/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 two In the last issue, **Aaron Gustafson** explained how to create a tabbed interface using CSS and JavaScript. Now he focuses on making the JavaScript more flexible and accessible

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

In the last issue, I walked you through how to create a very simple tabbed interface from semantically marked-up content (if you missed issue 194, there's a PDF of part one on the CD). Now, we'll revisit the JavaScript we wrote with a focus on making it more flexible and more accessible.

As you recall, the script we wrote segments content based on the natural language breaks created by headings. And, in the case of our example recipe, the script functioned perfectly because the headings were short and sweet – 'Ingredients', 'Directions', etc. But if this same script is applied to content with lengthier headings, the results would not be pretty (as the screengrab below shows). It would be simple for us to brush this off and require short titles from those who want to use the script, but that isn't very accommodating. In order to be truly useful, our script should offer a way for content authors to write without sacrificing the clarity of their prose, while simultaneously ensuring that the tabbed interface itself doesn't become unusable.

Looking at the problem from the perspective of a content author and a semanticist, the solution is clear: we need to provide an alternate label for the associated tab. That label is, in essence, advisory information about the section, and HTML just so happens to include an attribute – **title** – that serves that very function. Placing a **title** on the various heading levels grants authors full control over both the prose and the interface.

With a clear plan to accommodate alternate content for the tabs, we can adjust our script to take advantage of it. In our original script, we have code that worked great for handling shorter headings, so we can keep that, but we also need to be able to handle the new title-based tab labels, so we should tuck that into a conditional statement. If the heading has a title we'll use that, but if it doesn't, we'll default to the old method of grabbing the heading content.

Title inflation Nothing destroys our nice tabbed interface quite as quickly as long headings, so we need to provide an alternate label for the associated tab

```
heading = folder.getElementsByTagName( _tag )[0];
addClassName( heading, 'hidden' );
tab = _els.li.cloneNode( true );
tab.setAttribute( 'id', _id + '-' + i + '-tab' );
tab.onclick = swap;
tab.innerHTML = heading.innerHTML;
heading = folder.getElementsByTagName( _tag )[0];
if ( heading.getAttribute( 'title' ) ){
 tab.innerHTML = heading.getAttribute( 'title' );
} else {
 tab.innerHTML = heading.innerHTML;
 addClassName( heading, 'hidden' );
}
```

Note: In the case of the alternate tab label, the redundancy of having a matching tab label and heading is no longer a concern, so we won't need to add a class of **hidden** to the real heading in that instance.

We now have a much more flexible version of TabInterface, but we haven't explored the accessibility of this widget beyond the use of positioning to hide content. Most devs would stop at this point, pat themselves on the back and go out for a pint, assuming that most users of assistive technologies surf with JavaScript turned off and that those who don't would be accommodated by the content-hiding method we employed. I'm not so easily satisfied. Are you?

When JavaScript-based interactivity began spreading across the web like wildfire (thanks in no small part to the advent of Ajax), several people within the accessibility community began to realise that JavaScript and accessibility need not be mutually exclusive; in fact some, most notably Derek Featherstone, began evangelising that JavaScript could even improve accessibility.

The key to making assistive technology and JavaScript get along was finding a mechanism by which the markup could declare how it was being used so that screen readers and other devices would be capable of letting the user know what was happening and how to interact with various JavaScript-driven widgets. At the W3C, this concept was realised in the Web Accessibility Initiative's Accessible Rich Internet Applications (WAI-ARIA) spec. And one of the standard widget types addressed in the spec just so happens to be a tabbed interface. (Fancy that.)

Role playing

To make TabInterface WAI-ARIA-savvy, we need to use what the spec refers to as 'roles' and 'states'. Both terms mean exactly what you'd assume: roles are the parts elements are playing within the document and states indicate what each one is doing. In terms of the tabbed interface, we'll deal primarily with the following roles (indicated using the **role** attribute):

- **application**: identifies the page as being an application, which tells assistive devices to allow JavaScript to manage keystroke events
- **tab**: denotes that the element is acting as a tab within a tabbed interface

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

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

    <h2 title="Overview">This is absolutely the best pumpkin pie
you&#8217;ll ever taste</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 x 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
```

Information provider The **title** attribute is a simple way to provide advisory information that can be used as the text for our tabs

- **tablist**: denotes the element contains tabs that are part of a tabbed interface
- **tabpanel**: denotes the element is a panel of content within a tabbed interface

These roles provide the assistive device with the various players in our production, but don't tell the whole story. Without some sort of indication, we have no idea which tab is associated with which tabpanel. Thankfully, WAI-ARIA has that covered with two additional state-related attributes:

- **aria-labelledby**: an id reference to the associated tab, placed on the tabpanel
- **aria-describedby:** an id reference to the associated tabpanel, placed on the tab

To make TabInterface WAI-ARIA-savvy, we need to use 'roles' and 'states'

To indicate the current state of the tabbed interface, we'll also need these:

- **aria-hidden**: denotes whether or not the element is visible; takes a value of **true** or **false**
- **aria-selected**: denotes whether or not the element is currently activated; takes a value of **true** or **false**
- **aria-activedescendant**: placed on the containing element, this is an id reference to the currently active tabpanel

Set the properties

Setting these properties in the code is rather straightforward – just use the **setAttribute()** on the corresponding elements.

We'll get things rolling by adding the basics near the beginning of **initialize()**:

```
if( !_cabinet.getAttribute( 'id' ) ) _cabinet.setAttribute( 'id', _id );
// set the ARIA roles
_cabinet.setAttribute( 'role', 'application' );
_index.setAttribute( 'role', 'tablist' );
_els.div.setAttribute( 'role', 'tabpanel' );
_els.div.setAttribute( 'aria-hidden', 'true' );
_els.li.setAttribute( 'role', 'tab' );
_els.li.setAttribute( 'aria-selected', 'false' );
```

This puts us in a good position from the get-go by applying the necessary attributes to the various players, including our prototyped elements (remember that **cloneNode()** clones the attributes as well as the element when supplied an argument of **true**).

Further down in the code, we'll need to associate the tabs and tab panels that are being generated:

```
folder.setAttribute( 'id', _id + '-' + i );
folder.setAttribute( 'aria-labelledby', _id + '-' + i + '-tab' );
...
tab.setAttribute( 'id', _id + '-' + i + '-tab' );
tab.setAttribute( 'aria-describedby', _id + '-' + i );
```

Further on still, in the bit of code that establishes the first content chunk as the active one, we set the **aria-hidden** attribute to **false** (remember, it was **true** in the prototype element) and set the remaining states:

```
addClassName( folder, 'visible' );
folder.setAttribute( 'aria-hidden', 'false' );
tab.setAttribute( 'aria-selected', 'true' );
_active = folder.getAttribute( 'id' );
_cabinet.setAttribute('aria-activedescendant',_active);
```

Last but not least, we need to update the swap method to handle changing the requisite WAI-ARIA properties when closing one tab and opening another. In the interests of time, I'll leave that as an exercise for you

In depth Shawn Henry on how WAI-ARIA can help make our sites accessible

Shawn Henry

Job title Web accessibility evangelist
Sites www.uiaccess.com
About Shawn Shawn leads worldwide education and outreach activities at the W3C Web Accessibility Initiative

Websites are increasingly using functionality that's not available to some users with disabilities, especially people who rely on screen readers and people who can't use a mouse. Tree controls for site navigation, drag-and-drop and other complex user interface controls can be difficult or impossible to use because of limitations in today's development technologies.

A new technology to address this problem is on the horizon from the W3C: WAI-ARIA for Accessible Rich Internet Applications. WAI-ARIA defines a way to make websites and web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with Ajax.

A key aspect of WAI-ARIA is defining new ways for functionality to be provided to screen readers and other assistive technologies. WAI-ARIA techniques apply to widgets such as buttons, drop-down lists, calendar functions, tree controls, and others.

Another aspect of WAI-ARIA is providing robust and consistent keyboard access. Some people cannot use a mouse and use only a keyboard. Even relatively simple websites can be difficult if they require extensive keystrokes to navigate. (Try using your website without a mouse.)

WAI-ARIA describes new navigation techniques to mark menus, primary content, secondary content, banner information and other types of regions and web structures. Developers can then identify regions of pages and enable keyboard users to easily move among regions, rather than having to press tab repeatedly.

Web applications developed with Ajax, DHTML and other technologies can also have accessibility barriers. For example, if the content of a web page changes in response to user actions or time- or event-based updates, that new content may not be available to some people, such as people who are blind or who have cognitive disabilities and rely on a screen reader. WAI-ARIA includes technologies to map controls, Ajax live regions and events to accessibility APIs, including custom controls.

For making advanced web applications accessible and usable to people with disabilities, developers and users alike are singing the praises of WAI-ARIA. Check out www.w3.org/WAI/intro/aria.

>> because, while we may be close, we're not quite done yet. To truly make this widget accessible, we need to make it keyboard-aware.

Thinking beyond the mouse

Not everyone uses a mouse. People with disabilities are an obvious example, but power users often prefer the speed of a keyboard (hence the popularity of keystroke-based helper apps such as Enso and LaunchBar).

When it comes to keyboard events, WAI-ARIA recommends trying to mimic the desktop experience as much as possible. In our case, that means users need to be able to move into the tab list using the tab key. Once on a tab, they should be able to hit the tab key again to move into the associated tab panel (in order to have the content read to them) or navigate between the tabs using the arrow keys and other common commands (**Home**, **End** etc).

Movement from content block to content block is handled using the **tabindex** attribute: to remove an element from the tab order on a page, you can set its **tabindex** to **-1**; to establish its position within the source (and make it focusable), you use a value of **0**. This technique is often referred to as the 'roaming tabindex'. In the case of our tabs and tab panels, we need to remove their prototypes from the tab order by default and then reintroduce the active ones when they're activated:

```
_els.div.setAttribute( 'aria-hidden', 'true' );
_els.div.setAttribute( 'tabindex', '-1' );
_els.li.setAttribute( 'role', 'tab' );
_els.li.setAttribute( 'aria-selected', 'false' );
_els.li.setAttribute( 'tabindex', '-1' );
...
if( i === 0 ){
  ...
  folder.removeAttribute( 'aria-hidden' );
  folder.setAttribute( 'tabindex', '0' );
  tab.setAttribute( 'aria-selected', 'true' );
  tab.setAttribute( 'tabindex', '0' );
```

As with the application of the WAI-ARIA states, we also need to modify **swap()** to update the **tabindex** of the formerly and newly active elements. Again, I'll leave that to you. With the **tabindex** bit taken care of, we can move on to actually addressing the keyboard events. We'll start by adding two new event handlers to each tab:

To truly make this widget accessible, we need to make it keyboard-aware

```
tab.onclick = swap;
tab.onkeydown = moveFocus;
tab.onfocus = swap;
```

The first event handler captures all keystrokes, passing them to **moveFocus()**, a method we'll define in a moment. The second new event handler triggers **swap()** whenever a tab receives focus. As we've taken all but the first tab out of the tab order, this won't be invoked yet, but it's there to capture events we'll trigger shortly. Reading a user's keystrokes is quite simple once you get past the discrepancies between the standard W3C method of reading the keys and the Internet Explorer method. Thankfully, the spanner IE throws into the works isn't too large and it can be handled easily in a few lines of code:

```
function moveFocus( e )
{
 e = ( e ) ? e : event;
 var
```

Access all areas The YUI (Yahoo User Interface Library) team is working on implementing WAI-ARIA throughout its widget library

In depth **Glenda Sims** invites you to a meeting of JavaScripters Anonymous

Glenda Sims
Job title Senior systems analyst, University of Texas
Sites www.utexas.edu
About Glenda is an accessibility and web standards evangelist for UT, webstandards.org and knowbility.org

Do you suffer from compulsive use of JavaScript? Symptoms include:

- being frustrated by the limitations imposed by mobile browsers and assistive technology when exploring the edges of JavaScript
- using JavaScript "just for fun" (even when it isn't necessary)
- cursing while resuscitating JavaScript that ceased functioning after a new browser release
- living by the motto "No JavaScript. No Service"
- finding accessibility laws and guidelines overwhelming
- decrying the inability of search engine spiders to handle JavaScript

Is it possible for JavaScript to coexist peacefully with user environment demands and accessibility requirements? Yes! The answer is progressive enhancement. Web Standards Project co-founder Steve Champeon defines this as "a viable (web development) approach that enables the delivery of information as demanded by the users, while embracing accessibility, future compatibility, and determining user experience based on the capabilities of new devices" (bit.ly/RYIEv).

Imagine your site is like fine coffee. The raw content is the coffee beans. Your semantically marked-up content is the brewed coffee. CSS is the sugar and JavaScript is the cream. Using progressive enhancement, you start developing with quality content (beans) then build functional semantic code (coffee) that works for everyone and on everything.

Once your no-frills site works, you focus on presentation (sugar) and JavaScript (cream), which provide the rich user experience. No matter how good your creamy JavaScript and sweet CSS are, you must begin with quality coffee code.

The core of this approach is the philosophy of 'universal design'. As my colleague James Craig puts it, we should "design so thoughtfully it works for everyone from the start." By making your site universally satisfying you'll offer an experience that adapts to all customers' taste. This gives you the freedom to add JavaScript while achieving delicious SEO, sweet accessibility, hot mobile performance, and robust compatibility with future technologies.

```
 tab  = e.target || e.srcElement,
 key  = e.keyCode || e.charCode,
 pass = true;
 // keystroke handling goes here
 if ( ! pass )
 {
   return cancel( e );
 }
}
```

In this humble beginning to the **moveFocus** method, we've evened out the event models, as we did in **swap()**, before identifying the key that was pressed and the tab that currently has focus. We've also declared a variable, **pass**, that will define whether we should trap the keystroke or pass it on to other event handlers. By default we'll pass the keystroke on to be handled however it should be, but when we trap a keystroke we'll call another new method, **cancel**, that stops the event from bubbling (in a cross-browser-compatible way). You can check out the code for **cancel()** on the CD. Before we get to the actual handling of the keystrokes, let's define another method to control the movement (since that's most of what will be happening with each keystroke anyway):

```
function move( tab, direction, complete )
{
 if ( complete )
 {
   if ( direction == 'previous' )
   {
     tab.parentNode.childNodes[0].focus();
   }
   else
   {
     tab.parentNode.childNodes[tab.parentNode.childNodes.length-1].focus();
   }
 }
 else
 {
   var target = direction == 'previous' ? tab.previousSibling
                                        : tab.nextSibling;
   if ( target )
   {
     target.focus();
   }
 }
}
```

move() takes three arguments: the current tab (which we know), the direction (which we'll define, based on the keystroke), and whether or not the movement is intended to be made to the end of the tab list (which we'll also define based on the keystroke and will be influenced by the direction argument). With that method in place, we can return to **moveFocus()** and set it up to handle the necessary keystrokes.

We'll handle them with a simple **switch** statement:

```
switch ( key )
{
 case 37:
 case 38:
   move( tab, 'previous', false );
   pass = false;
   break;
 case 27:
   tab.blur();
   pass = false;
   break;
}
```

The above statement sets the actions for the left and up arrows (key codes 37 and 38, respectively) and **Escape** (27). Both arrows will trigger **move()** to focus on the previous tab (if it exists) and trap the key press. **Escape** will remove focus from the current tab and will, likewise, trap the key press.

Following this model, you should be able to create additional cases for the right and down arrows (39 and 40, respectively), **Home** (36) and **End** (35). Once complete, give the script a spin in your favourite browser and see how the whole thing works when using only your keyboard.

Standards bearer Dojo (whose TabContainer is part of the Dijits widget library) was one of the first JavaScript libraries to implement WAI-ARIA

Vigilance!

When programming in JavaScript (or any language for that matter), it's important to remember that each key you press affects a user's experience. Will you wield your power for good by engaging in progressive enhancement, keeping your scripts flexible and paying attention to accessibility concerns? I certainly hope so. But, if nothing else, perhaps this short series has opened your eyes up to how easy it can be to do the right thing.

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

Resources Where to find out more

Developing Accessible Widgets Using ARIA
video.yahoo.com/watch/4073211/10996186
Watch Todd Kloots give a fantastic presentation on how to integrate WAI-ARIA into UI widgets.

WAI-ARIA Best Practices
w3.org/TR/wai-aria-practices/
'The Roadmap for Accessible Rich Internet Applications'. The most recent version of the W3C's guide to working with ARIA can always be found at this URL.

About the author

Name Aaron Gustafson
Site easy-designs.net
Areas of expertise Front/back-end development, strategy
Clients Brighter Planet, Yahoo, Artbox.com
Which cancelled TV show would you most like to see brought back? *Pushing Daisies*