. In this instance, the selector we're using is:

```
this.hash
```

Since this is the current element (link in our case), and the hash matches CSS selector formats, when we click the copyright link the code being run looks like this:

```
$panels.filter('#copyright')
```

Then using this result, i.e. the single panel with the ID attribute of 'copyright', we call the show method.

Finally, we need to trigger the first tab link to be clicked, which will cause the tabs to setup properly and only show a single panel. This is achieve by chaining after the `$tabs.click` call:

```
$tabs.click(function () {
```

```
  ...
}).filter(':first').click();
```

The filter narrows the collection of links to the first element, and triggers a click. The `:first` selector is a jQuery specific selector that's a useful shortcut to the first element in a collection.

This entire system can be reused and presented in lots of different ways, and it's what we refer to as a tabbing system, also known as a tabbing design pattern.

Next we will look at how we can expand on this system and integrate the tabs further in to a web page, allowing uses to click on other links to access tab panels, link directly to a page and show the associated tab and support the back button.

Controlling tabs from other links

Now that we have our tabbing system in place, another common requirement is to trigger a tab *being* shown from a link that *isn't* particular an actual tab. This also applies if we have a link inside a panel that wants to link to another panel.

To solve this we need to find all the links that have a hash that matches one of our panel IDs, and say that when it's clicked, trigger a click on the associated tab link. We could repeat the code from listing 4.3, but by repeating code we risk incorrectly duplicating the code. For example, when you return to the code in years to come, or a new employee has to support the code, they may not realise that the code appears twice. So, we trigger the click handler on the *real* tab link.

One final requisite of the code, it must ignore the real tab links, which we'll see in the solution.

Solution

An arbitrary link on the page can trigger the associated tab to show.

```
<p>For more details see our <a href="#copyright">copyright</a> notice.</p>
```

The following code would be appended to listing 4.3 inside the $(document).ready() function.

Listing 4.4: Adding the functionality to click on any link to trigger the tab functionality

```
$panels.each(function () {
  $('a[hash=#' + this.id + ']').not('ul.tabs > li > a').click(function () {
    $tabs.filter('[hash=' + this.hash + ']').click();
    return false;
  });
});
```

Discussion

First and foremost, listing 4.4 must be appended to the end of the `$(document).ready()` call in listing 4.3. This is because we are reusing the $panels and $tabs variables.

If we put the code outside, or in it's own `$(document).ready()` block we would have to reinitialise these variables.

Just as we did in listing 4.3, we are using the each jQuery method to loop through each of the panels to search for links that point back at the panels:

```
$panels.each(function () {
```

Once we're looping through the tab panels, we need to find links anywhere on the page that link to this panel. However, there is an exception: we need to ignore the existing real tab links.

So to do this we look for all the anchors which have a hash value of `'#' + this.id`, which for the first panel would be #copyright, and then using the jQuery not method, filter out those links that match the selector: `ul.tabs > li > a`.

We have to prepend the # character to the selector because `this.id` is an HTML ID attribute. By itself, it doesn't match the hash value on the href. Since all hash attributes start with a # character, we need to always ensure it's included, either via the variable we search with, or by prepending it. If we didn't prepend the #, the selector would look like this:

```
$('a[hash=copyright]')
```

By including the # character, the hash matches exactly:

```
$('a[hash=#copyright]')
```

Next we use the not method to ignore some of the elements in our collection. The selector is being very specific about the links we need to ignore. The direct descendant selector ensures that we target the tab links, and only those tab links.

The first query (to find all links) and the not filter can be chained together for us to then attach a click handler.

```
$('a[hash=#' + this.id + ']').not('ul.tabs > li > a').click(function () {
```

An important thing to keep an eye on it the value of this at the different points in the code. Notice that in listing 4.4 we use this twice in different places. It's important to understand what this represents at each point. The first time it's used (as per the code line above), this represents the individual panels that are being looped through.

However, the next line, where we refer to `this.hash`, the value is different because we are inside the click handler. Within the click handler, this refers to the element that has been clicked, i.e. the link.

How to check what *this* is

If you're finding that you have obscure problems with your code, and are possibly confused by what this refers to at a certain point in the code, just log the value out to your console. Remember that Firebug is an extremely useful tool that is there to help!

To help debug the value of this, I would pepper my code with:

```
console.log("my label", this);
```

When this is logged in the console, we can roll our mouse over the element and it will highlight the element on the page.

Note that the code above should be part of the sidebar/callout

Figure 4.1: example of this logged and highlighted using Firebug

With this in mind, inside the click handler we search for the tab links that has the same hash as the link clicked (`this.hash`), and trigger a fake click on the link.

Since we've already captured the tab links in $tabs, we can filter the jQuery collection down just the link we want.

```
$tabs.filter('[hash=' + this.hash + ']').click();
```

The hash selector is a custom jQuery selector (as CSS2 attribute selectors run from the source document, rather than the final rendered DOM). This allows us to find a link based on only a part of the href: specifically the hash part.

We're going to reuse exactly this technique of searching for a link using the hash to support the permalinks, as seen in the next recipe.

Permalinks to tabs

A permalink is a permanent link to a specific resource. However, because we're using jQuery to create dynamic tabs, if we always default to show the first tab, a permalink may not be entirely correct. For example, if I'm sent to the following URL: http://addons.mozilla.org/en-US/firefox/#t-Reference, I expect that the Mozilla Add-ons page should load and show me the Reference panel in it's tabbing system, as seen in figure 4.2.

Figure 4.2: Mozilla Add-ons site with permalink to reference

In listing 4.3, the very last job of the code was to trigger the first tab to be selected:

```
$tabs.click(function () {
  ...
}).filter(':first').click();
```

However, to support permalinks, we need to check the URL the user arrive on, and then if they intended to view a tab panel, show the appropriate panel, otherwise show the first as usual.

Solution

Listing 4.5: selecting the correct tab by first checking the URL of the page

```
var preSelectedTab = ':first';
if (window.location.hash && $tabs.filter('[hash=' + window.location.hash +
']').length) {
  preSelectedTab = '[hash=' + window.location.hash + ']';
}
```

```
$tabs.click(function () {
  ...
}).filter(preSelectedTab).click()
```

Discussion

Within the global object window (which is available on every page in a browser), is the `window.location` object. This contains information about the URL that the user requested.

If I request a URL, and log out the `window.location` I can see all the properties that are available to me:

```
>>> console.log(window.location);
http://jqueryfordesigners.com/jquery-infinite-
carousel/?action=submit_comment#allComments
>>> console.dir(window.location);
hash        "#allComments"
host        "jqueryfordesigners.com"
hostname    "jqueryfordesigners.com"
href        "http://jqueryfordesigners.com/jquery-infinite-
carousel/?action=submit_comment#allComments"
pathname    "/jquery-infinite-carousel/"
port        ""
protocol    "http:"
search      "?action=submit_comment"
```

In particular, we see that the hash property is also available in the object, and as a CSS selector. So we can use this to filter through our tab links and select the one the user requested when they visited the page.

By default, we want to show the first tab, so our filter selector variable will store that initially:

```
var preSelectedTab = ':first';
```

The next task is to assert that:

1. There is a hash in the URL
2. Whether there is a tab link that matches that hash

The second part of the test is really the most important part, since it's checking if that tab actually exists to be clicked on.

This is solved by using the jQuery filter on the $tabs collection, and testing whether the selector found any elements, using jQuery's length value.

If found, then we update the preSelectedTab to be the hash selector:

```
if (window.location.hash && $tabs.filter('[hash=' + window.location.hash +
']').length) {
  preSelectedTab = '[hash=' + window.location.hash + ']';
}
```

Finally we use the preSelectedTab instead of `:first`, in the final filter and click, so that the right tab panel is shown:

```
}).filter(preSelectedTab).click();
```

Now it's great that we can link directly to page and the right tab turns up, and we can click through the different tab links and show the right panel, but what about when the user clicks on the back button? With our current solution, the browser will navigate away from our page.

Really the user expects to go *back* in our application, i.e. show the previously viewed tab. In the next section I'll show you how to readdress your solution so that it supports the back and forward browser buttons.

Supporting the back button

To support the back button in our tab system, we're going to employ the support of a plugin called *hashchange* written by Ben Alman. The plugin gives us a new event, just like a click event, but instead it fires when the browsers url changes. So if I navigate from http://example.com/ to http://example.com/#terms the hashchange event runs (think of this just like the user clicking on a link and it running the click event), they can even jump to a specific point in the history via the browser as seen in figure 4.XX.

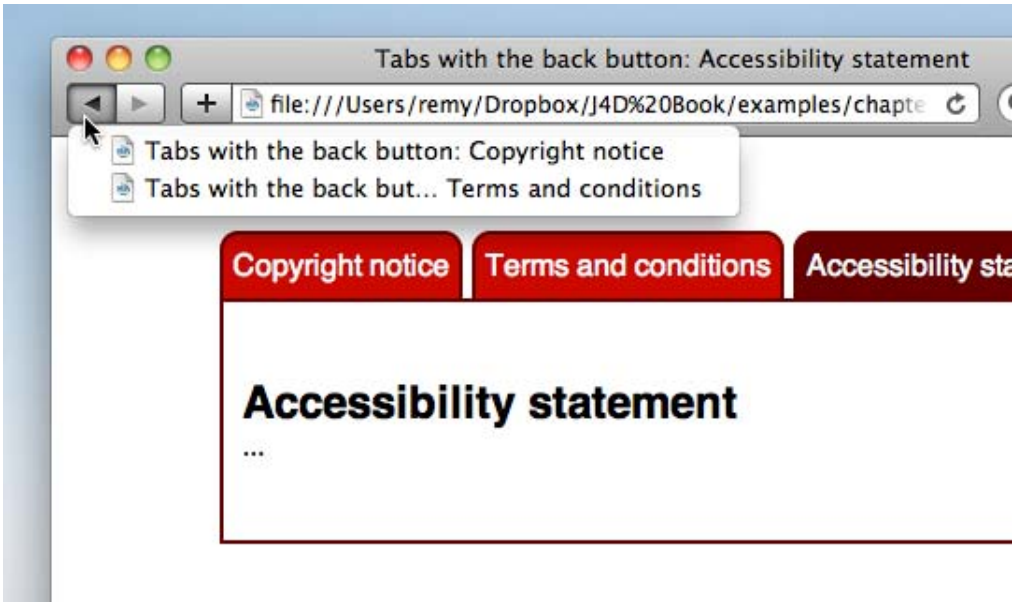


Figure 4.xx – With the hashchange plugin we can support the native back button properly

Taking the code from our first version of a tabbing system in listing 4.2, we have to modify it so that we're no longer tracking when the user clicks a tab, but when the url changes. Fundamentally the code is very very similar. The only exception is that because we're not running our when the user *clicks* the tab, we need to go and find the link that was clicked retroactively. I'll explain this as we look at the code and explain the differences in our listing below in 4.6 compared to 4.2. Also, a free benefit of using Ben's plugin is that our new back button supported page will also allow users to permalink to specific selected tabs.

Solution

Using the same tabbing template (copyright, terms and accessibility) and including the hashchange plugin directly below jQuery:

```
<script src="jquery.hashchange.min.js"></script>
```

Our solution follows:

4.6 Adding back button support to tabbing systems

```
$(document).ready(function () {
  var $tabs = $('ul.tabs a');
  var panelIds = [];

  $tabs.each(function () {
    panelIds.push(this.hash);
  });

  var $panels = $(panelIds.join(','));

  $(window).bind('hashchange', function () {
    var hash = window.location.hash || $tabs.first()[0].hash;

    $tabs.removeClass('selected');
    $panels.hide();

    $panels.filter(hash).show();
    $tabs.filter('[hash=' + hash + ']').addClass('selected');
  });

  $(window).trigger('hashchange');
});
```

Discussion

The big change is that we no longer listen for clicks on the tabs. In fact, I've moved all of the code out of the click event handler in to this new `$(window).bind('hashchange', function)` code.

When we listen for the hashchange event we're using the exact same logic as we did in listing 4.2, but the context is different. The context when the tabbing revolved around the links was the individual tab link that had been clicked. When the tab linked was clicked, it fired the click event which contained our code, and in particular, we were able to find which tab we needed to show using `this.hash` as it represents the CSS selector ID of the tab we want to show. Now, since the context *isn't* the link, we need to find the link that was clicked using the information from the window location.

Remember in listing 4.5, I used window.location.hash and we used it as part of our selector. We'll use the same technique here. When the url has changed, so has the window.location.hash. The first difference from listing 4.2 is instead of finding the right panel to show using (where `this` is the link clicked):

```
$panels.filter(this.hash).show();
```

We use the hash from the window location:

```
var hash = window.location.hash || $tabs.first()[0].hash;
```

```
...
    $panels.filter(hash).show();
```

Then using the same technique from the previous recipe, to find a link based on it's hash, we find the links that point to the newly selected panel and we add the selected class to it:

```
$tabs.filter('[hash=' + hash + ']').addClass('selected');
```

The final change we made was to trigger the hashchange event at the end of our document.ready function. There's an important thing not to forget here. When the hashchange event fires for the first time, there's not actually a hash on the url. So when our code runs, it doesn't have any value stored in the hash variable. To fix this, we need to give it a default:

```
var hash = window.location.hash || $tabs.first()[0].hash
```

This says that hash is either the value of the hash on the url, *or* take the hash from the *first* tab that we collected earlier in the code. This version is the *reusable* version of the code. If this doesn't make sense to you or you don't plan to reuse the code, you can hard code the default ID selector using the following, where '#copyright' is my default for my example:

```
var hash = window.location.hash || '#copyright'
```

Then we fire the initial hashchange event:

```
$(window).trigger('hashchange');
```

Now the example is complete, and we can use the native browser back and forward buttons. In fact, this plugin does a lot more for us for free, i.e. without us having to do anything. This method also supports *other* links on the page linking to our tab without us having to add any more code. Since a link on the page linking to #copyright would change the browsers url, and any change to the url is picked up by this code and shows the right panel, it means without any extra code, we've achieved the goals set out in "Controlling tabs from other links" recipe.

In fact, because this code also finishes by triggering the hashchange event, thus reading what's on the browser url, it means that this solution *also* supports permalinks to the tabs. All in all a very sweet little plugin that allows us to achieve a lot (permalinks, tabbing, back button support) in very little code and reduced complexity.

Next we'll look at how jQuery UI offers you a plugin that has all the bells and whistles built in.

jQuery UI Tabs

Looking at the Yahoo! homepage there's a few places where they use tabs, and they're definitely Ajax requests going on to grab the content and show it in each panel.

In figure 4.XXX we can see the News tabs (including Sport, Entertainment and Video) and the Mail tabs (including Messenger, Weather, Music, Local and Movies). In reality, Yahoo! are using Ajax to load the Mail tabs, and not the News. However, we want to take inspiration from their interface, and create tabs on our site that loads content via Ajax.

In addition, I want to look at the different ways in which we can reveal the tab - so far we've only switched it in to view, I want to look at whether we can use a cross fade or a sliding effect to show the panel content.

Figure 4.XX Yahoo uses tabbing systems in all manors of ways, for their search filters, their new tabs, the email and personalized information.

The jQuery UI (http://jqueryui.com) provides a collection of widgets (and effects) out of the box that make a lot of this work very easy. Specifically, we're going to look at using the jQuery UI tabs widget.

There's a few simple steps to getting the jQuery UI widget code, and for our example we're going to get a customised version of the code (i.e. instead of grabbing all the jQuery UI code that would allow us to create all manner of widgets).

Visit the jQuery UI site, and then select "Download" from the navigation at the top. From here we will be presented with a large list of options. The starting point is to "deselect all components". Now all the checkboxes should be unchecked.

We'll scroll down to the 'Tabs' checkbox under the 'Widgets' heading and select it. Once this is selected, the custom builder will automatically work out what the dependencies are,

and including them in the build. Now we can download the build from the button on the right.

Notice also that we're given the option for theme when we download. For now we'll use the default (currently UI lightness). The download should contain a number of folders and files. The index.html file contains samples and demos of all the components that we chose to download.

For our example, I've copied the folders in to a subdirectory called `ch4s6-assets` (chapter 4, section 6).

Solution

I've included the full markup for this solution, as we're starting with a blank sheet and not including the previous recipe solutions:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Chapter 4 - Section 6</title>
<link
href="http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/themes/base/jquer
y-ui.css" rel="stylesheet" type="text/css"/>
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>
<script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-
ui.min.js"></script>
<script>
$(document).ready(function () {
  $('#tabs').tabs();
});
</script>
</head>
<body>
<div id="tabs">
  <ul class="tabs">
    <li><a href="#copyright">Copyright notice</a></li>
    <li><a href="#terms">Terms and conditions</a></li>
    <li><a href="#accessibility">Accessibility statement</a></li>
  </ul>
  <div id="copyright">
    <h2>Copyright</h2>
    <p><a href="#terms">terms</a></p>
  </div>
  <div id="terms">
    <h2>Terms and conditions</h2>
    <p>...</p>
  </div>
  <div id="accessibility">
    <h2>Accessibility statement</h2>
    <p>...</p>
  </div>
</div>
</body>
</html>
```

Discussion

There's a few important items to notice in this solution, but the first you should notice is the jQuery code is significantly smaller - in fact it's just to say: when the document is ready, run the tabs plugin. The rest is handled by jQuery UI.

Out of the box, i.e. without any styling changes or any bespoke code we're able to create a tabbing system that's ready to use straight away, as seen in figure 4.XXX

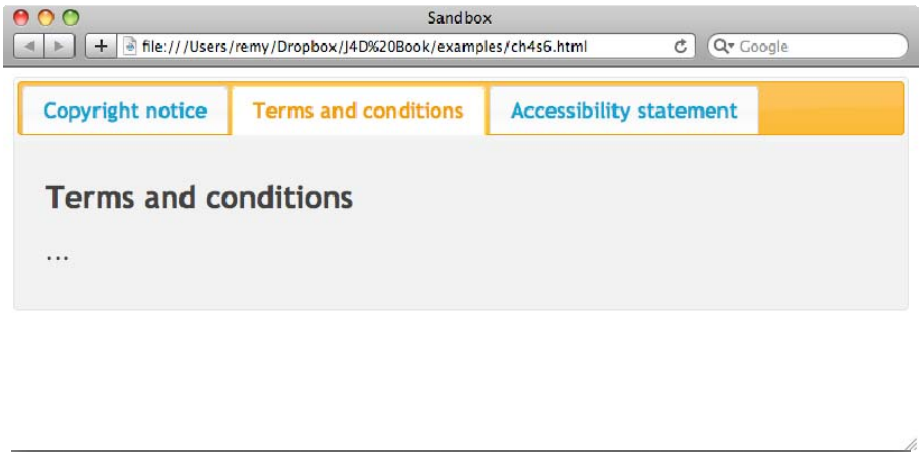


Figure 4.XXX: jQuery tabs UI with the UI Lightness theme applied

The key components of markup in this solution are:

1. The jQuery UI theme stylesheet
2. jQuery itself
3. jQuery UI library (customised to contain the tabs widget)
4. Initialising the tabs
5. Wrapping the tab system in a or appropriate element

Obviously initially the colour scheme may not be to your taste, but you're the designer and the expert on that, and it's all customisable.

If you don't want to work through the design yourself, you can use the jQuery UI themeroller (http://jqueryui.com/themeroller/), see figure 4.XXX, where you can either select a theme from a gallery of predesigned themes or customise the theme yourself.

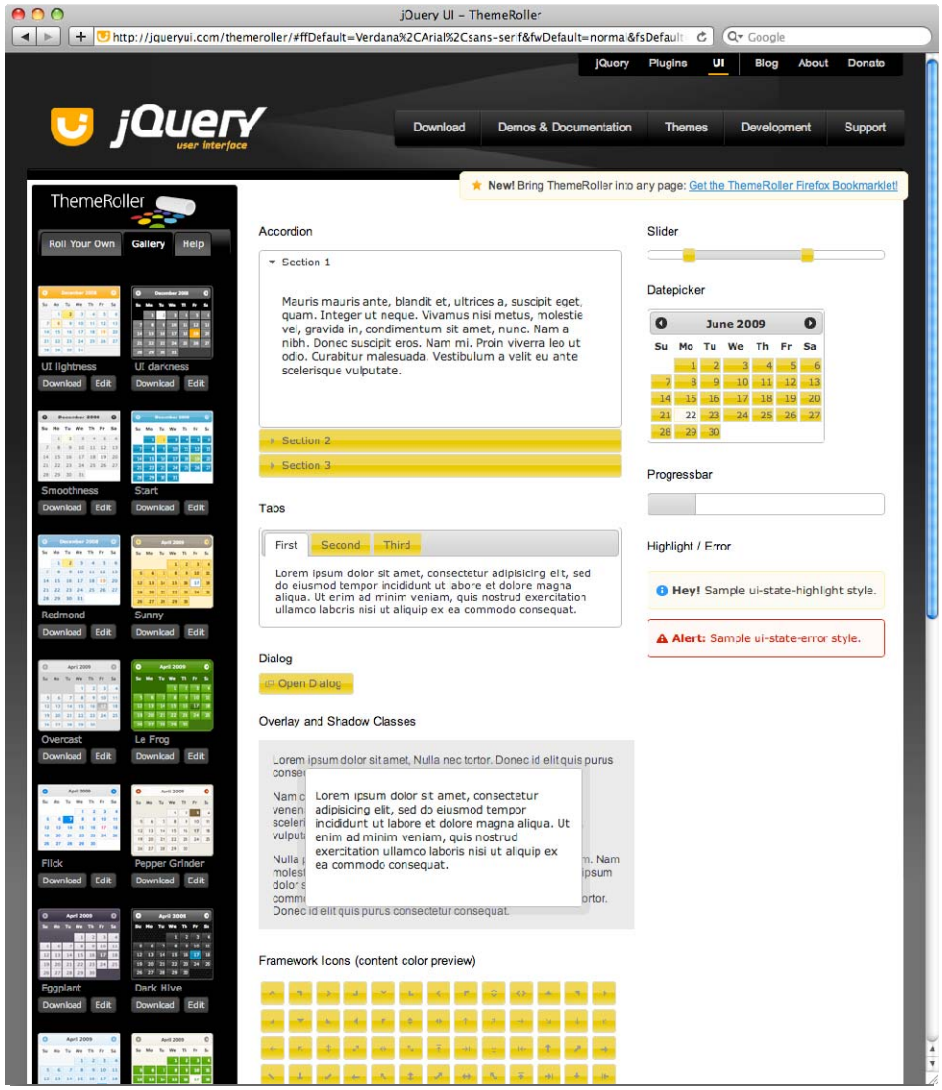


Figure 4.XXX: jQuery UI Themeroller

The only differences between this tabbing system is the prerequisite for the wrapping `<div>` around the entire tabbing markup. This is to allow the tab widget to target a specific block and render all the theme styles around it.

We did say earlier in this recipe that we wanted to support Ajax requests, which the jQuery UI tabs does out of the box for us, but we need to make a very small change to the code that we've written.

AJAX DRIVEN TABS

jQuery UI tabs makes it very easy to add tabs driven from an Ajax request. The only change is the href in the tab link. We need to change this:

```
<div id="tabs">
  <ul class="tabs">
    <li><a href="#copyright">Copyright notice</a></li>
    <li><a href="#terms">Terms and conditions</a></li>
    <li><a href="#accessibility">Accessibility statement</a></li>
  </ul>
  <div id="copyright">
    <h2>Copyright</h2>
    <p><a href="#terms">terms</a></p>
  </div>
  ...
</div>
```

To the following code, where we've removed the tab panels and we've changed the tab link hrefs to point to real URLs rather than IDs of content already on the page:

```
<div id="tabs">
  <ul class="tabs">
    <li><a href="copyright.html">Copyright notice</a></li>
    <li><a href="terms.html">Terms and conditions</a></li>
    <li><a href="accessibility.html">Accessibility statement</a></li>
  </ul>
</div>
```

The jQuery UI tabs widget automatically pulls in the contents from the URL given in the tab link in to a newly create panel. It also means that without JavaScript, this content can still be spidered by Google and other search engines.

ADDING EFFECTS

There are lots of different options available for tabs widget, and even easier to create special effects when revealing the panel.

Although we'll look at effects, specifically cross fades in chapter 6, we can easily apply this effect using the jQuery UI. This would mean that when the user clicks on the tab link, the active panel will fade away, and the newly selected panel fades in to view.

To achieve this, where we apply the tabs plugin in the jQuery code, we write the following:

```
$('#tabs').tabs({ fx: { opacity : 'toggle' }});
```

This simple extra option that we now passed to the tabs plugin means that the special effect will execute when the tabs are shown.

Notice how the term opacity is also a CSS style. This can easily be played with to try out other effects, such height which will slide the panels up and down as they're selected:

```
$('#tabs').tabs({ fx: { height : 'toggle' }});
```

You could try combining these CSS styles or try other CSS properties to prototype and see what kind of effects you can create with the jQuery UI tabs widget.

Simple Accordions

The accordion is the name given to the effect that resembles the musical instrument of the same name, but what do these have to do with tabbing systems? Well, as I've been showing you in some of the figures in this chapter, not all tabbing systems look like tabs. Accordions are fundamentally tabs, where there's a link that shows a panel of content. The difference is that they tend to collapse the visible panels and then expand the requested panel, creating an *accordion* (instrument) effect.

The Apple web site has an accordion to access the different downloads available on their site, grouping in to downloads, top downloads and top Apple downloads, the effect can be seen in figure 4.XXX

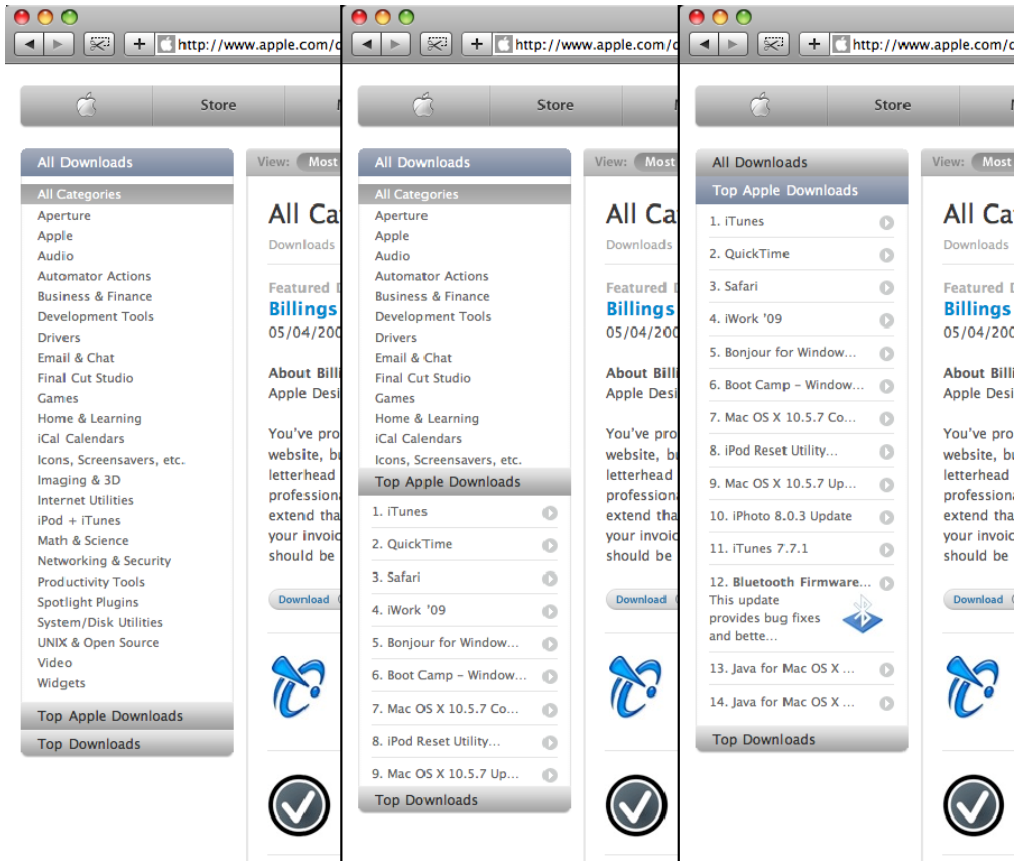


Figure 4.XXX: Example of the accordion in action for the Apple downloads (note that the middle frame was generated, typically the height is equal throughout the effect)

The markup is slightly different compared to a tabbed based layout, but the methods behind the effect are the same.

When an element is clicked, all the panels are hidden, and the selected panel is shown.

With this markup, however, the tab links are organised adjacent to the panel, so the markup reads: tab link, panel, tab link, panel, tab link, panel.

We will use this layout of markup to help us find the panels using the jQuery `next()` as seen in listing 4.8.

Solution

MARKUP

```
<h2><a href="#copyright">Copyright</a></h2>
<div id="copyright">
  <p>Our copyright notice (goto <a href="#terms">terms</a>)</p>
</div>
<h2><a href="#terms">Terms and conditions</a></h2>
<div id="terms">
  <p>Here be terms and conditions</p>
</div>
<h2><a href="#accessibility">Accessibility statement</a></h2>
<div id="accessibility">
  <p>The accessibility statement</p>
</div>
```

JQUERY

```
$(document).ready(function () {
  var $tabs = $('h2 > a');
  var $panels = $tabs.parent().next();
  $tabs.click(function () {
    var $panel = $(this).parent().next();
    $panels.slideUp();
    $panel.slideDown();
    $tabs.removeClass('selected');
    $(this).addClass('selected');
  }).filter(':first').click();
});
```

Discussion

As you can see from the jQuery code, it's very similar to the tabbing system. The main difference, aside from the layout of the markup, is how we collect the tab panels. Since the tab panels are always adjacent to the tab link, or headings, we can use the jQuery `next()` method to find them.

The `next()` call changes the collection we're working with, so that when we call:

```
var $panels = $tabs.parent().next();
```

Our collection goes from:

```
[a, a, a]
```

jQuery works through each anchor element, finds it's parent, and the stores the next element in the DOM, so the collection becomes:

```
[div#copyright, div#terms, div#accessibility]
```

Which is the collection we're working with. Now that we've captured the tab links and panels, we need to bind click handlers to the `$tabs` so that the panels are shown.

Since this is an accordion and not a tab UI, we want to use the `slideUp` and `slideDown` methods to animate the effect like the Apple site has.

Inside the click handler we need to:

1. Find the panel this tab link is supposed to show
2. Animate all the panels up (which should only be one tab in practice)
3. Animate the panel we just found and slide it down (from it's up/hidden state)
4. Manage the 'selected' classes on the tab links

The sequence of selections in figure 4.XXX show how the collection changes when we run this code:

```
var $panel = $(this).parent().next();
```

Where this is the anchor, we see at first we have the anchor, then calling `parent()` moves up the DOM tree, and then finally `next()` moves to the adjacent element.

[TODO INSERT IMAGE OF DOM TREE NAVIGATION]

The next sequence of events are simple, in that the panel has the `slideDown` animation run against it, then the `$panels` collection has `slideUp` (which will only ever run on a single panel).

It's important to note the order, that we're sliding everything up **first** and then sliding the selected panel down, because if we showed the selected panel first, then hid all the panels, the final state would be all the panels will be hidden.

```
$panels.slideUp();
$panel.slideDown();
```

Lastly within the click handler, we remove the 'selected' class from all the tabs, and add it to the tab link that the user clicked - exactly as we did in recipe 4.2.

Finally we trigger the first tab link to be clicked which initialises the display - again, exactly as we did in recipe 4.2.

Handling accordion quirks

With this simple block of code, there are a couple of quirk that makes effect not quite perfect:

1. When the page loads, all the panels slide up, then the first one slides down. It should just appear in that state to start with.
2. If you click on a panel that's already open, it slides down and then back up again. If this happens, it shouldn't animate at all.

To fix these issues, we need to make a few changes to our jQuery. To prevent the initial loading effect, we need to hide all the panels to start off with, and then show, not `slideDown`, the first panel.

However, I don't want to write fresh code to handle the showing - because in the future there may be other tasks that run within the click handler, so I still want to trigger the click handler.

We'll still use the `$tabs.filter(...).click()` technique, but we'll disable the animations before we trigger the click, and turn them back on afterwards.

Disabling animations is a new feature in jQuery 1.3.2, and is accessed via `$.fx.off = true`.

We'll remove the chained `.filter(':first').click()` and add the following code after we assign the click handler:

```
$.fx.off = true;
$panels.hide();
$tabs.filter(':first').click();
$.fx.off = false;
```

This fixes the initial loading the page, and not it correctly starts with one panel open, and doesn't animate while loading up.

To fix the issue where clicking an open tab fires the animation again, we need to just say: if the panel is visible, don't run the animation. We'll wrap the `slideUp` and `slideDown` code with the following if statement:

```
if (!$panel.is(':visible')) {
  $panels.slideUp(200);
  $panel.slideDown(200);
}
```

Now our example opens properly, and doesn't go haywire when we click open tabs. Next we'll look at a recipe that uses the same markup, and the same code, but just keeps the panels open, so that you can toggle the accordion.

Toggle Accordions

Following on from recipe 4.7, the toggle accordion is the same fundamental UI element, but the panels stay open, and only close when you click on the header of the open panel.

Solution

```
$(document).ready(function () {
  var $tabs = $('h2 a');
  var $panels = $tabs.parent().next();
  $tabs.click(function () {
    var $panel = $(this).parent().next();
    $panel.slideToggle();
    $(this).toggleClass('selected');
  });
  $.fx.off = true;
  $panels.hide();
  $tabs.filter(':first').click();
  $.fx.off = false;
});
```

Discussion

As you can see from the code, compared to recipe 4.7, it's very similar. We still have the same `$.fx.off = true` technique to show the first tab on load.

The difference is that we don't `slideUp` all the panels, and the tab link highlighting isn't removed, but toggled.

By calling `$panel.slideToggle()`, it reads the current state of the panel, and animates it up or down. Since the default state is hidden (from the line `$panels.hide()`), the first click will slide the panel down.

Similarly, applying the 'selected' class to the link uses a toggle, and as the initial state is not to have any class, it will add the 'selected' class matching the open state of the panel.

TIP: LOTS OF CHAINING

The click handler for the toggle accordion could be rewritten in to a single line, since everything revolves around this keyword.

We could toggle the class first, then navigate the DOM to find the panel, and then toggle the slide animation, so that the code reads as:

```
$(this).toggleClass('selected').parent().next().slideToggle();
```

The problem is how it actually reads, particularly months, or even years later. Or how it reads to a new member of the team you work on. It looks like a lot is going on, and there's a lot being juggled, even though the non-chained code reads much, much simpler.

The tip is really to know when to draw the line in chaining, which we'll see again and again!

Note that the above code should be part of the callout

jQuery UI Accordion

We've already seen with the Tabs UI recipe, that if we using jQuery UI, there's a lot of functionality that comes straight out the box for us. Which means we don't have to manually code any of the functionality to create an effect, which is excellent for fast prototyping, or live wireframes.

jQuery UI also has an accordion, which just requires us to add the markup and our content in the typical accordion pattern of heading, content, heading, content, etc. Then this whole pattern is wrapped in another element, a for example, and finally trigger our jQuery UI plugin.

Obviously if we're taking the solution straight off the shelf, it means we may have more work to style the element, but certainly for prototyping, it's much, much faster.

Using the jQuery UI plugins also means we can very easily, quickly and cheaply experiment with what functionality or effects work.

One key effect that is achieved on the Apple web site, if you look carefully, is the height of the accordion is maintained exactly, even though the contents may be shorter for each panel. We could write our code for this, but the jQuery UI library offers an easy way for us to achieve this.

For our solution, we'll re-use the markup from recipe 4.7 and apply the jQuery UI accordion for an instant effect. In addition to ensuring the height is always maintained, we will also add the option to drive the active panel from the URL, thus supporting permalinks as we did in recipe 4.4.

Solution

As discussed, the markup is exactly the same as before, except that we have wrapped the element with a surrounding `<div>`.

Listing 4.10 - jQuery UI accordion plugin

```
<style>
  #accordion { height: 300px; }
</style>
<div id="accordion">
  <h2><a href="#copyright">Copyright</a></h2>
  <div id="copyright">
    <p>Our copyright notice (goto <a href="#terms">terms</a>)</p>
  </div>
  <h2><a href="#terms">Terms and conditions</a></h2>
  <div id="terms">
    <p>Here be terms and conditions</p>
  </div>
  <h2><a href="#accessibility">Accessibility statement</a></h2>
  <div id="accessibility">
    <p>The accessibility statement</p>
  </div>
</div>
```

The following jQuery initialises the plugin, and tells it to maintain a static height even if the contents of the panel doesn't fill that height:

```
$(document).ready(function () {
  $('#accordion').accordion({ fillSpace : true, navigation: true });
});
```

Discussion

The final rendered result, with only one line of jQuery can be seen in figure 4.XXX. Currently it's still using the default styles that come with jQuery UI, but regardless, as a quick way to demonstrate to your client the interactive functionality of the widget, jQuery UI makes your job very easy.

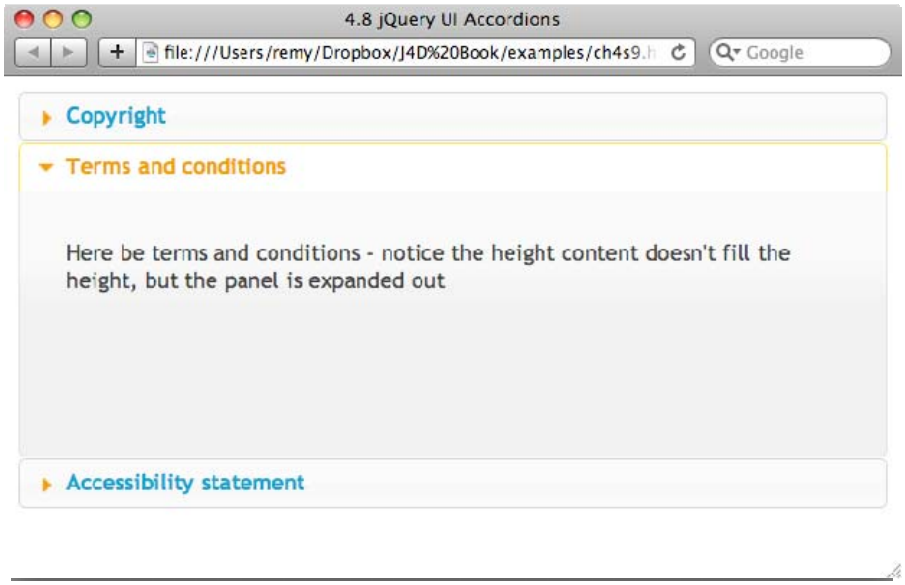


Figure 4.XXX - Example of jQuery UI accoridon plugin

The jQuery UI plugin also offers easy ways to specify the type of animation easing to be used, as well as saying that the panel should show when the mouse rolls over the heading, rather than just a click. These are both easy with very small changes to the jQuery:

```
$('#accoridon').accoridon({
  fillSpace: true,
  navigation: true,
  animation: 'bounceslide',
  event: 'mouseover'
});
```

There's lot of examples online embedded in the jQuery UI documentation located at http://jqueryui.com/demos/accordion/ that you can test out live in the browser, and experimentation is the best way to understand the full capabilities of these widget plugins.

Summary

Tabbing systems have been the theme of all of these recipes. Although we started with a relatively simple show and hide, the technique is what is fundamentally behind how tabbing systems work.

jQuery makes the job of building that tabbing system, and it's interactive components very simple to build, but also to enhance - such as the back button support and permalinks.

You now know how tabbing systems work, and will be able to recognise them and deconstruct the effect yourself.

Equally you'll recognise accordion effects, which are just like the tabbing system under their sliding exterior.

Finally, you've seen how the jQuery UI plugins is a great resource for creating the effects very, very quickly and with very little code, making the jQuery UI library an excellent resource for very fast prototyping and for live wireframing - something that could help you win the business from your potential client.

Next we'll be looking at how we can use jQuery to bring the browsers to a level playing field, either with respect to we develop with the browser, or how it behaves, be it Internet Explorer 6 or the latest browsers such as Google Chrome.

5

Upgrading the Browser with jQuery

If you're new to web development or you've been in the game for years, you'll know that toughest part, or perhaps the most interesting part of the job is getting your site to work in all the browsers.

It's not so much the issue of whether your design appears the same in each browser (which you may or may not subscribe to, but for me, this says it all: www.dowebsitesneedtolookexactlythesameineverybrowser.com), the issue is that there are features that *should* be available in all the browsers, and sometimes they're not. This may be due to bugs, or it may be that the feature or technology just isn't prolific yet. jQuery can help us fix that problem.

jQuery is very good at diving into the DOM, and making adjustments for us. For example, IE6 doesn't support PNG transparency. Using jQuery we can manipulate the DOM around the PNG files and enable transparency.

Using jQuery, we can make the browser an easier working environment for our day-to-day business.

We'll start by looking at how jQuery can plug some of the annoyances in browsers, and level out the playing field by fixing the inconsistencies that are now considered bugs.

Unfortunately IE is mostly the focus of our fixes in the first part of this chapter. We'll look at the different ways to fix the PNG transparencies, pros and cons for each. Then we'll fix the `:hover` and `:focus` pseudo selectors and finally (for IE) we will fix the way it renders overflowing text, which it does inconsistently to all other browsers.

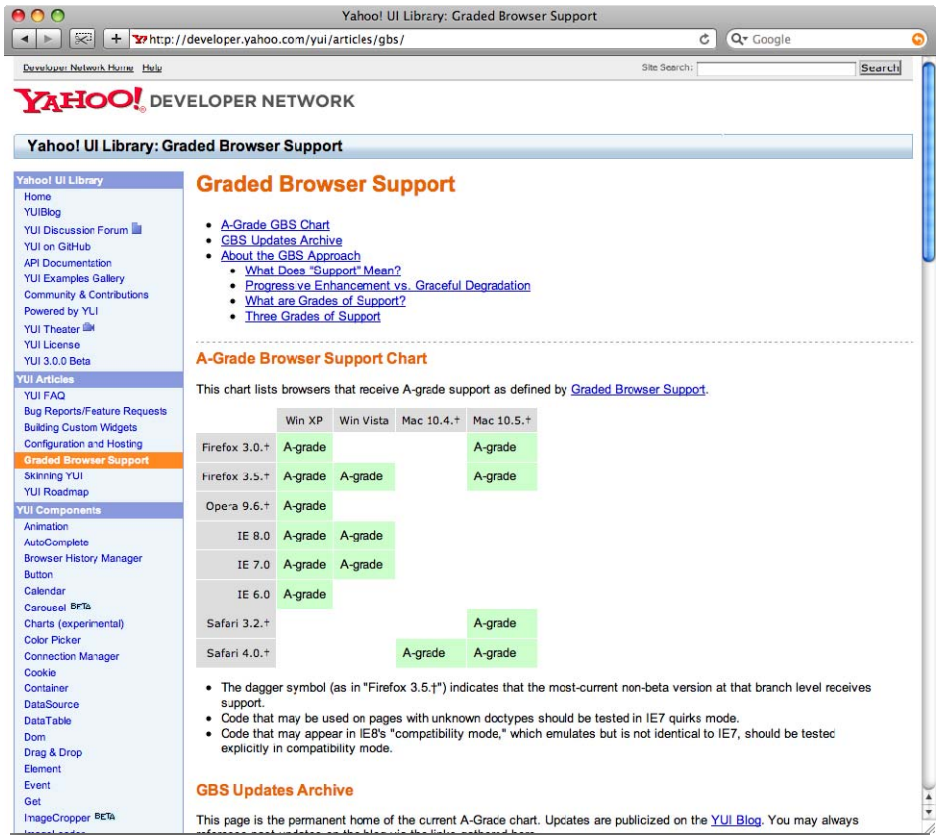
In addition, I'll show you how to make use of jQuery's superb support for CSS selectors to plug better CSS selector support in the browser, and then show you how to make anything clickable, bespoke CSS selectors and easy for validation.

In the second part of this chapter I'm going to explain how we can upgrade the browser using jQuery, and I'll show you how you can use CSS3 selectors in all the browsers, how you can make anything clickable, including areas of the page that it wouldn't even be possible to

just wrap with a link and how to add form validation to your web sites. These aren't particularly existing problems with browsers that we're fixing, but they're features that have been missing from browsers for the last 5 or 10 years and often web developers have had to roll their own solutions to solve these issues. Once again, jQuery takes that headache away from you and makes what might seem like a complicated job, quite easy.

A-grade browsers

Yahoo! provides a table of "A-grade" browsers. These are browsers that still have a majority share in the active use. You may not need to support all of these browsers, but it's good starting place to direct your clients to when agreeing on which browsers to support.



http://developer.yahoo.com/yui/articles/gbs/

Fixing the broken browsers

In this section as we've mentioned before, unfortunately, we are almost entirely dedicated to fixing issues with IE6 and IE7. However, once you've got these issues, like PNG transparency, fixing the :hover pseudo selector and the other recipes we look at, you'll be able to include the solutions in your web sites and never have to burden your day again with how to fix these mundane issues that IE6 & IE7 keep throwing at you.

We will be looking at the PNG transparency issue in IE, and comparing the different solutions available and looking at the pros and cons.

Then I'll show you how IE6's version of overflowed text is completely inconsistent with all other browsers, and how we can correct this.

We'll also give the :hover and :focus pseudo selectors to IE6, since that browser only supports the selectors on links and no other elements.

Finally I'll show you how to exploit jQuery's CSS selector support to shoehorn in some of the very useful CSS3 selectors.

PNG transparency

PNG transparency is a long standing issue with IE6 and previous incarnations of Internet Explorer. The fix is to typically include a small JavaScript library to deal with PNGs for us.

This solution is going to be less of a recipe, but more to explain why you might choose which PNG transparency fix, as each may have their different issues.

Ultimately we want to end up with some code that gives a relatively easy win to fixing the PNG transparency issue.

For reference, Safari is rendering our example page with correct transparency, as seen in figure 5.1.

Figure 5.1 - Two reference examples of PNG transparency, the first using an image with a shadow, the second with a background image with an transparent gradient

There are three PNG fixing methods I'm aware of:

1. The HTC PNG fix
2. AlphaImageLoader
3. VML

Let's look at the three methods we mentioned above and weigh up the pros and cons.

OLD WAY: HTC PNG FIX

It sounds like a lot of gobbly-gook and to be honest, it's some clever Microsoft specific code that fixes foreground PNG images using proprietary HTC file as behaviours in CSS.

The only benefit of this fix is that you just include it in your CSS and it works for you.

I'm not going to show you the code, or explain how it works, however, I will point out two very big problems with this approach:

1. It works on foreground images, and not background images (via CSS)

2. For every image you have, IE6 (which isn't a fast browser to start with) will fire off a new request for the fixpng.htc file, which could lead to hundreds of inadvertent web requests

These two reasons are so important that I implore you **not** to use this method. The negatives far outweigh the benefit of simplicity of the fix. The following two methods are much faster and cleaner for your web site.

OLD RELIABLE: ALPHAIMAGELOADER

This is just the name given to the IE proprietary CSS filter that gives the browser alpha transparency (for IE versions 5.5 and and above).

There are some related issues that come up when you use the AlphaImageLoader, specifically: if you use the filter on an element that contains an a element or input element then those elements become unclickable! Fear not though, it's easy to fix by adding position: relative to those unclickable elements, and they become available again.

One of the best jQuery plugin amongst many available is Drew McLellan's Supersleight PNG fix. It handles foreground images, background transparency and the unclickable elements all within the single plugin available here:

http://allinthehead.com/retro/338/supersleight-jquery-plugin

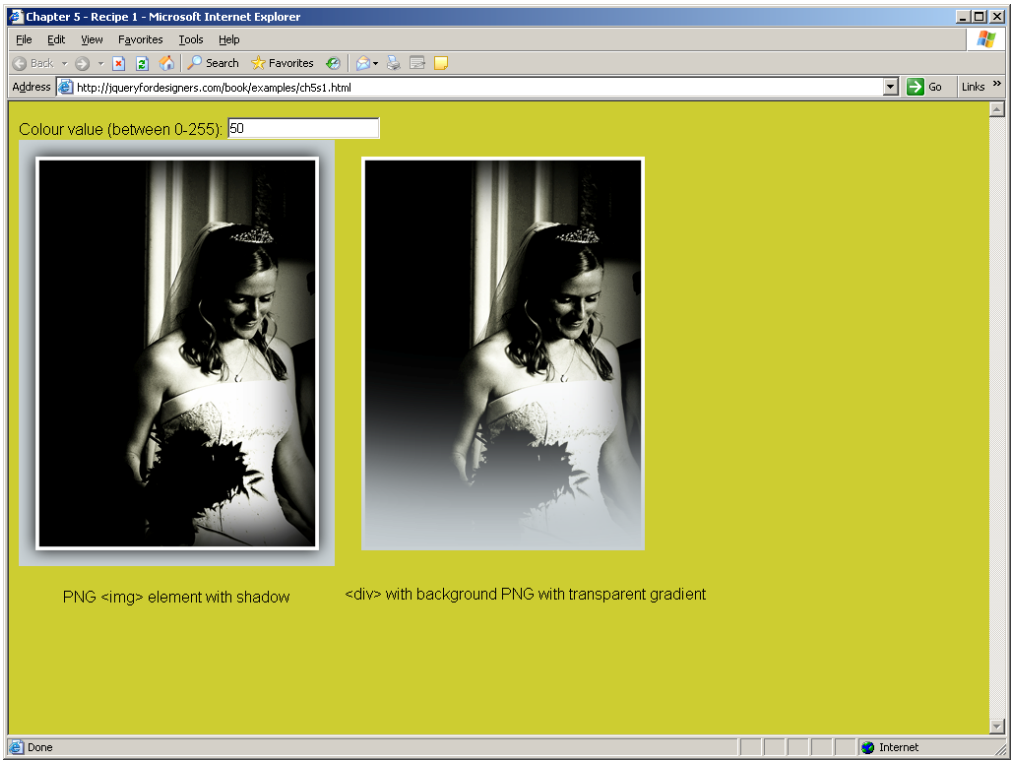
By default the jQuery plugin fixes all the PNG issues, unless you specifically choose to disable them. The only dependency is that you need to host a 1x1 transparent gif (which all of the AlphaImageLoader solutions require), and we can override the defaults and tell the plugin where this is located:

Listing 5.1 - PNG images and backgrounds

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset=utf-8 />
  <title>Chapter 5 - Recipe 1</title>
  <style type="text/css" media="screen">
  body {
    background: rgb(205,205,50);
  }
  #gradient {
    background: url(images/julie-with-grad.png) no-repeat center;
    width: 321px;
    height: 433px;
  }
  </style>
  <script src="jquery-latest.js"></script>
  <script src="supersleight.plugin.js"></script>
  <script>
  $(document).ready(function () {
    $('body').supersleight({ shim : 'images/blank.gif' });
  });
  </script>
</head>
<body>
```

```
    <img src="images/julie-with-shadow.png" />
    <div id="gradient"></div>
  </body>
  </html>
```

The only issue with the AlphaImageLoader approach is that CSS background offsets don't work. In this contrived example, the that shows the image with a gradient is slightly oversized, in that it's bigger than the actual background image - which is why I've centre aligned the image in the CSS as seen in figure 5.2.



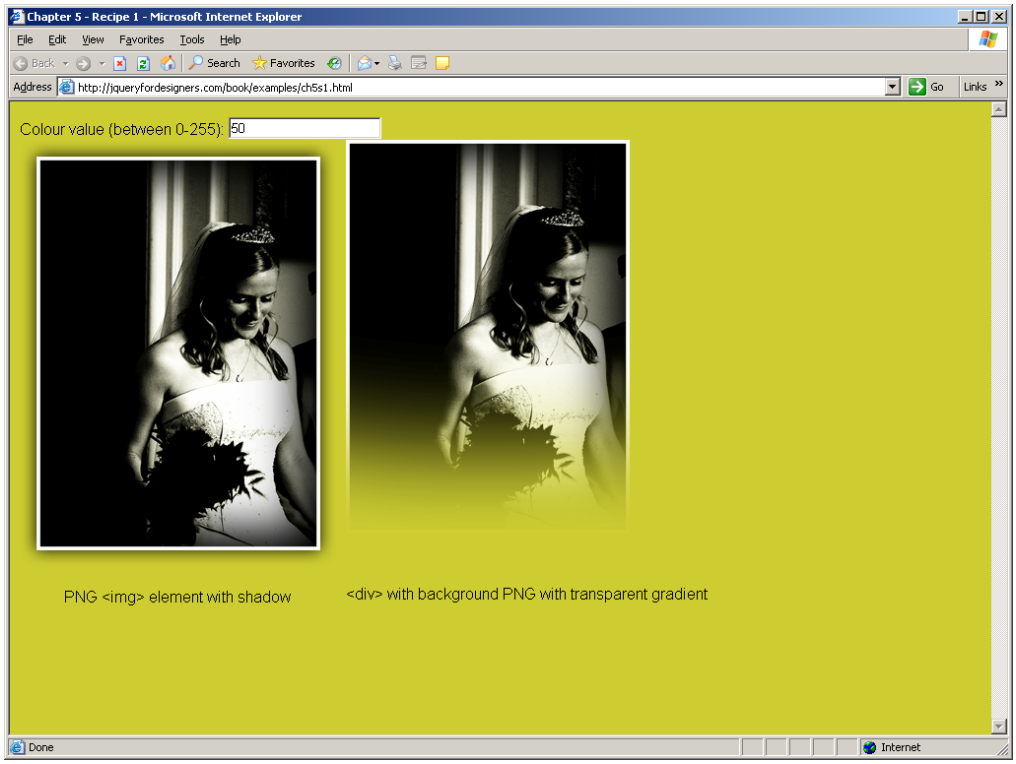


Figure 5.2 - The AlphaImageLoader PNG fix applied, but you can see the gradient image on the right is slightly misaligned

The VML solution fixes this problem, which we'll see next.

THE NEW KID ON THE BLOCK: VML

A new and completely different approach was developed recently by Drew Diller, using Microsoft's implementation of VML (a vector markup language) called: DD belated PNG fix:

http://www.dillerdesign.com/experiment/DD_belatedPNG/

This script has some limitations, in particular you can't apply it to the whole `<body>` tag like you can with the solution above, nor can if fix PNG backgrounds on `<tr>` elements.

However, since it's using VML it has some extra tricks up it's sleeve over the AlphaImageLoader approach: in particular, it can position the background transparent PNG correctly, as seen in figure 5.3.

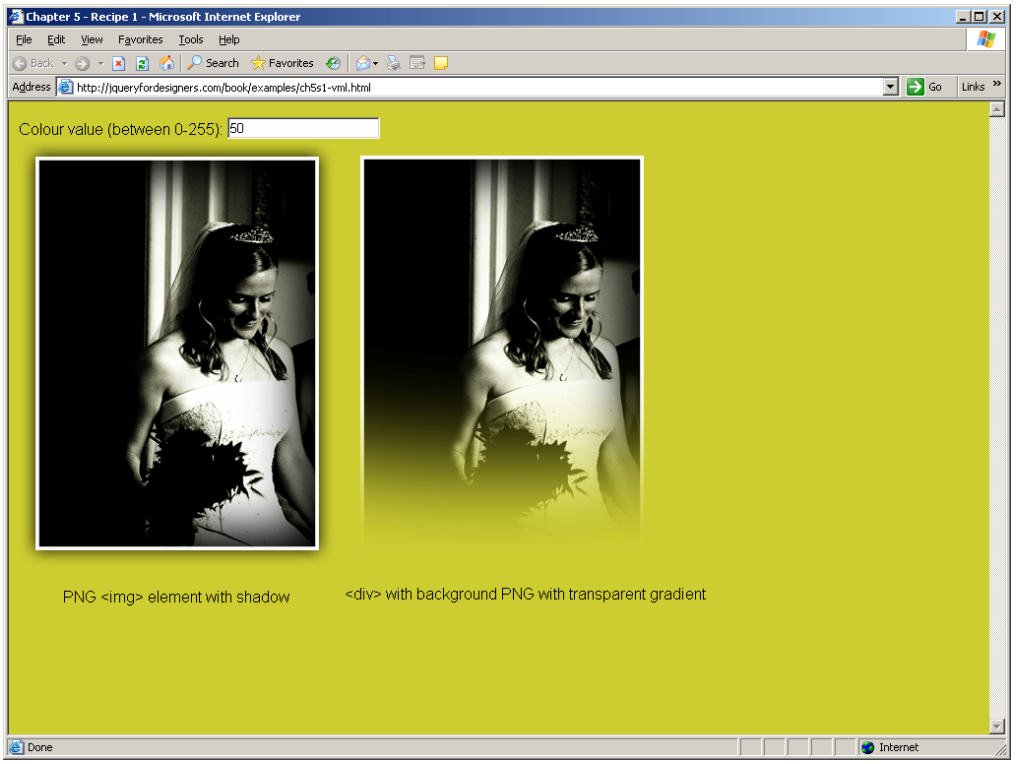


Figure 5.3 - The DD belated PNG fix with correct background positioning

Using the same markup from listing 5.1, we'll change the JavaScript include to use the DD belated library and trigger the plugin once the document is loaded:

```
<script src="jquery-latest.js"></script>
<script src="DD_belatedPNG_0.0.8a.js"></script>
<script>
$(document).ready(function () {
  DD_belatedPNG.fix('img, #gradient');
});
</script>
```

Notice that the syntax isn't particularly jQuery-esque, but that's okay. We're calling the DD belated PNG fix once the document has loaded, and we're applying it to all the `<img>` elements and the element with the ID `#gradient`.

I'd recommend looking at your web site, project by project, and deciding which out of the AlphaImageLoader and VML solution to use, since there are pros and cons to each.

Supporting :hover & :focus on anything

The `:hover` and `:focus` pseudo CSS behaviours are useful for more than just highlighting links. We can also use the `:hover` selector to reveal additional information just using CSS. However, only IE7 and above supports `:hover` and `:focus` selectors any element as outlined in the CSS 2.1 recommendations.

This is easy to fix using jQuery. We can use conditional comments to target IE6 and versions below, and using jQuery's hover method we can replicate the `:hover` selector, and separately using blur and focus we can replicate the `:focus` selector:

Listing 5.2 - Enabling :hover and :focus selectors for IE6

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset=utf-8 />
<title>Chapter 5 - Recipe 2</title>
<style type="text/css" media="screen">
body {
  background: #D8CF9D;
  margin: 20px;
}
#gradient {
  background: url(../images/chapter5/julie-with-grad.png) no-repeat center;
  width: 288px;
  height: 400px;
}
#gradient div.info {
  opacity: 0;
  filter:alpha(opacity=0);
  color: #eee;
  margin: 4px;
  padding: 20px;
  background: #000;
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
}
#gradient:focus,
#gradient.focus { /* duplicated for IE6 support */
  border: 4px solid #C5BD85;
}
#gradient:hover div.info,
#gradient:focus div.info,
#gradient.hover div.info, /* duplicated for IE6 support */
#gradient.focus div.info {
  opacity: 0.7;
  filter:alpha(opacity=70);
}
#gradient div.info p {
  font-family: georgia, times;
  font-style: italic;
  text-align: center;
```

```
}
</style>
<script type="text/javascript" src="jquery-latest.js"></script>
<!--[if lte IE 6]>
<script src="DD_belatedPNG_0.0.8a.js" type="text/javascript"></script>
<script type="text/javascript">
 $(document).ready(function () {
   DD_belatedPNG.fix('#gradient');
   $('#gradient').hover(function () {
     $(this).addClass('hover');
   }, function () {
     $(this).removeClass('hover');
   }).focus(function () {
     $(this).addClass('focus');
   }).blur(function () {
     $(this).removeClass('focus');
   });
 });
</script>
<![endif]-->
</head>
<body>
  <div class="box">
    <div tabIndex="0" id="gradient">
      <div class="info">
        <p>June 19th, 2004: My wife to be.</p>
      </div>
    </div>
  </div>
</body>
</html>
```

Aside from the jQuery code in the conditional comment, the only extra overhead is to duplicate the hover and focus rules with straight CSS classes where we've duplicated the `#gradient:hover div.info` with `#gradient.hover div.info`.

However, we're already used to this kind of extra support for IE specific rendering as you can see where we use the opacity, we need to specifically target IE using the alpha filter - so I'd argue this is justifiable:

```
#gradient.focus div.info {
  opacity: 0.7;
  filter:alpha(opacity=70);
}
```

As you can see in figure 5.4 the hover and focus effect works in IE6, and since we've targeted the browsers that don't support these pseudo selectors, the default browser effects kick-in in all the other browsers like Firefox, Opera and Safari.

Also note that by adding the `tabIndex="0"` attribute to the `<div>` element, it can be tabbed to using the keyboard triggering the focus event.

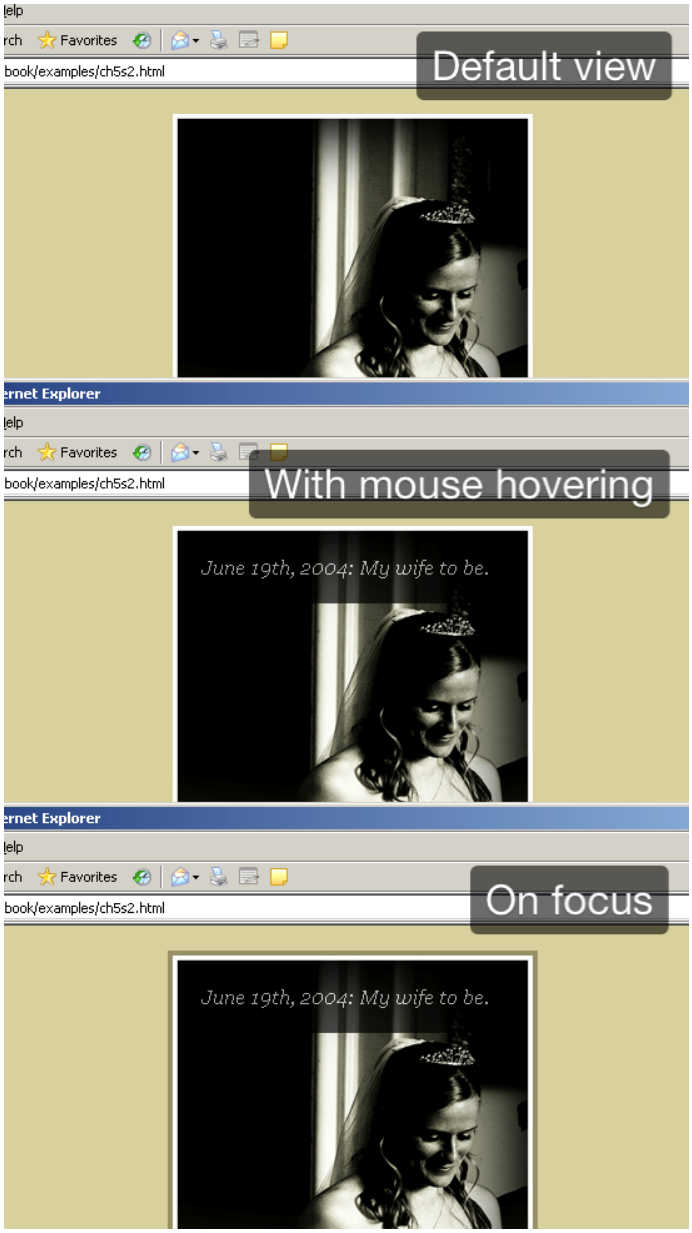


Figure 5.4 - IE6 with the :hover and :focus selectors fixed

Fixing overflow heights

The overflow is a CSS property that hides content that flows outside the bounds of the element. This is often used in blog posts showing code snippets where you don't want the text to break the layout of your site.

We use `overflow: auto` to say if the content is too big for it's container, show scrollbars to let the user view the oversized content.

This is typically fine in most browsers, and if you're overflowing text, then it will flow on the horizontal axis.

However, in IE7 and below, the overflow is rendered slightly differently to other browsers (though this is fixed in IE8 now).

The issues is the scrollbars in IE7 and below are placed inside the overflowing element, where as the vertical scrollbar should be on the outside of the overflowing element as seen in figure 5.5:

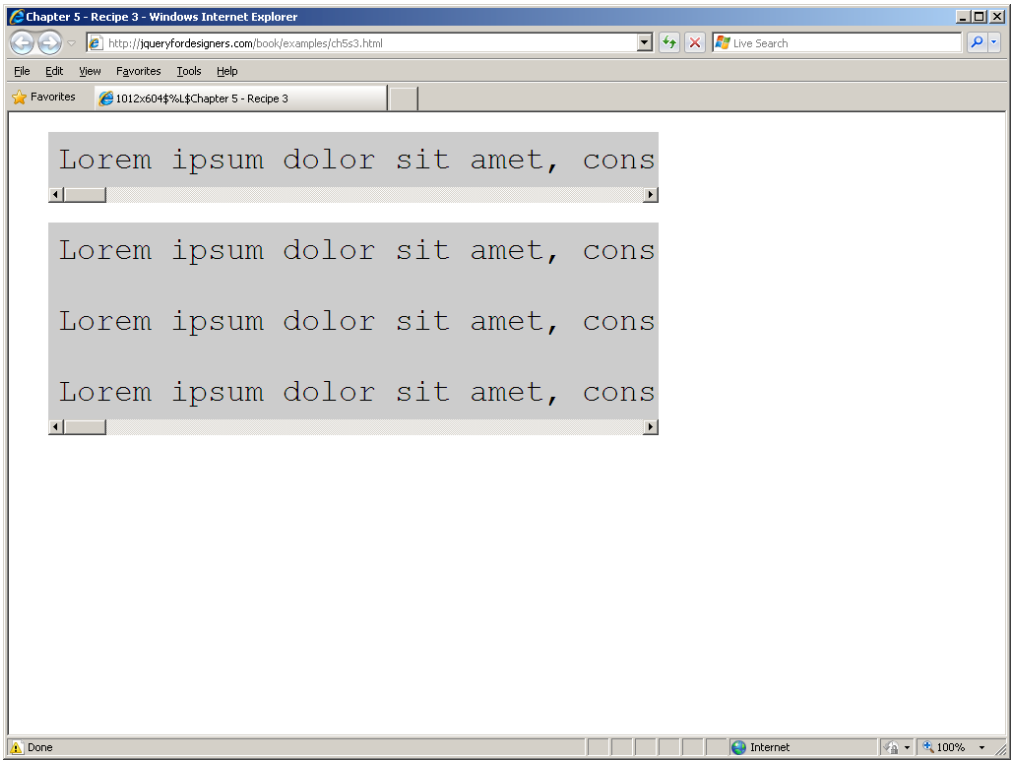


Figure 5.5 - IE8's overflow rendered correctly and used for reference

When IE7 and below render the scrollbars inside the element, it causes the element to automatically overflow vertically as well, which results in the vertical scrollbar appearing - which just looks ugly (as seen in figure 5.6):

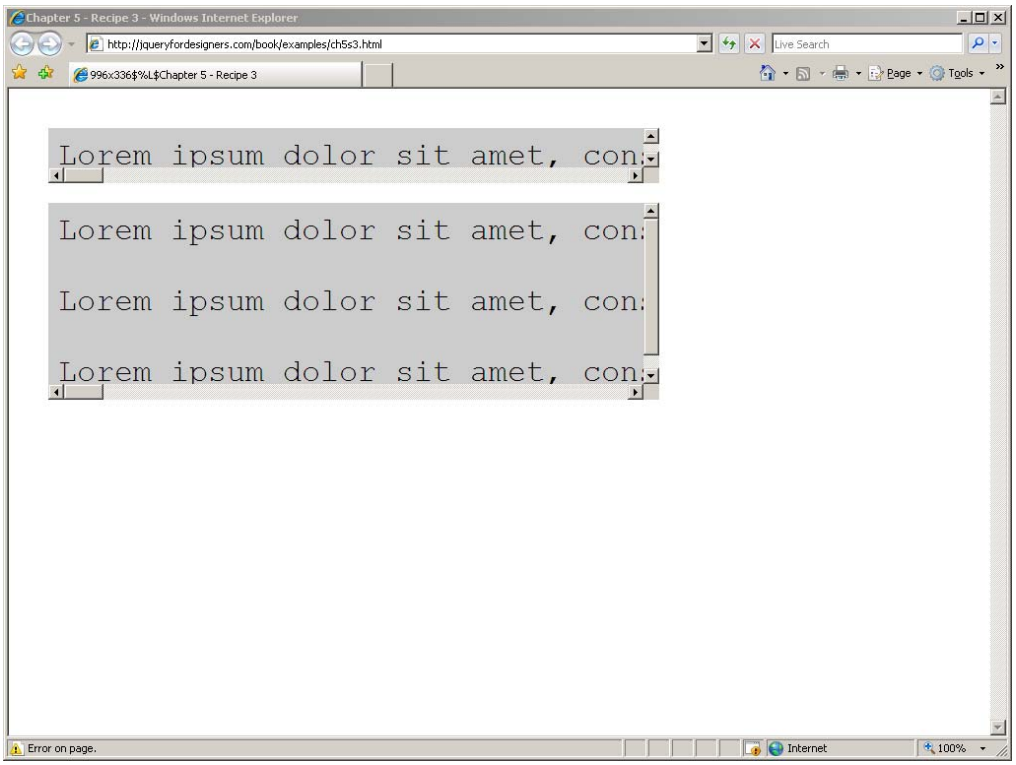


Figure 5.6 - IE7's overflow incorrectly placing the scrollbars inside the element

The solution is to detect when these element are overflowing, and fix them by manually pushing the padding at the bottom of the overflowing element, which will allow room for the horizontal scrollbar, which means the vertical scrollbar *won't* trigger.

SOLUTION

Remember that this issue only affects IE7 and below, so we're going to create and use a plugin targeted at those browsers only. Because the plugin is inside the conditional comment, it **won't** be available to other browsers.

Listing 5.3 - Fixing overflow for IE7 and below

```
<!--[if lte IE 7]>
```

```
<script type="text/javascript">
$.fn.fixoverflow = function () {
  return this.each(function () {
    if (this.scrollWidth > this.offsetWidth) {
      $(this).css({
        'padding-bottom' : '20px',
        'overflow-y' : 'hidden'
      });
    }
  });
}
$(document).ready(function () {
  $('div').fixoverflow();
});
</script>
<![endif]-->
```

DISCUSSION

The key is being able to determine whether the element is actually overflowing or not, and within our plugin this is achieved using the following:

```
if (this.scrollWidth < this.offsetWidth)
```

Remembering that this is a keyword and the reference to the current element in the loop (current based on the plugin's use, whatever that may be).

By asserting whether the scroll width (`scrollWidth`) is actually wider than the actual visible width (`offsetWidth`), we know we need to fix the overflow.

Next we just adjust the padding on the element to compensate for the newly introduced horizontal scrollbar, and for good measure, we hide the vertical overflow.

Adding 20 pixels to the padding is a slightly arbitrary number, but it is based on looking at how IE renders the scrollbars and taking the best approximation of the height of the scrollbar.

The result is correctly rendered overflow scrollbars in IE7 and below that match that latest browser IE8, as seen in figure 5.7:

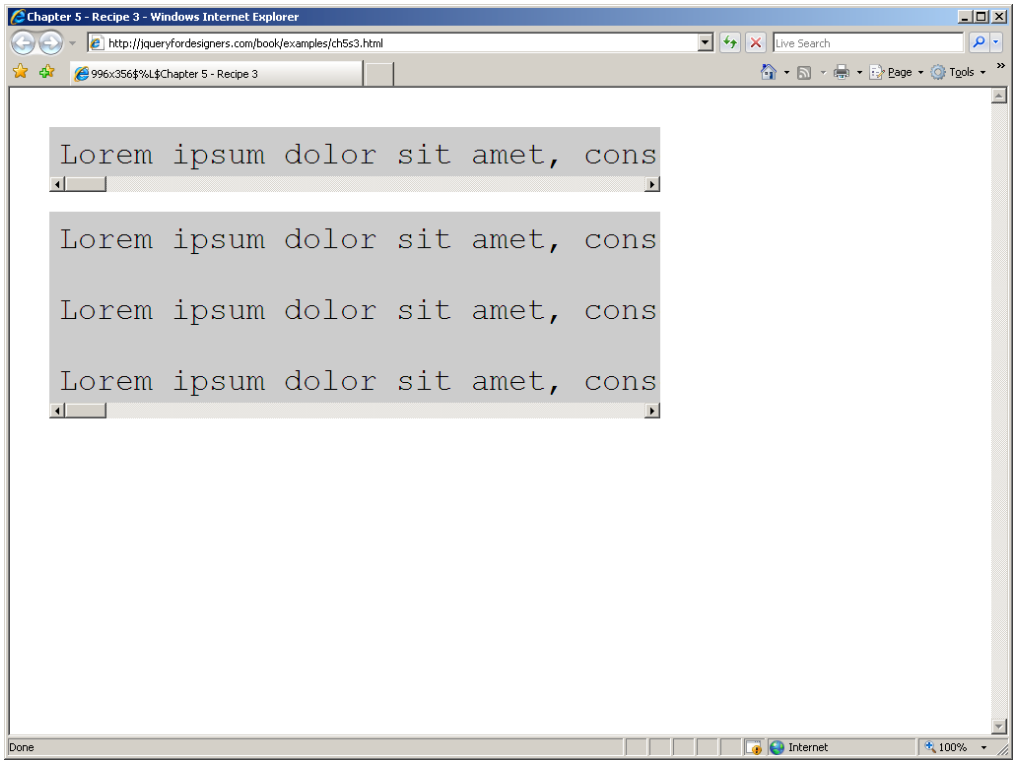


Figure 5.7 - IE7's overflow correctly rendering

Adding CSS2 & CSS3 selector support

CSS2.1 has been around for a good while now, over 10 years, though the first 100% complete implementation has only recently arrived in IE8, we're already using a lot of the features of CSS.

However, CSS3 is relatively new, and particular the Internet Explorer chain of browsers don't have any support for CSS3.

We can use jQuery to patch the support for missing CSS selectors, just as we did for the `:hover` and `:focus` pseudo selectors in recipe 5.2.

I would argue that you probably shouldn't be looking to patch simple selectors such as the child selector (E > F) as good markup and CSS should solve that problem.

However, jQuery is very powerful for filling in some of the more complicated selectors that aren't available in all the A-grade browsers, such as:

- n-th child selectors, e.g. E:nth-child(n)

- Attribute selectors, where attribute ends with or starts with, e.g. E[foo^="bar"]
- UI state selectors: E:checked
- Content selectors: E:contains("foo")

In this recipe I'll show you how easy it is to apply CSS selectors that aren't available in your browser.

I have a number of list elements and I want to wrap them so that each row contains 4 elements, and each column is a different colour.

There's a number of different ways to achieve this, but I want to use the following style of CSS:

```
li:nth-child(4n+1) {
  clear: left;
  background: #c00;
}
li:nth-child(4n+2) {
  background: #0c0;
}
li:nth-child(4n+3) {
  background: #00c;
}
li:nth-child(4n+4) {
  background: #c0c;
}
```

However, since this doesn't work in any of the Internet Explorer versions, nor Firefox 3 as seen in figure 5.8, so we're going to simplify the CSS, and use jQuery to target the elements to style them.

CSS3 nth-child

The CSS selector `li:nth-child(4n+3)` is the equivalent of saying: select all list elements, starting from the 3rd item, and every fourth item there after.

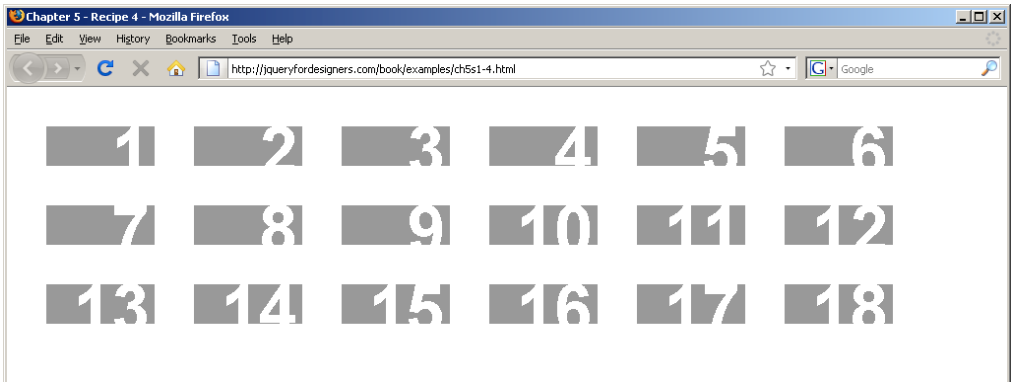


Figure 5.8 - Firefox 3 using nth-child selectors failing

SOLUTION

We've modified and simplified the CSS from using the nth-child selector, to using straight classes which we'll apply using jQuery:

```
li.nplus1 {
  clear: left;
  background: #c00;
}
li.nplus2 {
  background: #0c0;
}
li.nplus3 {
  background: #00c;
}
li.nplus4 {
  background: #c0c;
}
```

Now using the same nth-child CSS syntax, we'll select the elements using jQuery and CSS3 and apply the appropriate classes:

```
$(document).ready(function () {
  $('li:nth-child(4n+1)').addClass('nplus1');
  $('li:nth-child(4n+2)').addClass('nplus2');
  $('li:nth-child(4n+3)').addClass('nplus3');
  $('li:nth-child(4n+4)').addClass('nplus4');
});
```

DISCUSSION

Since jQuery has a near complete CSS3 selector engine (with the exception of a few CSS3 selectors such as `:target`), we can reuse some of the more complicated selectors using jQuery where the browser support isn't very good.

The result using jQuery is the same as if we were using a browser with the latest support for CSS3 selectors, as seen in figure 5.9 in Firefox 3 (which doesn't natively support nth-child).

Figure 5.9 - Firefox 3 using nth-child selectors working thanks to jQuery

In the next section, we'll be looking at how to add missing functionality that isn't available in the browser starting by making absolutely *anything* clickable!

Adding missing functionality to the browser

As we'll see through this book, jQuery is very easy to extend through plugins. This means there's a lot of plugins available on the web to solve lots, and lots of different problems.

However, by missing functionality, I mean functionality you would expect from a browser rather than just anything that enhances the user experience. The kind of functionality you would expect to be available since you require it many of the projects you work on.

There's obviously a very thin line between what we expect from a browser, what's a bug, and what's too much. The last recipe, adding CSS3 support, could have well been in this section. This only shows that some of the recipes in this section aren't currently things we're trying to fix, but in the next 5 to 10 years, could well end up being issues we consider bugs if they're not supported.

Two out of the following three recipes are part of existing specifications (even if they've not reached "recommendation" stage yet): anything clickable and validation. I would hope that in 5 years time, after this book is published, these two will be supported natively in the browser. If they're not, then they're "bugs" that we'll want to fix.

Making anything clickable

With HTML 4 (and XHTML), in a valid document, you can't put block layout elements inside an anchor element. For example you can't wrap a link around a heading, an image and a paragraph as it won't validate.

In HTML5, this is valid code, but there's still cases where this doesn't work, for example making clickable table cells.

Having block level clickable objects is very useful from two points of view:

1. You don't have to write repetitive code, i.e. maintaining lots links in the block which could lead to mistakes
2. It reduces the markup required to do the job, which is always good for bandwidth for your site, but again makes the code easier to maintain

We're going to look at how we can use jQuery to make anything clickable, either a group of elements or even a table cell where you can't physically wrap a link around.

There's two ways to approach this problem - the first is to use jQuery to create a clickable area without having a physical link. I don't like this approach, because it means that without JavaScript enabled, there's no link for the user to click. It also means that search engines can't spider your content since they don't run with JavaScript.

The second approach, the one we'll use, is to find the link inside our group of elements, and use it as the url we'll take the user to when they click.

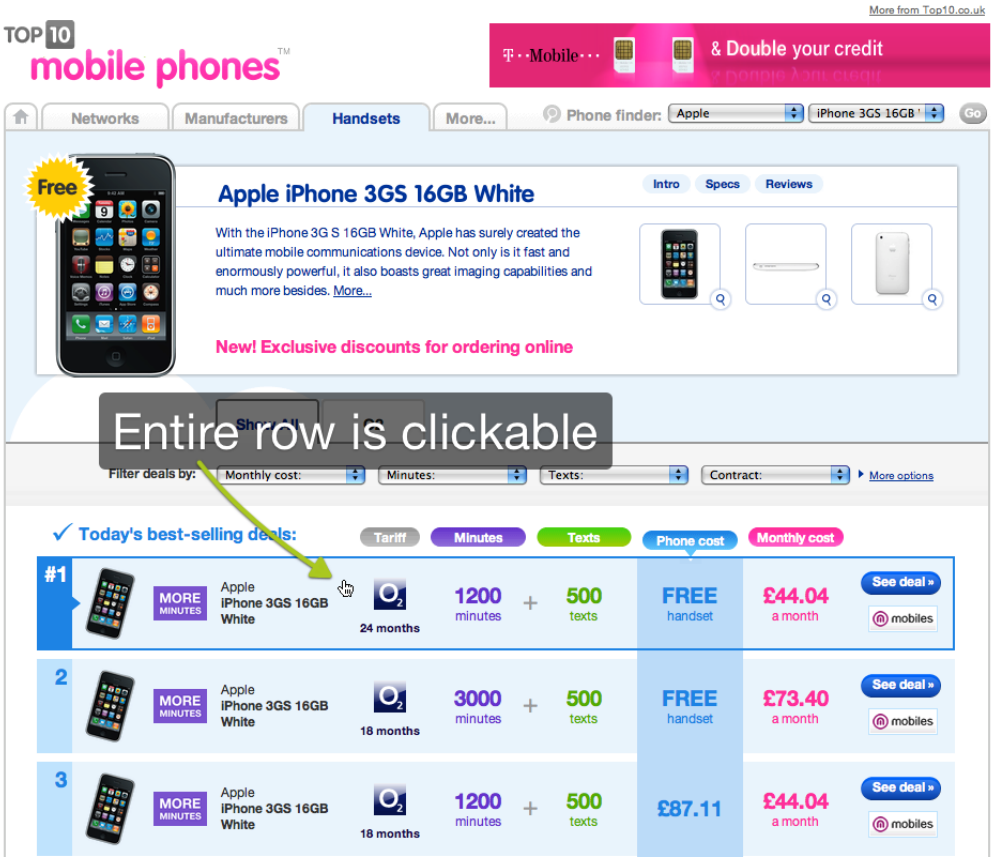


Figure 5.10 - Example of an entire row being clickable

In figure 5.10 the user is presented with a number of rows in a table. There's only one real anchor element, but the entire row is clickable. We're going to use jQuery to find the rows, then from within each row, find the link that we're going to take the URL from, and use this for the click event on the row.

In addition to being clickable, we also need to provide a visual cue to the user the element can be clicked on. For our example, we're going to use the pointer cursor as the user hovers over - but this shouldn't be turned on by default, again because if JavaScript is disabled, the row isn't clickable. We simply solve this by adding a class with cursor: pointer to the row once we've attached the click event.

SOLUTION

```
$('tr').each(function () {
  var $row = $(this);
  var url = $row.find('a').attr('href');
```

```
    $row.click(function () {
      window.location = url;
    }).addClass('clickable');
  });
```

DISCUSSION

Our solution runs through each row element, it finds the link and stores the url and uses this to redirect the browser to if the row is clicked on. Finally, it adds the class 'clickable', which I'm saying includes the CSS style of cursor: pointer.

The first line of our code captures a jQueryified version of the row element:

```
var $row = $(this);
```

We could have used `$(this)` in the code, but if you find you need to jQueryify this more than once, it's good practise to store it in a variable, as I did in the first line.

Next, using the row element as the context of our search, we look for the link inside our element, and capture the href:

```
$row.find('a').attr('href')
```

If you have more than one link in the row, you can use a class selector to narrow down the search (though the attr will return the value from the first element). If you do have more than one link in the row element, you should only apply this technique if all the links have the same url.

If your markup does have different links inside the row element, it's recommended that you don't use this technique. In fact, you shouldn't make the entire row clickable at all. If you did, the user would only be able to access the one link that you make available, and rather than enhancing the browser, you would actually be blocking a natural and useful feature of the browser.

Finally, we bind the click handler to the entire row, saying that if the row is clicked, we change the location of the browser to the url found earlier, and then chain the `addClass` to give the visual cue of a clickable area in the document.

Validation

Form validation has been around on the web for a long time. In fact, it was one of the most common uses of JavaScript in it's earliest years. The problem is that doing form validation with JavaScript hasn't gotten any easier over the years.

The Opera browser supports the Web Forms 2.0 specification, which allows you to include validation rules in the markup, such as defining that a text field should be a type of email or url, or saying that a specific field is required, etc.

For the rest of the browsers, currently, they don't support these validation features, and to add them laboriously each time to each project can be a pain.

Jörn Zaefferer, a well respected developer within the jQuery community, has written a jQuery plugin that solves the validation problem with very little effect on your side.

The jQuery validator plugin is available here: http://bassistance.de/jquery-plugins/jquery-plugin-validation/

Using the validator plugin, we're going to add validation rules to a registration form with very little extra work beyond the markup.

The validation plugin offers much more functionality than just vanilla validation. We can create bespoke rules, and we can perform Ajax request to help validate, such as checking whether a username is available to the user.

In our contrived example, we'll demonstrate how to validate email, and URL, a field with a minimum length of 2 characters, a username validated against the server and a field that must match "Remy" or "Julie" (but this will be case insensitive).

SOLUTION

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset=utf-8>
<title>Chapter 5 - Recipe 5.2.4 - Validation</title>
<style>
body { font-family: helvetica, arial; }
fieldset { padding: 10px; border: 2px solid #ccc; width: 600px; margin: 0
auto; }
legend { color: #ccc; }
fieldset div { clear: left; padding: 10px 0; }
label { float: left; display: block; width: 300px; }
label.error { font-size: 90%; color: #c00; margin-left: 50%; }
em { margin-left: 10px; }
</style>
<script src="jquery-latest.js"></script>
<script src="jquery.validate.js"></script>
<script>
$.validator.addMethod('who', function(value) {
  value = value.toLowerCase();
  return value == 'remy' || value == 'julie';
}, 'Please enter "Remy" or "Julie"');
$(document).ready(function () {
  $('form').validate({
    rules: {
      username: {
        remote: "username.php"
      },
      who: "who"
    },
    messages: {
      username: {
        required: "Select a username",
        remote: $.format("{0} is already in use")
      }
    }
  });
});
</script>
</head>
<body>
  <form method="post" action="">
    <fieldset>
      <legend>Register for our service</legend>
```

```
      <div>
        <label for="fullname">Name (required, at least 2
characters)</label>
        <input id="fullname" name="fullname" class="required" minlength="2"
/>
      </div>
      <div>
        <label for="email">Email (required)</label>
        <input id="email" name="email" class="required email" />
      </div>
      <div>
        <label for="who">Remy or Julie?</label>
        <input id="who" name="who" class="required" />
      </div>
      <div>
        <label for="username">Username</label>
        <input id="username" name="username" class="required" minlength="2"
/>
      </div>
      <input class="submit" type="submit" value="Submit"/>
    </fieldset>
  </form>
</body>
</html>
```

DISCUSSION

Our form is doing quite a bit here, but for the very basics, just to say a field is required, or that it should take an email format (though the validator plugin supports many formats), it's very easy.

Let's for a moment ignore the bespoke rules I've created and just focus on the 'name' and 'email' field. All that's required is:

1. Include the validate plugin after we include jQuery
2. Give the field the appropriate class names
3. Call the validate plugin against the form we're working with

To indicate to the plugin that a field is required, we simply add:

```
class="required"
```

Then if it's an email address, or URL or some other format that's supported by the plugin (you can see the complete list from the documentation, you can include this in the class also:

```
class="required email"
```

Finally we just need to tell the validate plugin to look at our form - remember that in the full solution I've gone further, but just to validate simple details as we've already described, all we need is:

```
$('form').validate();
```

That's it. It's extremely simple to create entry level client side validation, and it's only required one line of jQuery and a few semantic classes on the input elements.

To validate the length of the name and username fields, I included the `minlength` attribute. This isn't a valid HTML attribute, but serves to inform the plugin about what the

minimum length or maximum length of the field should be. Now let's look at how we solved some bespoke validation problems.

I've created a bespoke validation rule that only allows the value of "remy" or "julie". Any other value, and the form won't be valid.

To achieve this, we add a new method to the validator, giving it a name and a function, which receives the value we want to validate and finally the text that appears if the value is incorrect.

The function in our bespoke validation method should return true to indicate that the value the user has inputted is okay, or false otherwise:

```
$.validator.addMethod('who', function(value) {
  value = value.toLowerCase();
  return value == 'remy' || value == 'julie';
}, 'Please enter "Remy" or "Julie"');
```

I've called the validation method "who", and when the value is passed in, I convert it to lowercase, just so I can avoid having to testing for different versions of "Remy" or "remy" (and so on).

Then we simply test if the value is "remy" or, indicated by the two pipes, the value is "julie". That test, or evaluation, is either true or false, and returned to the validator plugin.

We can assign the new validation method, by putting it in the class name as we did for the email validation. However, just to show you there's more than one way to skin a cat, I've included it in the rules setting.

Both the rules and messages are settings objects whose key is the id of the element, and the value is either the rule name, such as "email" or for our new method "who". Alternatively, the value can be an object that points the validator to perform an Ajax request to validate the field. That's all we need to create our own bespoke validation method.

However, if we want to perform some validation on the server side, such as checking whether a username is available or not, we need to use the remote key and then set the value to our server script that will handle the validation for us:

```
$('form').validate({
  rules: {
    username: {
      remote: "username.php"
    }
  }...
```

The PHP script "username.php" will be sent the value from the element with the id of username, and the PHP script will tells us whether the name is available or not.

I'm going to show you the username.php script. It's a very simplistic script for this example, and no doubt that the real world version that you might work on will be both much more complicated, but also managed and maintained by a server side developer. You just need to tell that developer what the script should print: the script should print either: "true" or "false". "true" means the field is valid, and "false" means it's invalid, and that's it.

The PHP script I've written checks whether the name is in a predefined list of names, and if it is, the server side script prints "false". This is all handled within the validator plugin, so

you don't have to worry about performing the Ajax request yourself, or processing the response:

```
<?php
$taken_usernames = array('remy', 'julie', 'jeremy', 'josh', 'andy',
'andrew', 'dave');
$username = strtolower($_GET['username']);
if (in_array($username, $taken_usernames)) {
  echo 'false'; // taken
} else {
  echo 'true'; // available
}
?>
```

Finally, all we need to worry about is how we format the errors. As mentioned before, the messages object also uses the id of the element as the key and the value is an object with the validation method against the error string.

In our example, we're using the "required" validation rule, so this is the key, and if the value isn't entered, the error message is "Select a username":

```
$('form').validate({
  rules: {
    username: {
      remote: "username.php"
    }
  },
  messages: {
    username: {
      required: "Select a username",
      ...
```

Now for the error message if the username is in use we want to include the actual string the user entered. We could have a static non-changing string along the lines of "That username is taken", but by including the username they entered, we're showing the user that we've not made any mistakes and we definitely just checked the username that they just entered.

To achieve this, we are using the `$.format` which is included as part of the validate plugin (i.e. this isn't a core part of the jQuery library). We can use `{0}` to represent the value that the user specified, and together with `$.format` we can create a dynamic error message that shows the value they actually entered:

```
$('form').validate({
  rules: {
    username: {
      remote: "username.php"
    }
  },
  messages: {
    username: {
      required: "Select a username",
      remote: $.format("{0} has already been taken")
    }
  }
});
```

Remember that you can combine both the defined rules and messages via the settings object and you can define the validation rules via the class names on the actual markup,

which brings it much closer to the HTML 5 Web Forms 2.0 approach of including simple form validation inside the markup.

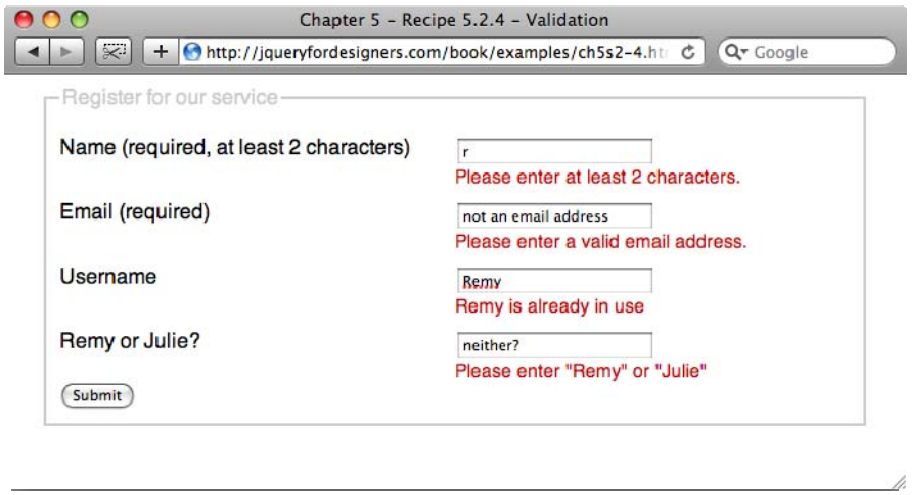


Figure 5.11 - our example failing all the validation checks

When your site visitor hits that submit button for your form, the validator plugin will run through all the fields in the form, checking it against the validation rules, and if any of the checks fail, it won't submit the form, but also shows the errors. In addition, the errors will appear when the field loses focus, helping the user spot errors as they go along, as seen in figure 5.11.

Finally I want to show you how you can extend jQuery to create you own selector, like $(':justLinks'). Although this isn't something that will become native to the browser, it's a useful feature of jQuery to know and understand if you find you need to navigate the DOM in a particular way.

Custom selectors

Just as creating custom plugins for jQuery is easy, creating your own selectors is easy in jQuery too.

The following code is a simplified method to creating your own custom selectors:

```
jQuery.expr[ ":" ].justLinks = function (el) {
  return el.nodeName == 'A';
};
```

Obviously the above example is contrived, since it's called `justLinks` and we could select these using $('a'). This contrived example means we can now use our new custom selector to collect all the links from the page using the following syntax:

```
$(':justLink')
```

Notice that we don't particularly have prefix the selector with anything, this means to run the selector against every element on the page (which isn't typically a good thing - but works for our example).

For our recipe, we want something a little more complicated. One tricky selector that's been requested a number of times on the jQuery IRC channel (see Appendix B [note to self appendix b - is getting help]) is how to select an element that isn't within another particular element.

We could solving this using a function, but since it's been requested a few times now, and it might be useful to have as a reusable utility, we'll create a custom selector so we can reuse it at a later date.

SOLUTION

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset=utf-8>
<title>Chapter 5 - Recipe 5</title>
<script src="jquery-latest.js"></script>
<script>
$.expr[":"].notin = function (el, i, match) {
  return !($(el).parents(match[3]).length);
};
$(document).ready(function () {
  $('a:notin(ul)').each(function () {
    $('#debug').append('found: ' + this.hash + '<br />');
  });
});
</script>
</head>
<body>
  <ul>
    <li><a href="#one">1</a></li>
    <li><a href="#two">2</a></li>
    <li><a href="#three">3</a></li>
  </ul>
  <a href="#outside">Link outside</a>
  <div id="debug"></div>
</body>
</html>
```

Running this code gives us:

found: #outside

As we've filtered down to only those anchor elements that are not in the unordered list element.

DISCUSSION

The custom selector is the following part of code:

```
$.expr[":"].notin = function (el, i, match) {
  return !($(el).parents(match[3]).length);
};
```

A selector receives a number of arguments in the following order:

1. Current element (el).
2. Index of that element in the matched elements (i).
3. An array that contains the expression be queried (match), if anything is being passed in to the selector, then that value being passed in is accessed from the four array element (as the array is zero-based, you access it via match[3]).

To use the selector, we're running the following:

```
$('a:notin(ul)')...
```

In English, we're finding all the anchor elements, and only returning those elements that do not have a parent ul element (anywhere in the DOM hierarchy, i.e. we're checking all parent nodes going right back up to the html element).

Keeping in mind that a custom selector works like a filter against what we have already, from the custom selector, we return true if we want to include the element in the list, and false if we want to exclude it.

Our custom selector runs the following jQuery against each anchor element:

```
$(el).parents(match[3]).length
```

...and returns false if any elements are found.

match[3] is the "ul" bit, i.e. the argument passed in to the custom selector (in bold below):

```
a:notin(ul)
```

This way we're able to determine whether the element in question (el) is inside the ul, and if it is, return false. Otherwise we return true. The result is a filtered down list of elements who are not in the element passed in as an argument to notin.

There's already a good collection of custom selectors in jQuery out of the box, in addition James Padolsey has a great article on his blog showing a collection of jQuery custom selectors, including one to query any data set against an element: http://james.padolsey.com/javascript/extending-jquerys-selector-capabilities/

Summary

Enhancing and fixing the browsers where they fall short of our expectations, I believe, should be done carefully. By that, I specifically mean to ensure you're choosing the right bug to fix. If the same solution can be achieved without having to rely on jQuery then that's good - because really you'd rather there was the bug or deficiency in the browser in the first place, so you wouldn't have to fall back to jQuery to solve the issue.

However, using the jQuery library to solve these issues can be extremely cost effective, perhaps in monetary terms for your business, but most certainly in terms of the time (and probably hassle) it saves you to solve the issue.

Since jQuery plugins are so easy and accessible to use, there's a great number of plugins available on the Internet. This fact alone means that many problems have been solved already, the PNG transparency fix is a great example. Instead of having to understand the deep going ons of the PNG bug, we can use a reliable jQuery plugin, Super Slight, to do all the hard work for us.

Equally, because jQuery relies on CSS selectors as it's method of navigating the DOM, and jQuery has very good support for CSS3 selectors, we can exploit that, target the DOM in ways we perhaps can't do using CSS today, and apply the appropriate styles.