Web Accessibility & Design Tutorials

Replacing <noscript> with accessible, unobtrusive DOM/JavaScript

Last update: 2017-02-18

Author: Frank M. Palinkas, Senior Technical Writer, Information Architect © 2015

Introduction

In his book "ppk on JavaScript" , Peter-Paul Koch points out that the HTML <noscript> element has a limitation. Modern user agents with JavaScript enabled will hide content contained within the `<noscript>` tags, and reveal it when JavaScript is disabled. User agents that do not support JavaScript will display the content within it.

User agents with partial/antiquated JavaScript capabilities however interpret the element correctly and do not show the content, but when JavaScript is disabled also do not show the content — it never gets rendered. This has an impact on the accessibility of the content.

If your writing is targeted at modern, standards-based, compliant, and fully capable JavaScript user agents, employing the `<noscript>` element is no problem. If the user agents among your audience are unpredictable, however, replacing the `<noscript>` element with another mechanism becomes significant. This article looks at one such solution.

Solving the `<noscript>` problem

So, how can behavior intended by the HTML `<noscript>` element be enabled in these antiquated/partially-capable JavaScript user agents? The workflow is as follows:

1. Insert an unobtrusive JavaScript "hook" via an id attribute value in the parent `<div>` container's opening tag.
2. Associate the hook with an external JavaScript function, which sets up a parent-to-child relationship between the `<div>` container and its content.

When JavaScript is enabled, the external function keeps the alternative content hidden. If the user agent cannot interpret the JavaScript expressions, the alternative content in the markup is revealed. When JavaScript is disabled the function no longer works, and the alternative content in the markup is revealed. This ensures graceful degradation of the script.

Important note: If the ability to hide the alternative content with JavaScript does not exist, the browser/user agent will be able to render the content ensuring accessibility and graceful degradation.

Building the HTML markup

In this section we'll build the HTML markup portion of the example.

1. In the `<head>` element tag of the web page, insert a `<script>` element containing the path to the external noscript.js file.
2. Create a `<div>` element within the `<body>` element tags to contain the hidden content.
3. Give the opening tag of the `<div>` element an `id` attribute with a value of `"noscript"`, that is, `<div id="noscript">`. This sets the unobtrusive hook to the external JavaScript noscript() function.
4. Place all elements and content to be hidden while JavaScript is enabled within the `<div id="noscript>...</div>` container.

The following HTML markup sample illustrates this:

```
<!DOCTYPE html>
    <html lang="en">
        <head>
            <title>noscript_example</title>
            <meta charset="UTF-8" />
            <script src="scripts/noscript.js" type="text/javascript"></script>
        </head>
    <body>
        <div id="noscript">
            <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</p>
            <ul>
                <li><a href="#">Hyperlink A</a></li>
                <li><a href="#">Hyperlink B</a></li>
            </ul>
        </div>
    </body>
    </html>
```

Building the unobtrusive DOM/JavaScript

In this section we'll build the two main sections of the external noscript.js JavaScript file.

Writing the `if()` conditional statement of the function

1. Between the if parentheses, place the document.removeChild expression. This tests to see if it is supported by the user agent.
    1. If it **is** supported the function will continue, and the `<div>` content will remain hidden.
    2. If it **is not** supported the function progresses to the `else if()` conditional statement and tests it.
2. Declare the container `<div>` element and identify its id value as the external, unobtrusive "hook" to the markup.
3. Instruct the parent `<div>` to remove all child content within it.

Writing the `else if()` conditional statement of the function

1. Between the `else if` parentheses, place the `document.getElementById` expression to test if it is supported by the user agent.
    1. If it **is** supported the function will continue, and the `<div>` content will remain hidden.
    2. If it **is not** supported the function will end, and the `<div>` content will be revealed, establishing graceful degradation of the script.
2. Return the element `id` where the style will be applied.
3. Set its `style.display` to `"none"` to control the presentation aspect of the child content.

The following script is contained in the noscript.js file which is kept external, and linked to the web page as described in the Building the HTML markup section.

```
/* noscript.js file */
function addLoadEvent(func) {
   var oldonload = window.onload;
   if (typeof window.onload !== "function") {
       window.onload = func;
   } else {
       window.onload = function () {
           if (oldonload) {
               oldonload();
           }
           func();
       };
   }
}
var noscript = addLoadEvent(noscript);
addLoadEvent(function () {
});
function noscript()
 {
   if (document.removeChild)
     {
       var div = document.getElementById("noscript");
           div.parentNode.removeChild(div);
     }
   else if (document.getElementById)
     {
       document.getElementById("noscript").style.display = "none";
     }
 }
```

Summary

This article demonstrates a solution that replaces the HTML `<noscript>` element with an external, unobtrusive JavaScript function named "noscript". This `noscript()` function solves the problem that partially-capable/antiquated JavaScript user agents have with not revealing alternative content if JavaScript is disabled. Employing this function reveals the content as intended. This ensures graceful degradation of the script, and full accessibility to the alternative content.

About the author: Frank M. Palinkas

Frank is an American living and working in Silicon Valley, California as a Senior Technical Writer, Information Architect/Developer. He authors all content, markup, presentation, behavior code using HTML5, CSS, RDFa 1.1, JSON, JSON-LD, Schema.org and Dublin Core Metadata Initiative (DCMI) ontologies, and Unobtrusive DOM/JavaScript. His technical writing incorporates web standards, accessibility, and linked/structured data.