/JavaScript/create a tabbed interface

Your essential CD All the files you'll require for this tutorial can be found on this issue's CD.

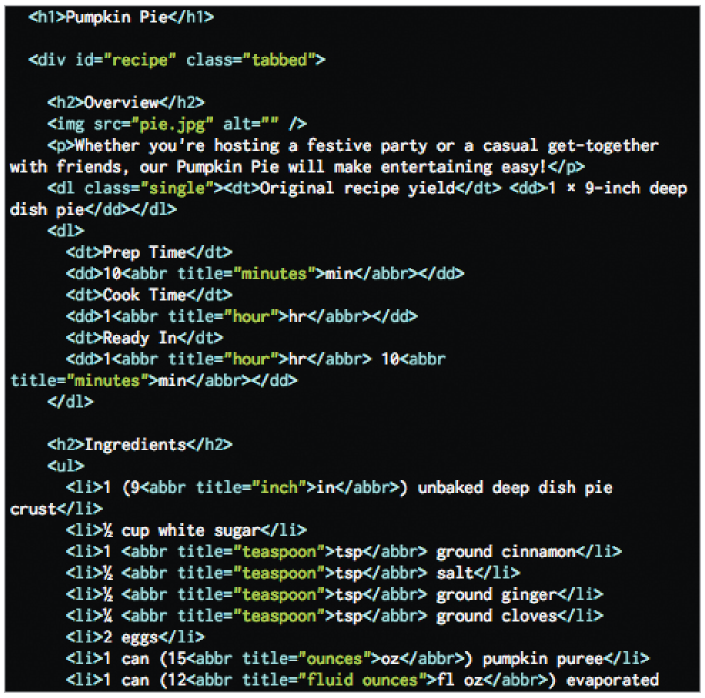
Part one In the first of a two-part tutorial, Aaron Gustafson demonstrates how to create a script for a lightweight yet powerful tabbed interface using CSS and basic JavaScript

Knowledge needed	Basic JavaScript, (x)HTML, CSS
Requires	A text editor
Project time	30-45 minutes

I've always been intrigued by the concept of a tabbed interface. My main issue with previous attempts at building one was that they almost always required you to add extra markup to the document – such as section wrappers and a list of links that would become the tabs – so they could be leveraged by JavaScript when constructing the interface.

I wanted to find a way around manually adding that cruft. After all, that extra markup was useless without JavaScript on, so it made no sense for it to be hard-coded into the document to begin with. JavaScript is perfectly adept at manipulating the DOM, so why not use it to inject all of the code necessary to drive the tabbed interface when we know it can actually be used? In other words, I was yearning for an unobtrusive way to construct a tabbed interface.

Generating the markup needed for a widget via script is nothing new, but the challenge lies in determining where to break the content up. I'd always felt the need for some sort of markup-based hook to indicate where the content chunks should start and end. Then I realised that headings (h1-h6) – because they create a document outline – were exactly the kind of natural language section divider that I was looking for.



```
  <h1>Pumpkin Pie</h1>

  <div id="recipe" class="tabbed">

    <h2>Overview</h2>
    <img src="pie.jpg" alt="" />
    <p>Whether you're hosting a festive party or a casual get-together
with friends, our Pumpkin Pie will make entertaining easy!</p>
    <dl class="single"><dt>Original recipe yield</dt> <dd>1 × 9-inch deep
dish pie</dd></dl>
    <dl>
      <dt>Prep Time</dt>
      <dd>10<abbr title="minutes">min</abbr></dd>
      <dt>Cook Time</dt>
      <dd>1<abbr title="hour">hr</abbr></dd>
      <dt>Ready In</dt>
      <dd>1<abbr title="hour">hr</abbr> 10<abbr
title="minutes">min</abbr></dd>
    </dl>

    <h2>Ingredients</h2>
    <ul>
      <li>1 (9<abbr title="inch">in</abbr>) unbaked deep dish pie
crust</li>
      <li>¾ cup white sugar</li>
      <li>1 <abbr title="teaspoon">tsp</abbr> ground cinnamon</li>
      <li>½ <abbr title="teaspoon">tsp</abbr> salt</li>
      <li>½ <abbr title="teaspoon">tsp</abbr> ground ginger</li>
      <li>¼ <abbr title="teaspoon">tsp</abbr> ground cloves</li>
      <li>2 eggs</li>
      <li>1 can (15<abbr title="ounces">oz</abbr>) pumpkin puree</li>
      <li>1 can (12<abbr title="fluid ounces">fl oz</abbr>) evaporated
```

Off the hook This script relies only on a single hook: a class of **tabbed**. All other necessary markup is generated by the script when it runs

Feeling inspired, I set out to write the script we'll walk through here. As with any great piece of writing, I needed a goal and a plan of action for reaching it. What was the script going to do exactly? What were the constraints? What sort of flexibility needed to be built in? I started with a list of requirements. It needed to: work in any container-type element (most likely a div, but not necessarily); separate the content into tabs based on the first encountered heading level; and offer a good deal of flexibility for styling.

With those guidelines, I decided on the following steps for the code to use as a blueprint: find any element classified as **tabbed** (that single class would be the hook that triggers the script); parse the contents of that element, breaking the content into chunks based on the first heading level encountered within it; generate the wrapper markup for each section and insert the content; generate the tab allowing access to that content; assign the appropriate event handlers to the tabs; and append it all back into the document.

The steps seemed pretty straightforward except the chunking bit; that would require a little regular expression magic, but we'll get to that shortly.

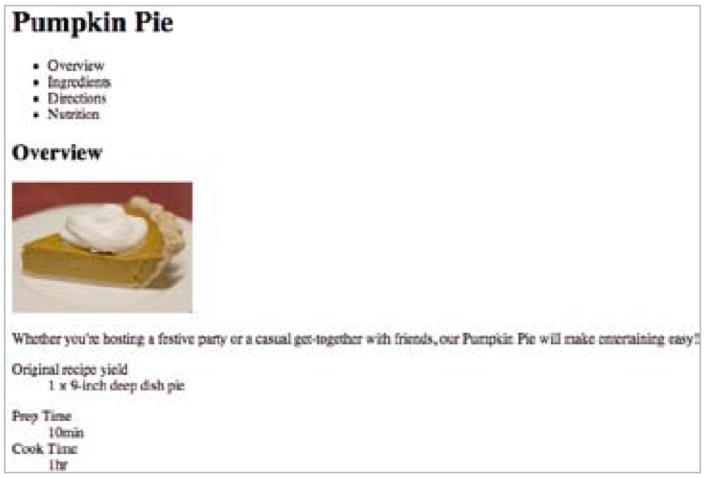
With a plan in place, I threw together a simple page with a recipe for making a pumpkin pie. Here's a rough overview of its markup:

JavaScript is perfectly adept at manipulating the DOM, so we'll use it to inject the code

```
<h1>Pumpkin Pie</h1>
<div class="tabbed">
 <h2>Overview</h2>
 <img src="pie.jpg" alt="" />
 <p>Whether you're hosting a festive party or a casual get-together
 with friends, our Pumpkin Pie will make entertaining easy!</p>
 ...
 <h2>Ingredients</h2>
 ...
</div>
```

Next, I manually tweaked the page to determine what markup I wanted to use in the final markup and what hooks I needed for styling. I opted for a fairly traditional structure of wrapper divs, with the whole thing contained within that initial container classified as **tabbed**. I decided the script would use variable names based upon the concept of a filing cabinet (folders, tabs, etc) and I carried it through to the generated markup as well, classifying each division as a 'folder', with the first one receiving a class of **visible** since it would be the one shown by default. I also classified each h2 as 'hidden' so they could be removed from view, and then added the list of tabs to the end as **ul.tab-list**, giving the currently active tab a class of, well, **active**.

Finally, I changed the overall container's class from **tabbed** to **tabbed-on**. The script itself will look for **tabbed** and then swap it for this new class



Pumpkin Pie

- Overview
- Ingredients
- Directions
- Nutrition

Overview

Whether you're hosting a festive party or a casual get-together with friends, our Pumpkin Pie will make entertaining easy!

Original recipe yield
1 x 9-inch deep dish pie

Prep Time
10min
Cook Time
1hr

Bare bones With no styles applied, the page looks pretty ugly, but is still usable

when it's ready to run in order to ensure styles are not applied prematurely. This technique, known as class-swapping, is a great strategy for maintaining separation between presentation and behaviour. With the markup in place, I wrote some basic styles to lay out the tabbed interface. Of particular note is how I chose to hide the inactive content sections:

```
.tabbed-on .folder {
  position: absolute;
  top: 0;
  left: -999em;
}
```

and then make them visible again:

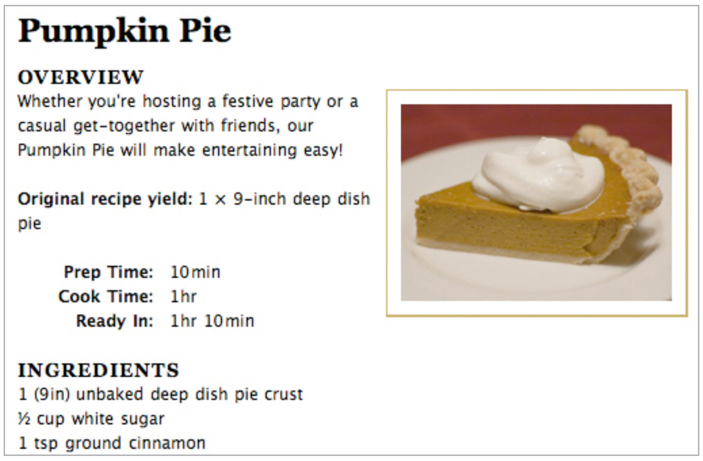
```
.tabbed-on .folder.visible {
  position: static;
}
```

Armed with a plan and the knowledge of the markup I'd need to make it all work, I set about writing the script that we'll now build together. To start things off, create an object constructor called **TabInterface**:

```
function TabInterface(){}
```

It's really nothing more than a function but, as everything in JavaScript is an object, it's also an object and it can be instantiated as many times as necessary on a page.

So, using a library like jQuery, we could say:



Pumpkin Pie

OVERVIEW

Whether you're hosting a festive party or a casual get-together with friends, our Pumpkin Pie will make entertaining easy!

Original recipe yield: 1 x 9-inch deep dish pie

Prep Time: 10min
Cook Time: 1hr
Ready In: 1hr 10min

INGREDIENTS

1 (9in) unbaked deep dish pie crust
½ cup white sugar
1 tsp ground cinnamon

Styled up With basic typographic and colour styles applied, the page looks a little better and is completely functional, even in the absence of JavaScript

```
$(document).ready(function(){
  $(".tabbed").each(function(){
    new TabInterface();
  });
});
```

This little bit of code (which could be replicated easily in the library of your choice or via native JS methods) runs when the DOM is loaded and scans through the document, creating a new **TabInterface** instance for every element it encounters with a class of **tabbed**. Of course, as we only have the skeleton of a constructor, the **TabInterface** object isn't set up to capture and manipulate the content that needs tabbing, but we'll get there.

Construction time

There's no reason to expose **TabInterface** object's inner workings, so we'll build them all as private members of this object. As we'll be operating inside of a function, we'll need to create the object's properties as variables and its methods as functions. (And yes, functions can contain other functions.) Each property and method will remain private to **TabInterface** unless we expose it by directly assigning it as a property of this. The only member we'll expose is the version of the script (available as **TabInterface.Version** and set as **this.Version**), in case anyone wants to implement an extension to the script and wants to see which version it is. Here's a basic outline of the properties we need to create:

- **Version** – the current script version
- **_active** – the ID of the active folder (false by default)
- **_index** – a DOM-generated unordered list that will contain the tabs
- **_els** – an object literal containing two properties of its own
- **li** – a DOM-generated list item
- **div** – a DOM-generated division.

Of note is the final property, **_els**. JavaScript can create elements on the fly, but cloning an element (using **element.cloneNode()**) is much faster than generating a new element (using **document.createElement()**) each time you need it. As we'll be generating multiple divisions around the content chunks and multiple list items for our tab list, it makes sense to create those elements before we need them so we can clone them easily later on.

>>

Reign in your styles

Don't let them run amok!

If you've worked a lot with CSS, you'll know there's a balance to be struck with selectors. Make a selector too generic and you could unintentionally set a property on an element you didn't mean to target; make it too specific and your whole style sheet can erupt into an arms race of increasingly specific selectors when you want to override a property. When JavaScript enters the picture, things get a bit more complex.

For one, you have to make sure the styles needed for the JavaScript widget don't bleed over and begin to affect other elements on the page (ones that, for instance, share the same class). This can be avoided by hanging your styles off of a class or id that's specific to the JavaScript object. For example, if your script is **WickedCool.js**, make the wrapper element **.WickedCool** or **#WickedCool** and preface all related selectors with that unique hook.

Be wary of timing when you apply your widget-related styles. For example, if a user has JavaScript turned off, you don't want to have styles applied to the page that hide content the widget was supposed to make visible; the user will never be able to see it. This issue is easily resolved by hanging all of your styles off of a class that's not used directly in the markup, instead using one that's set by JavaScript.

This can be done in a few different ways: change the form of the employed class (eg from **change-me** to **changed**); append **-on** to the employed class (eg from **tabbed** to **tabbed-on**); add a second class (eg **on**). All of these options act as a switch that your script can trigger when it knows it's safe for the styles to be applied. It should be noted, however, that compound class selectors are not supported by IE6, so if you need to support that browser, don't consider the third option.

>> Once you've added those properties, create two new functions to serve as the core methods for **TabInterface**:

- **initialize** – the function that will build the tabbed interface
- **swap** – the method that will handle changing currently displayed content

Because we'll be doing some class manipulation, include two helper methods, **addClassName** and **removeClassName**, using the following code:

```
function addClassName( e, c ){
 var classes = ( !e.className ) ? [] : e.className.split( ' ' );
 classes.push( c );
 e.className = classes.join( ' ' );
}
function removeClassName( e, c ){
 var classes = e.className.split( ' ' );
 for( var i=classes.length-1; i>=0; i– ){
   if( classes[i] == c ) classes.splice( i, 1 );
 }
 e.className = classes.join( ' ' );
}
```

With the preliminaries out of the way, let's start building out **TabInterface** in earnest. The first thing we need to do is provide a means for the element that needs tabbing to be passed into the object so we can manipulate it. To do that, add an argument to the **TabInterface** function itself. While you're at it, add a second argument to accept an iteration number (which we'll put to good use in a moment). Name these two arguments **_cabinet** and **_i**, respectively, as you will need to reference them later. Your code should look (roughly) like this:

```
function TabInterface( _cabinet, _i ){
 this.Version = '0.3';
```

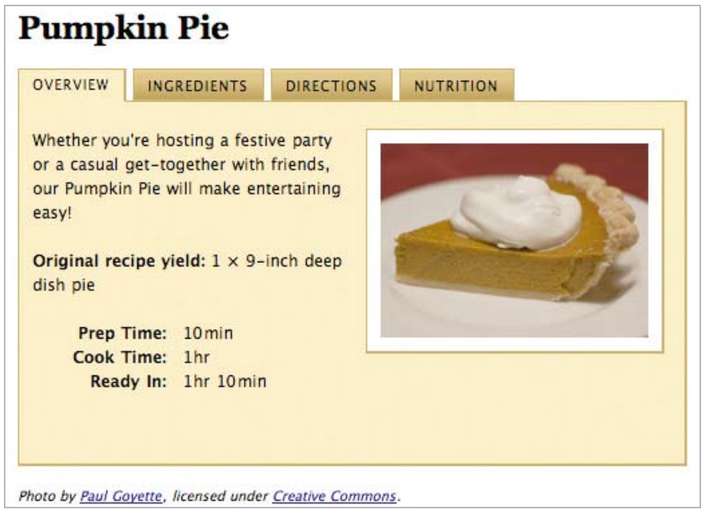
Now let's tackle **initialize()**. You'll want to update the initialisation of **TabInterface** (above) to pass in the arguments we just added, if you want to see your work in action as you build. To allow the script to work efficiently, we'll generate unique ids for each of the sections as well as the tabs. To keep things organised and keep the ids sensible, each should relate back to the id of the containing element (passed as **_cabinet**). Of course, **_cabinet** may not have an id, so we may need to generate one. We can do that using the passed iterator (**'cabinet-' + _i**). Once you have the id, store it to a local variable called **_id** so you can reference it later:

```
function initialize(){
 // set the id
 var _id = el.getAttribute( 'id' ) || 'cabinet-' + _i;
 if( !el.getAttribute( 'id' ) ) el.setAttribute( 'id', _id );
```

Resources Where to find out more

Understanding Progressive Enhancement
For a bit more background on the concept of progressive enhancement, check out this primer in A List Apart, as well as its two companion articles.
alistapart.com/articles/understandingprogressiveenhancement

Unobtrusive JavaScript for Progressive Enhancement
Phil Hawksworth put together this wonderful demonstration of how JavaScript can be used in smart ways, and helpfully it comes with an in-depth explanation of how to apply the same techniques to your projects.
unobtrusify.com



Pumpkin Pie

OVERVIEW | INGREDIENTS | DIRECTIONS | NUTRITION

Whether you're hosting a festive party or a casual get-together with friends, our Pumpkin Pie will make entertaining easy!

Original recipe yield: 1 × 9-inch deep dish pie

Prep Time: 10min
Cook Time: 1hr
Ready In: 1hr 10min

Photo by Paul Goyette, licensed under Creative Commons.

Tab happy With the widget's styles applied, the pumpkin pie recipe is transformed into an elegant, compact, tabbed interface

The next step is to divvy up the content by determining the heading level (h1-h6) upon which to base the chunking. But first, let's make the node iteration a little easier (and more cross-browser compatible) by stripping out any nodes of whitespace that are children of **_cabinet**:

We must provide a means for the element that needs tabbing to be passed into the object

```
var node = _cabinet.firstChild;
while( node ){
  var nextNode = node.nextSibling;
  if( node.nodeType == 3 &&
      !/\S/.test( node.nodeValue ) ){ _cabinet.removeChild( node ); }
  node = nextNode;
}
```

Then, you can determine the heading level of **_cabinet**'s **firstChild** by testing it against a list of known heading levels:

```
var headers = [ 'h1', 'h2', 'h3', 'h4', 'h5', 'h6' ];
for( var i=0, len=headers.length; i<len; i++ ){
  if( _cabinet.firstChild.nodeName.toLowerCase() == headers[i] ){
    var _tag = headers[i];
    break;
  }
}
```

This code block loops through the header types and tests each against the lower-cased tag name (**nodeName**) of the first child element (**firstChild**) of the container (**_cabinet**). When a match is encountered, the script assigns the correct heading level to **_tag** and the loop is broken.

Now that we have the heading level, we can chunk the content of **_cabinet** using regular expressions and innerHTML, storing the chunks as an array in the local variable **arr**. The split I employ uses a regular expression replacement that inserts an arbitrary, yet uncommon series of characters (||||) in front of the opening header tag before the content is split (creating an array) using those very same characters.

The first member of the array will be empty (because it comes before the first instance of the heading element), but **shift()** removes it:

```
var rexp = new RegExp( '<(' + _tag + ')', 'ig' );
var arr  = _cabinet.innerHTML.replace( rexp, "||||<$1" ).split( '||||' );
arr.shift();
```

With the content of **_cabinet** tucked safely away in **arr**, you can empty the container's contents and do a little class-swapping to turn on your styles:

```
_cabinet.innerHTML = '';
removeClassName( _cabinet, 'tabbed' );
addClassName( _cabinet, 'tabbed-on' );
```

Next, loop through **arr**'s members and create both a folder and a tab for each content chunk. The folders will be clones of **_els.div** and should be appended directly to **_cabinet**. The tabs will be cloned from **_els.li** and should be appended to the ul we created and stored as **_index**. The text for the tab will come from the heading element, and classifying the heading as **hidden** will allow you to hide it with CSS, keeping users from seeing the same text twice. Along the way, assign unique ids to the tabs and folders using **_id**.

While you're at it, add an onclick event handler to the tab (a reference to the swap method we'll get to in a moment), classify the first folder as **visible** and store its id in the **_active** variable, and then activate the current tab by giving it a class of **active**. Your code should look similar to this:

```
for( i=0, len=arr.length; i<len; i++ ){
  var folder = _els.div.cloneNode( true );
  folder.setAttribute( 'id', _id + '-' + i );
  folder.innerHTML = arr[i];
  addClassName( folder, 'folder' );
  _cabinet.appendChild( folder );
  var heading = folder.getElementsByTagName( _tag )[0];
  addClassName( heading, 'hidden' );
  var tab = _els.li.cloneNode( true );
  tab.setAttribute( 'id', 'id', _id + '-' + i + '-tab' );
  tab.innerHTML = heading.innerHTML;
  tab.onclick = swap;
  _index.appendChild( tab );
  if( i === 0 ){
    addClassName( folder, 'visible' );
    _active = _id + '-' + i;
    addClassName( tab, 'active' );
  }
}
```

Finally, with the loop complete, classify **_index** as an **index** and add it to **_cabinet**:

Out of the box In The Amanda Project (bit.ly/JKze6), Jason Santa Maria took the tab metaphor out of its traditional box and integrated it perfectly with the site's aesthetic

Button it An early version of this script was used on Nestlé's Idol Elimination site. The id-based hooks generated by this script enabled the addition of Next and Previous buttons

```
_index.className = 'index';
_cabinet.appendChild( _index );
```

With the markup adjustments complete, all that's left to do is fill in the logic for the swap event handler. The logic doesn't need to be very complex; it simply needs to swap visible folders and activated tabs.

Almost there

To get the currently active folder, tap into the **_active** property of **TabInterface**, as it contains the active folder's id. Then use the helper methods to move the appropriate classes from the currently active folder and tab to the newly activated ones, and update the **_active** property to refer to the newly opened folder:

```
function swap( e ){
 e = ( e ) ? e : event;
 var tab = e.target || e.srcElement;
 var folder_id = tab.getAttribute( 'id' ).replace( '-tab', '' );
 removeClassName( document.getElementById( _active + '-tab' ), 'active' );
 removeClassName( document.getElementById( _active ), 'visible' );
 addClassName( tab, 'active' );
 addClassName( document.getElementById( folder_id ), 'visible' );
 _active = folder_id;
}
```

The final step is to trigger **initialize()** to run when a new **TabInterface** is created. To do that, add a call to it at the end of the **TabInterface** function:

```
initialize();
```

In less than 100 lines of code, you've build a powerful, yet simple tabbed interface script, and you did it using a combination of regular expressions, object-orientation and unobtrusive scripting.

Don't miss part two of this article in next month's issue, in which we'll improve the flexibility of this script and make it more accessible using WAI-ARIA roles and states. ●

Note: TabInterface is available under the liberal MIT licence. The complete latest version of the script can be downloaded from GitHub at: easy-designs.github.com/tabinterface.js.

About the author

Name	Aaron Gustafson
Site	easy-designs.net
Areas of expertise	Front- and back-end development, strategy
Clients	Brighter Planet, Yahoo, Artbox.com
Favourite ice cream	Mint chocolate chip