Extracted from:

Web Development Recipes

The Pragmatic Bookshelf

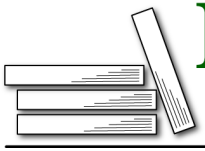
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Web Development Recipes

Brian P. Hogan,
Chris Warren,
Mike Weber,
Chris Johnson,
and Aaron Godin

edited by Susannah Davidson Pfalzer



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-683-8
Printed on acid-free paper.
Book version: P1.0—January 2012

Problem

When we need to present long, categorized lists on a website, the best way to do it is with nested, unordered lists. However, when users are presented with this kind of layout, it can be hard to quickly navigate, or even comprehend, such a large list. So, anything we can do to assist our users will be appreciated. Plus, we want to make sure that our list is accessible in case JavaScript is disabled or a user is visiting our site with a screen reader.

Ingredients

- jQuery

Solution

A relatively easy way to organize a nested list, without separating the categories into separate pages, is to make the list collapsible. This means that entire sections of the list can be hidden or displayed to better convey selective information. At the same time, the user can easily manipulate which content should be visible.

For our example, we'll start with an unordered list that displays our products grouped by subcategories.

Download collapsiblelist/index.html

```
<h1>Categorized Products</h1>

<ul class='collapsible'>
  <li>
    Music Players

    <ul>
      <li>16 Gb MP3 player</li>
      <li>32 Gb MP3 player</li>
      <li>64 Gb MP3 player</li>
    </ul>
  </li>
  <li class='expanded'>
    Cameras & Camcorders

    <ul>
      <li>
        SLR
        <ul>
          <li>D2000</li>
          <li>D2100</li>
        </ul>
      </li>
      <li class='expanded'>
        Point and Shoot
```

```
        <ul>
          <li>G6</li>
          <li>G12</li>
          <li>CS240</li>
          <li>L120</li>
        </ul>
      </li>
      <li>
        Camcorders
        <ul>
          <li>HD Cam</li>
          <li>HDR-150</li>
          <li>Standard Def Cam</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

We'll want to be able to indicate that some of the nodes should be collapsed or expanded from the start. It would be tempting to simply mark the collapsed nodes by setting the style to `display: none`. But that would break accessibility since screen readers ignore content hidden like this. Instead, we're going to rely on JavaScript to toggle each node's visibility at runtime. We did this by adding a CSS class of "expanded" to set the initial state of the list.

If we knew the user wanted to look at "Point and Shoot Cameras" when they first reached this page, for example, this markup wouldn't show the limited list yet. Right now it will show the full categorized product list, as shown in Figure 13, *Our full categorized list without collapsibility*, on page 7. But once the list is made collapsible, the user would see only the names of the products they were looking for, as shown in Figure 14, *Our collapsed list*, on page 7. Then, without navigating away from the page, they can still choose to look at any of our other product categories.

Next we need to write the JavaScript for adding our collapsible functionality, as well as some `Expand all` and `Collapse all` helper links at the top of the list. Notice that we're adding the links via the JavaScript code as well. Like the collapsible functionality itself, we don't want to change the markup unless we know this code is going to be used. This also gives us the advantage of being able to easily apply this behavior to any list on our site without having to change any markup beyond adding a `.collapsible` class to a `<ul>` element.

To start things off, we will write a function that toggles whether a node is expanded or collapsed. Since this is a function that will act on a DOM object, we will write it as a jQuery plug-in. That means we will assign the function

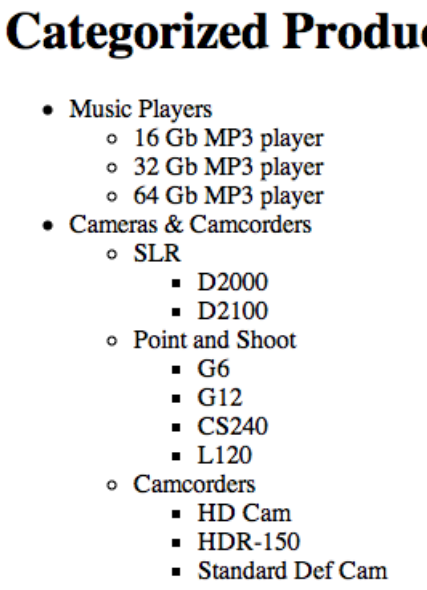


Figure 13—Our full categorized list without collapsibility

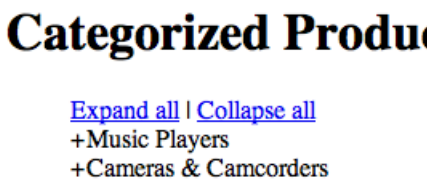


Figure 14—Our collapsed list

definition to the jQuery.fn prototype. We can then trigger the function within the scope of the element that it was called against. The function definition should be wrapped within a self-executing function so we can use the $ helper without worrying about whether the $ helper has been overwritten by another framework. Finally, to make sure that our jQuery function is chainable and a responsible jQuery citizen, we return this. This is a good practice to follow when writing jQuery plug-ins; *our* plug-in functions will work the same way that we expect other jQuery plug-ins to work.

Download collapsiblelist/javascript.js

```
(function($) {
  $.fn.toggleExpandCollapse = function(event) {
    event.stopPropagation();
    if (this.find('ul').length > 0) {
```

```
      event.preventDefault();
      this.toggleClass('collapsed').toggleClass('expanded').
        find('> ul').slideToggle('normal');
    }
    return this;
  }
})(jQuery);
```

We will bind the toggleExpandCollapse() to the click event for all <li> elements, including the elements with nothing underneath them, also known as *leaf nodes*. That's because we want the leaf nodes to do something crucial—absolutely nothing. Unhandled click events bubble up the DOM, so if we only attach a click observer to the <li> elements with .expanded or .collapsed classes, the click event for a leaf node would bubble up to the parent <li> element, which *is* one of our collapsible nodes. That means the code would trigger that node's click event, which would make it collapse suddenly and unexpectedly, and we'd be liable for causing undue harm to our users' fragile psyches. To prevent this Rube Goldberg–styled catastrophe from happening, we call event.stopPropagation(). Adding an event handler to all <li> elements ensures the click event will never bubble up and nothing will happen, just like we expect. For more details on event propagation, read *Why Not Just Return False?*, on page 9.

As mentioned at the beginning of the chapter, we want to give our users helper links that appear at the top of the list to toggle all of the nodes. We can create these links within jQuery and prepend them to our collapsible list. Because building HTML in jQuery can become verbose, we're better off moving the click event logic into separate helpers to prevent the prependToggleAllLinks() functions from becoming unreadable.

Download collapsiblelist/javascript.js

```
function prependToggleAllLinks() {
  var container = $('<div>').attr('class', 'expand_or_collapse_all');
  container.append(
      $('<a>').attr('href', '#').
      html('Expand all').click(handleExpandAll)
    ).
    append(' | ').
    append(
      $('<a>').attr('href', '#').
      html('Collapse all').click(handleCollapseAll)
    );
  $('ul.collapsible').prepend(container);
}

function handleExpandAll(event) {
  $('ul.collapsible li.collapsed').toggleExpandCollapse(event);
}
```

Joe asks:
Why Not Just Return False?

In a jQuery function, return false works double duty by telling the event not to bubble up the DOM tree and not to do whatever the element's default action is. This works for most events, but sometimes we want to make the distinction between stopping event propagation and preventing a default action from triggering. Or we may be in a situation where we always want prevent the default action, even if the code in our function somehow breaks. That's why at times it may make more sense to call event.stopPropagation() or event.preventDefault() explicitly rather than waiting until the end of the function to return false.[a]

a. http://api.jquery.com/category/events/event-object/

```
function handleCollapseAll(event) {
  $('ul.collapsible li.expanded').toggleExpandCollapse(event);
}
```

We can quickly create a DOM object by wrapping a string representing the element type we want, in this case an <a> tag, in a jQuery element. Then we set the attributes and HTML through jQuery's API. For simplicity's sake, we're going to create two links that say "Expand all" and "Collapse all" that are separated by a pipe symbol. The two links will trigger their corresponding helper functions when they're clicked.

Finally, we will write an initialize function that gets called once the page is ready. This function will also hide any nodes that were not marked as .expanded and add the .collapsed class to the rest of the <li> elements.

Download collapsiblelist/javascript.js

```
function initializeCollapsibleList() {
  $('ul.collapsible li').click(function(event) {
    $(this).toggleExpandCollapse(event);
  });
  $('ul.collapsible li:not(.expanded) > ul').hide();
  $('ul.collapsible li ul').
    parent(':not(.expanded)').
    addClass('collapsed');
}
```

We bind the click event to all of the <li> elements that are in a .collapsible list. We also added the expand/collapse classes to all of the <li> elements, except the products themselves. These classes will help us when it comes time to style our list.

When the DOM is ready, we'll tie it all together by initializing the list and adding the "Expand all" | "Collapse all" links to the page.

Download collapsiblelist/javascript.js

```
$(document).ready(function() {
  initializeCollapsibleList();
  prependToggleAllLinks();
})
```

Since this is a jQuery plug-in, we can easily add this functionality to any list on our site by adding a .collapsible class to an unordered list. This makes the code easily reusable so that any long and cluttered list can be made easy to navigate and understand.

Further Exploration

If we start out by building a solid, working foundation without JavaScript, we can build upon that foundation to add in extra behavior. And if we write the JavaScript and connect the behavior into the page using CSS classes rather than adding the JavaScript directly to the HTML itself, everything is completely decoupled. This also keeps our sites from becoming too JavaScript dependent, which means more people can use your sites when JavaScript isn't available. We call this *progressive enhancement*, and it's an approach we strongly recommend.

When building photo galleries, make each thumbnail link to a larger version of the image that opens on its own page. Then use JavaScript to intercept the click event on the image and display the full-sized image in a lightbox, and then use JavaScript to add any additional controls that are useful only when JavaScript is enabled, just like we did in this recipe.

When you're building a form that inserts records and updates the values on the screen, create the form with a regular HTTP POST request first, and then intercept the form's submit event with JavaScript and do the post via Ajax. This sounds like more work, but you end up saving a lot of time; you get to leverage the form's semantic markup and use things like jQuery's serialize() method to prepare the form data, rather than reading each input field and constructing your own POST request.

Techniques like this are well-supported by jQuery and other modern libraries because they make it easy to build simple, accessible solutions for your audience.

Also See

- Recipe 9, *Interacting with Web Pages Using Keyboard Shortcuts*, on page ?
- Recipe 11, *Displaying Information with Endless Pagination*, on page ?