Extracted from:

Web Development Recipes Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

2nd Edition

Web Development Recipes

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

edited by Rebecca Gulick

Web Development Recipes
Second Edition

Brian P. Hogan
Chris Warren
Mike Weber
Chris Johnson

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Eileen Cohen; Cathleen Small (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-056-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2015

Recipe 5

Creating and Styling Accessible Tooltips

Problem

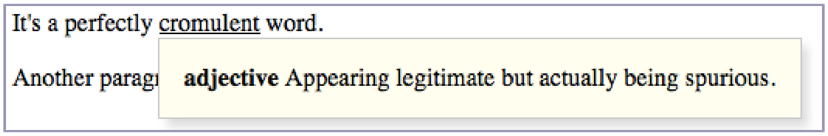
We have a page with lots of jargon, and we've been asked to build in functionality that lets visitors hover over terms to see their definitions. However, we have to ensure that the functionality can be used with assistive devices such as screen readers, since the page we're building will be accessed by people with disabilities.

Ingredient

- jQuery

Solution

With a small amount of CSS, some jQuery, the HTML5 ARIA specification,[7] and only a tiny amount of effort, we can create tooltips that work for everyone. When we're done we'll have something that looks like this:


It's a perfectly cromulent word.

Another paragr

adjective Appearing legitimate but actually being spurious.

We'll construct a library that'll work for widespread use throughout our site, but let's develop it by making a prototype page with a basic HTML skeleton:

accessible_tooltips/index.html
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Definitions</title>
    <link rel="stylesheet" href="tooltips.css">
  </head>
  <body>

  </body>
</html>
```

7. http://www.w3.org/TR/html5-author/wai-aria.html

The skeleton includes link to a style sheet file, tooltips.css, which will control the visibility of elements and the way our tooltips look. It'll also contain code that styles the word so it's apparent to users that they can interact with it.

Next, let's add some dummy text. We need a paragraph, and in that paragraph we want to have a specific keyword. When we hover over that word we want the definition to appear, so let's mark up the paragraph like this:

accessible_tooltips/index.html
```
<p>It's a perfectly
  <span class="definition" aria-describedby="def-test" tabindex="0">
    cromulent
    <span class="tooltip" id="def-test" role="tooltip">
      <b>adjective</b>
      Appearing legitimate but actually being spurious.
    </span>
  </span>
  word.
</p>

<p>Another paragraph of text.</p>
```

We place the keyword in a <span> tag, and we place the definition of that word inside its own <span>. We apply a tabindex to the outer <span> so that visitors can interact with the keyword via the keyboard by pressing the Tab key.

We also associate the keyword to its definition in our markup, using the aria-describedby tag, and we apply role="tooltip" to the element that makes up the tooltip. These small touches are what make interfaces more friendly to technologies like screen readers, which are used by blind and low-vision visitors who need the text on the screen read to them by the computer.

Now let's link up jQuery and our own custom tooltips.js file:

accessible_tooltips/index.html
```
<script
  src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js">
</script>
<script src="tooltips.js"></script>
```

We'll look through our document for any elements that have the definition class. For each one we find, we'll find its associated tooltip and hide it. But we won't use jQuery's show() or hide() methods. Instead, we modify the aria-hidden attribute of the tooltip, setting its value to true to ensure that screen-reading software is aware of the tooltip's visible state:

accessible_tooltips/tooltips.js
```
(function($){
  var $definitions = $('.definition');
  $definitions.find('.tooltip').attr('aria-hidden','true');
})(jQuery);
```

Then in tooltips.css we locate the elements with the aria-hidden attributes and style them appropriately:

accessible_tooltips/tooltips.css
```
.definition .tooltip[aria-hidden='true'] {
  display: none;
}

.definition .tooltip[aria-hidden='false'] {
  display:block ;
}
```

As soon as our JavaScript code sets the aria-hidden attribute to true, these CSS rules hide the element. And when we set the value to false, the elements show up again.

While we're here, let's add the styling for the definition. We add an underline to the word so we let users know it's something they can interact with. And we set the display property of the word we're defining to inline-block, which helps the definition appear closer to the word and ensures that any trailing spaces aren't underlined. We also add a slight drop shadow and a background to the tooltip:

accessible_tooltips/tooltips.css
```
.definition {
  display: inline-block;
  text-decoration: underline;
}

.definition .tooltip {
  background-color: #ffe;
  box-shadow: 5px 5px 5px #ddd;
  padding: 1em;
  position: absolute;
}
```

All that's left to do is apply the actual behavior. When the user hovers or tabs to a keyword, we want to show the definition. And when the user moves focus away, we want to hide it. That means we need to handle mouse events as well as focus events for keyboard navigation. That turns out to be pretty easy with jQuery:

accessible_tooltips/tooltips.js
```
function showTip(){
  $(this).find('.tooltip').attr('aria-hidden', 'false');
}
```

```
function hideTip(){
  $(this).find('.tooltip').attr('aria-hidden', 'true');
}

$definitions.on('mouseover focusin', showTip);
$definitions.on('mouseout focusout', hideTip);
```

And if you want to support other events, such as the touch events we work with in Recipe 25, *Mobile Drag and Drop* on page ?, you can add those to the event handlers, too.

That's all there is to it. When we open the page, we can hover over our word and see the definition. Best of all, because we applied a tabindex, we can activate it when we hit the Tab key also. And because the tooltip is associated with its parent, it should work well for screen-reading software.

Further Exploration

In our implementation, the tooltip is a child element of the element we hover on, and we used a <span> element, so we can't place <div> elements or other block-level elements in the tooltip. But it doesn't have to work that way. We could move the tooltip contents elsewhere in the markup and then use the aria-describedby role to locate the element and display its contents in our Java-Script code. Then we could place video content, images, or pretty much any-thing we want in that tooltip. And it would be accessible to everyone.

In this recipe we used our tooltips for definitions, but we can place any content we want, whether it's more information about a hyperlink or an inline help documentation for user interface items. Don't get carried away; the information you place should supplement the main content. After all, it does require interaction from the user to read the content you've hidden. Also, be sure you don't attach it to an element in such a way that it's triggered accidentally, obscuring the text on the screen. Some people track the words they read with the mouse, and surprise pop-ups won't keep you in their good graces.

Also See

- Recipe 31, *Cleaner JavaScript with CoffeeScript* on page ?
- Recipe 30, *Building Modular Style Sheets with Sass* on page ?
- Recipe 37, *Testing JavaScript with Jasmine* on page ?
- Recipe 25, *Mobile Drag and Drop* on page ?