

(ce document est une retranscription de l'article original de Chris Heilmann réalisé avec son accord. Cet article est [disponible en ligne sur son weblog](#))

Transcript of the Paris Web 2007 workshop on Unobtrusive JavaScript

This is a step-by-step description accompanied by code examples of the “Unobtrusive JavaScript” workshop at Paris Web 2007 in Paris, France.

You can download all the files used in this here: [parisweb2007_workshop.zip](#)

At the start of the workshop I gave two promises:

- Participants who know JavaScript will find out that they do often write far too much code
- Participants who don't know JavaScript will learn how they can prepare and help JS developers building unobtrusive products.

First Step: Analyzing the task and the HTML

We started with an HTML template ([01_template.html](#)) and analyzing it. We pointed out that the template is not optimal as it contains embedded CSS which in a live product should be in a separate CSS file. We also pointed out that it is OK to start like this, as it keeps maintenance easier.

The template is a list of links (a table of contents) pointing to a group of headings further down the page. The idea is that visitors can click the table of contents(ToC) and move down the document. “Back to top” links allow them to go back up again. The ToC has an ID of “toc”, the back links a class of “back”.

We have a block of CSS in the head of the document and a script block at the end of the body. Having a script block at the end of the body makes sure that all the HTML we want to reach in our JavaScript is available when it gets executed.

The task we wanted to achieve in the workshop was to turn this template into a dynamic interface that hides all the content and only shows the one connected to the link in the ToC when a visitor clicks it.

Second Step: Knowing browser issues and using them to our advantage

The problem we had was to know which section to show and hide when the links are clicked. The logical connection was easy: every link inside the ToC has an href attribute that points to an ID in the heading it connects to. We can use this in our script. The problem was that we have no idea about the content that follows the heading. We needed a way to make sure we can group all the elements we need to hide into an element.

The thing that helped us there is a lesser known [Internet Explorer bug](#) if you use your keyboard to tab through links pointing to anchors inside the page the page does jump there, but the keyboard focus does not move with it. The easiest workaround is to nest the anchor element inside an element that [hasLayout](#). In our case we added DIV elements around each heading and content section and set their widths to 100% ([02_iefix.html](#)).

Third Step: The script

We had all the HTML we needed and the page already did what we wanted to. Now we started thinking about how to override the default behaviour and make it work for us. The main changes we wanted to do was:

- Hide all the content sections
- Hide the back links as we don't need them when we show only one content section
- Show the content section connected with the link in the TOC when it is clicked

We discussed the different possibilities to do that (looping through the DOM, getting all child elements of the DIV and set their style.display value to 'none' and so on). I then proposed that the easiest way by far is to write a simple JavaScript that adds a class to the body of the document when JS is available and define CSS that hides the necessary elements. That way we can easily let CSS do all the heavy DOM traversing and we offer designers a hook to style the non-JavaScript and the JavaScript version differently.

The JavaScript:

```
document.body.className = 'js';
```

The CSS:

```
/* Scripting dependent styles */
body.js #toc{
    float:left;
    width:20%;
    margin-right:5%;
}
body.js div{
    float:right;
    width:70%;
}
body.js div,body.js .back{
    position:absolute;
    left:-9999px;
}
```

We then found out that this would not be safe, as there might already be a class applied to the body element, which means we need to check for that and append the new class if needed.

```
document.body.className = (document.body.className + ' ' || '') + 'js';
```

This made sure all the necessary elements are initially hidden ([03_hideandstyle.html](#)). We also pointed out that we use the off-left technique instead of hiding the content with display:none as that makes sure it is still accessible for screen readers.

Event Handling

This is as far as we got with JavaScript and the DOM. Now we needed to find out how we can show the section we want to show when the appropriate link is clicked. We need to use event handling for that and apply event listeners to the different elements. Originally the participants considered adding a click event listener to every event but after going through the idea of event handling using human guinea pigs ([video on dailymotion](#)) we realized that there is only need to apply a single event handler to the TOC and use [event delegation](#) to do the rest.

We used the W3C approach to event handling, which resulted in the following code ([04_events.html](#)):

CSS:

```
body.js div.show{
    position:relative;
    left:0;
}
```

JavaScript:

```
document.body.className = (document.body.className + ' ' || '') + 'js';
var toc = document.getElementById('toc');
if(toc){
    function toggle(e){
        var t = e.target;
        if(t.nodeName.toLowerCase() = 'a'){
            var sectionID = t.getAttribute('href').split('#')[1];
            var section = document.getElementById(sectionID);
            if(section){
                section.parentNode.className = 'show';
            }
        }
    };
    toc.addEventListener('click',toggle,false);
};
```

This made sure we can actually show the elements when we click on them but it also had the problem that we didn't hide the previously shown section. For this, we needed to store the information and reset it when `toggle()` was called. I mentioned as a tip that it is always a good plan to use an object to store state of an interface as that means you can keep as many properties as you want without needing to introduce more variables. The change was only a few lines storing the section in a current object and removing the class when there is already one set ([05_keepingstate.html](#)):

```
document.body.className = (document.body.className + ' ' || '') + 'js';
var toc = document.getElementById('toc');
if(toc){
  var current = {};
  function toggle(e){
    var t = e.target;
    if(t.nodeName.toLowerCase() = 'a'){
      if(current.section){
        current.section.className = '';
      };
      var sectionID = t.getAttribute('href').split('#')[1];
      var section = document.getElementById(sectionID);
      if(section){
        section.parentNode.className = 'show';
        current.section = section.parentNode;
      };
    };
  };
  toc.addEventListener('click',toggle,false);
};
```

This was it in terms of (very basic) functionality for the course (we only had an hour) and the last few things to remember was to make this code work with other scripts and how to make it easier to maintain.

Namespacing the script.

The easiest trick to make the script work well with others is to use [the module pattern](#) and nest it in a variable assigned to an anonymous function ([06_namespaced.html](#)):

```
var sectionCollapse = function(){
    document.body.className = (document.body.className + ' ' || '') + 'js';
    var toc = document.getElementById('toc');
    if(toc){
        var current = {};
        function toggle(e){
            var t = e.target;
            if(t.nodeName.toLowerCase() = 'a'){
                if(current.section){
                    current.section.className = '';
                };
                var sectionID = t.getAttribute('href').split('#')[1];
                var section = document.getElementById(sectionID);
                if(section){
                    section.parentNode.className = 'show';
                    current.section = section.parentNode;
                };
            };
        };
        toc.addEventListener('click',toggle,false);
    };
}();
```

Making maintenance easier

In order to make it easy for the next developer taking over from us, we then agreed to not have any ID or class names in the script itself but move them out to an own config object.

The last step we were able to cover was to move the CSS and JavaScript out into own documents and our example was done ([07_final.html](#)):

```
var sectionCollapse = function(){
    // start configuration - edit here
    var config = {
        JSavailableClass:'js',
        showClass:'show',
        tocID:'toc'
    }
    // end configuration
    document.body.className = (document.body.className + ' ' || '') +
config.JSavailableClass;
    var toc = document.getElementById(config.tocID);
    if(toc){
        var current = {};
        function toggle(e){
            var t = e.target;
            if(t.nodeName.toLowerCase() = 'a'){
                if(current.section){
                    current.section.className = '';
                };
                var sectionID = t.getAttribute('href').split('#')[1];
                var section = document.getElementById(sectionID);
                if(section){
                    section.parentNode.className = config.showClass;
                    current.section = section.parentNode;
                };
            };
        };
        toc.addEventListener('click',toggle,false);
    };
}();
```

Conclusion

I hope that the participants got an insight how you can make CSS and JavaScript can work together, and learnt some ways to make their JavaScript easier to maintain. I personally wanted most of all people to start analyzing problems before starting the code :)