A LIST APART

Now You See Me

by **Aaron Gustafson** · May 03, 2011

Published in JavaScript

A note from the editors: We are delighted to present an excerpt of Aaron's new book *Adaptive Web Design*.

Perhaps the most heavily-repeated pattern in JavaScript-based page manipulation is showing and hiding content. Tabbed interfaces. Collapsible elements. Accordion widgets. It crops up nearly everywhere. In and of itself, this pattern is not a bad thing, but few people realize how profoundly your choice of hiding mechanism can influence the accessibility of your content to assistive technologies like screen readers.

When building custom JavaScript-based widgets, it's quite easy to fully control the hiding mechanism, but when you begin working with animation libraries like jQuery or Scriptaculous, the hiding mechanism is typically dictated by the library, leaving you little control over the accessibility of your content. And that's a problem.

Peek-a-boo

When it comes to hiding content, there are several mechanisms for doing it and each affects the page differently, as summarized in the table below:

Mechanisms for Hiding Content

CSS Rules	Display Effect	Accessibility Effect
`visibility: hidden;`	Element is hidden from view, but is not removed from the normal flow (i.e., it still	Content is ignored by screen readers

	takes up the space it normally would)	
`display: none;`	Element is removed from the normal flow and hidden; the space it occupied is collapsed	Content is ignored by screen readers
`height: 0; width: 0; overflow: hidden;`	Element is collapsed and contents are hidden	Content is ignored by screen readers
`text-indent: -999em;`	Contents are shifted off-screen and hidden from view, but links may “focus” oddly and negative indent may not prove long enough to fully hide content	Screen readers have access to the content, but the content is limited to text and inline elements
`position: absolute; left: -999em;`	Content is removed from the normal flow and shifted off the left-hand edge; the space it occupied is collapsed	Screen readers have access to the content

The first two mechanisms are probably the most popular, with `display: none;` being the go-to option implemented by nearly every JavaScript library on the planet and the lion’s share of ready-made JavaScript widgets. If you don’t want your hidden content to be read by a screen reader, those defaults may work for you, but if you want to ensure users have access to content (even if it isn’t displayed visually in the current interface), the final option (positioning content offscreen) is really the way to go.

Note: the offscreen positioning assumes a left-to-right language page. For right-to-left, swap a `right` offset for the `left` one.

Hollaback scripts

If you roll your own JavaScript library, positioning content off-screen to hide it is pretty easy to implement. If, however, you are using a third-party JavaScript library, such as jQuery or Prototype, this task becomes much more difficult to accomplish because making the change requires overwriting or otherwise changing the internals of the library. Unless, of course, you’re smart about how you do it.

Most libraries include, as part of their animation suite, a mechanism for including what are referred to as “callback functions.” A callback function is a function that you supply to another function (or object method) so that it can be called at a predetermined time. If

you've used JavaScript to load content via Ajax, you're probably familiar with the concept: callback functions are used to do something with the data you got back from the server.

In most cases, JavaScript libraries only offer a callback function that runs at the completion of a given activity, but some libraries also provide hooks for various other points during the execution of a given routine, such as before the routine begins. Even without additional callback hooks, however, it's possible to create more accessible show/hide operations. Take the following jQuery-based snippet, for example:

```
(function(){
  var $button = $('#myButton'),
      $text   = $('#myText'),
      visible = true;
  $button.click(function(){
    if ( visible ) {
      $text.slideUp('fast');
    } else {
      $text.slideDown('fast');
    }
    visible = ! visible;
  });
})();
```

This script finds two elements (`#myButton` and `#myText`), assigning them to two local variables (`$button` and `$text` , respectively) before setting a third local variable (`visible`) to track the current state of things. It then goes on to assign an `onclick` event handler to `#myButton` that toggles the visibility of `#myText` by adjusting its height. Pretty straightforward, right?

This script works as you'd expect, but jQuery currently uses `display: none` when you call `slideUp()` , so `#myText` is being hidden via a method that prohibits the hidden text from being read by a screen reader. By making a subtle tweak to the code, however, we can trigger the addition of a `class` we control that provides for a more accessible means of hiding content:

```
(function(){
```

```
var $button = $('#myButton'),
    $text   = $('#myText'),
    visible = true;
$button.click(function(){
  if ( visible ) {
    $text.slideUp('fast',function(){
      $text.addClass('accessibly-hidden')
           .slideDown(0);
    });
  } else {
    $text.slideUp(0,function(){
      $text.removeClass('accessibly-hidden')
           .slideDown('fast');
    });
  }
  visible = ! visible;
 });
})();
```

This script is almost identical to the last one, in that when the content is being hidden, the library is allowed to manage the animation, but then the script swaps the default completion state for our custom `class` "accessibly-hidden," thereby keeping the content available to assistive devices. When the script goes to show the content, the steps are reversed, with the content being hidden by the script again before the `class` is removed and the actual animation is performed.

The added benefit of this approach is that you control the method of hiding content completely, as opposed to leaving it up to the JavaScript library. That means you can upgrade your "accessibly-hidden" to use a different technique if something better comes along and you don't have to wait for the library to upgrade its hiding mechanism (if it ever does).

JavaScript is a powerful tool for building rich interactions online. We already know that, when we do it, we need to be unobtrusive *(http://www.alistapart.com/articles/behavioralseparation)* and build the behavior layer following progressive enhancement

(http://www.alistapart.com/articles/progressiveenhancementwithjavascript/), but sometimes that only gets us so far. Thankfully, however, with a little thoughtful reflection, we can take our scripts that extra step and make our pages just a little more accessible.

About the Author

Aaron Gustafson

As would be expected from a former manager of the Web Standards Project, Aaron Gustafson is passionate about web standards and accessibility. He has been working on the web for nearly two decades and recently joined Microsoft as a web standards advocate to work closely with their browser team. He writes about whatever's on his mind at aaron-gustafson.com *(http://www.aaron-gustafson.com/)*.

MORE FROM THIS AUTHOR

Stop Forking with CSS3 *(http://alistapart.com/article/stop-forking-with-css3)*, Progressive Enhancement with JavaScript *(http://alistapart.com/article/progressiveenhancementwithjavascript)*, Progressive Enhancement with CSS *(http://alistapart.com/article/progressiveenhancementwithcss)*