Pocket Guides  Go Make Things

BUGS & BROWSER QUIRKS

with Vanilla JavaScript

CHRIS FERDINANDI

Bugs and Browser Quirks

By Chris Ferdinandi

Go Make Things, LLC

v2.0.0

Table of Contents

Intro

In this guide, you'll learn:

- The easy way to ensure cross-browser compatibility.
- How to catch bugs early… before they become a problem.
- How to scope your scripts to prevent conflicts with your variables and functions.
- How to debug your code when things go wrong.
- A simple trick that makes writing JavaScript faster and easier.
- How to put it all together and cleanup a small script.

Let's get started…

Cross-Browser Compatibility

Vanilla JavaScript browser compatibility can be inconsistent and difficult to figure out. This is one of the big allures for libraries and frameworks.

In this chapter, I'm going to teach you a super-simple technique you can use to ensure a great experience for your users, regardless of their web browser.

Support vs. Optimization

The web is for everyone, but support is not the same as optimization. [1]

Rather than trying to provide the same level of functionality for older browsers, we can use progressive enhancement to serve a basic experience to all browsers (even Netscape and IE 5), while newer browsers that support modern APIs and techniques get the enhanced experience.

To be clear, I'm not advocating dropping support for older and less capable browsers.

They still have access to all of the content. They just don't always get the same layout or extra features.

Cutting the Mustard

“Cutting the Mustard” is a feature detection technique coined by the BBC.[2]

A simple browser test determines whether or not a browser supports modern JavaScript methods and browser APIs. If it does, the browser gets the enhanced experience. If not, it gets a more basic one.

```
var supports = 'querySelector' in document && 'addEventLi
stener' in window;

if ( supports ) {
    // Codes goes here...
}

// or...

if ( !supports ) return;
```

What browsers are supported?

To quote the BBC:

> - IE9+
> - Firefox 3.5+
> - Opera 9+ (and probably further back)
> - Safari 4+
> - Chrome 1+ (I think)
> - iPhone and iPad iOS1+
> - Android phone and tablets 2.1+
> - Blackberry OS6+
> - Windows 7.5+ (new Mango version)
> - Mobile Firefox (all the versions we tested)
> - Opera Mobile (all the versions we tested)

If you're using `classList`

If you're using `classList`, you need to either include the polyfill,[3] or check for `classList` support. Without the polyfill, your IE support starts at IE10 instead of IE9.

```
var supports = 'querySelector' in document && 'addEventLi
stener' in window && 'classList' in document.createElemen
t('_');
```

Don't hide content until JavaScript loads

If you have an accordion widget, you might use some CSS like this to hide the content:

```
.accordion {
    display: none;
}
```

When JavaScript adds an `.active` class back on to that content, you show it again like this:

```
.accordion.active {
    display: block;
}
```

The problem with this approach is that if the visitor's browser doesn't support your JavaScript APIs, or if the file fails to download, or if there's a bug and it break, they'll never be able to access that content.

Add an activation class

In your scripts, include something like this as part of the initialization process:

```
document.documentElement.className += ' js-accordion';
```

This adds the `.js-accordion` class to the `<html>` element. You can then hook into that class to conditionally hide my accordion content *after* you know my script has loaded and passed the mustard test.

```
.js-accordion .accordion {
    display: none;
}
```

This ensures that all users can access that content, even if your script breaks.

Strict mode

Strict mode is a way of telling browsers (and JavaScript debuggers) to be, well, stricter about how they parse your code. MDN explains:[4]

> Strict mode makes several changes to normal JavaScript semantics. First, strict mode eliminates some JavaScript silent errors by changing them to throw errors.

This is highly desirable. I know that sounds counterintuitive. Why would you want more errors in your code?

Here's the thing: there were already errors in your code. The browser just wasn't telling you about them, so they might crop up in unexpected ways that are harder to find.

Turning on strict mode helps you find errors sooner, before they become bigger problems. And it forces you to write better code.

Always use strict mode on your scripts.

How do you activate strict mode?

Simple, just add this to your scripts:

```
'use strict';
```

Integrating with your text editor

Some text editors support plugins that will check your code for strict mode violations in real time as you code. SublimeLinter for Sublime Text[5] and AtomLint for Atom[6] are good options.

Debugging

Your code will break. While you're learning, it will break a lot. This is an inevitable aspect of being a web developer.

Let's talk about how to figure out what's going on and get it working again.

Developer tools and the console

If you're not already familiar with your browser's developer tools, and in particular, the console tab, you should play around with them a bit.

The console tab is where JavaScript errors will get displayed. The error will typically specify what the error is, what file it's occuring, and what line number it's on. It will also let you click that line number to jump directly to the error in the file.

You can also write JavaScript directly in the console window, which is useful for quickly testing out a few lines of code.

All modern browsers have developer tools baked in. They differ in subtle ways, so it's really a matter of preference which one you choose.

The debugging process

There's no magic process for debugging.

1. Identify the last working piece of code.

If there's an error in the console window, it will tell you a line number for the code that's throwing the error. Start there.

If not, go to the very first piece of code in your script that runs.

2. Use `console.log()` to log all of your work in the console window.

If you're using grabbing an element in the DOM with `querySelector`, log it an make sure you're actually getting the element. If you're running an event listener, console.log some text (like `console.log('it worked!');`) and then try to trigger the event, so that you can confirm it's working.

Keep working your way back through the code until you find the thing that's not behaving the way you expect it to.

This is a tedious, annoying process. But it will also help you learn a *lot* about how to structure code and avoid common issues. Over time, you'll have to do this less and less.

I typically start at the last working piece of code, using `console.log()` to check what's happening until I find the piece of code I messed up or that's returning an unexpected result.

Scoping

Scoping is a way of keeping variables and functions from conflicting with each other (often with unintended consequences).

At a high-level:

1. Functions have access to variables set outside themselves, but not variables set inside other functions.
2. Assigning a variable with `var` creates a new variable. Assigning it without overwrites a previously defined variable.

For example:

```
var sandwich = 'turkey';
var snack = 'chips';

console.log(sandwich); // "turkey"
console.log(snack); // "chips"

var updateLunch = function () {
    var sandwich = 'tuna';
    snack = 'carrots';
    var dessert = 'ice cream';

    console.log(sandwich); // "tuna"
    console.log(snack); // "carrots"
    console.log(dessert); // "ice cream"
};

updateLunch();

console.log(sandwich); // "turkey"
console.log(snack); // "carrots"
console.log(dessert); // undefined
```

Immediately Invoked Function Expression

There are times you may want to expose a function or variable to the global scope (for example, a lightweight framework you want other scripts to be able to use), but generally speaking, you want to keep your

functions and variables out of the global scope.

One way to do this is to wrap your code in an Immediately Invoked Function Expression (or IIFE). An IIFE is an anonymous (as in, unnamed) function that runs immediately. Because all of your code is inside a function, it's removed from the global scope.

```
;(function (window, document, undefined) {
    'strict mode';
    // Your code goes here...
})(window, document);
```

Planning out your script

This may sound ridiculous, but I actually plan out my scripts on paper before I ever open a text editor. Like, real paper. In a physical notebook. With a pen.

Code is really just a series of small steps, run one after another, that tell computers (or in our case, browsers) exactly what to do.

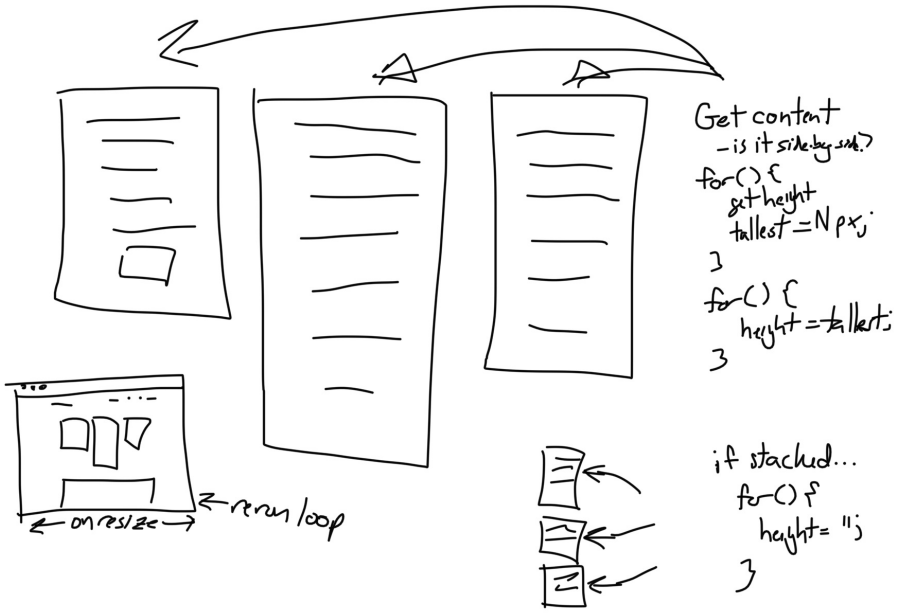
I find it helpful to identify what each of those steps should be ahead of time, which makes it easier for me to think through which functions and techniques I need to use, which elements on the page I need to target, and so on.

An example

Here's an actual example from when I created my Right Height plugin.[7]

Right Height will take a set of content areas that are different heights and make them all the same height. When those content areas are all stacked one-on-top-of-the-other (say, in a single-column mobile view), it leaves them at their natural height.

First, I sketched out what I was trying to accomplish on paper.



Sketching out rightHeight.js

Next, I identified and wrote out each of the steps.

1. Get all of the content areas in a group.
2. Determine if the content areas are side-by-side or stacked one-on-

top-of-the-other.

    - If they're side-by-side…
        1. Get the height of the tallest content area.
        2. Set each content area to the height of the tallest one.
    - If they're one-on-top-of-the-other…
        1. Remove any height setting and let the content flow naturally.

3. If the browser is resized, do it all over again.

And here's how that translated those steps into specific tasks for the script.

1. Get all of the content areas in a group.
    a. Wrap content in a parent element with a selector (like `.right-height` or `[data-right-height]`).
    b. Use `.querySelectorAll()` to get all content groups.
    c. Loop through each group, and use `.querySelectorAll()` to get all content areas within it.
2. Determine if the content areas are side-by-side or stacked one-on-top-of-the-other.
    - Get the distance from the top of the first two content areas to the top of the page. If they're the same, the content areas are side-by-side. If they're different, they're stacked.
    - If they're side-by-side…
        1. Get the height of the tallest content area.
            a. Set a variable for `height` to `0`.
            b. Loop through each content area and measure its

height. If it's greater than the current `height` variable value, reset `height` to that content area's height.

2. Set each content area to the height of the tallest one.
    a. Loop through each content area again and give it a `style.height` equal to the `height` variable value.

- If they're one-on-top-of-the-other…
    1. Remove any height setting and let the content flow naturally.
        a. Loop through each content area and set the `style.height` to nothing.

3. If the browser is resized, do it all over again.
    a. Add a event listener for browser resizing.

Obviously, there's a bit more to writing code than just outlining the steps. But this outline gives me a huge headstart on actually writing the script and helps keep me focused on the bigger tasks I need to get done.

Give it a try!

Putting it all together

To make this all tangible, let's clean up a small project together. We'll take a show/hide script and apply some of the techniques we learned in this guide to avoid bugs and browser errors.

The starter template and complete project code are included in the source code[8] on GitHub.

Getting Setup

Our script includes a small amount of CSS to show and hide content using `display: none` and `display: block`. It also includes an event listener and a few modern JavaScript methods.

CSS

```
.accordion {
    display: none;
}

.accordion.active {
    display: block;
}
```

JavaScript

```
// Listen for click events
document.addEventListener('click', function (event) {

    // Only run if the clicked link was an accordion togg
le
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it an
d quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        return;
    }

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll('.accordio
n');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
```

```
    }

    // Open our target content area
    content.classList.add('active');

}, false);
```

Markup

```
<a class="accordion-toggle" href="#content">Show Content<
/a>

<div class="accordion" id="content">
    The content
</div>
```

Scoping

First, let's wrap the code in our JavaScript file in an IIFE to prevent our variables and functions from bleeding into the global scope. We'll also add string mode.

```
;(function (window, document, undefined) {

    'strict mode';

    // Listen for click events
    document.addEventListener('click', function (event) {
        // ...
    }, false);

})(window, document);
```

Mustard Test

Our script makes uses of a few modern APIs, including `querySelector` and `classList`. Let's include a mustard test to make sure we don't throw any errors on older browsers.

```
;(function (window, document, undefined) {

    'strict mode';

    // Feature test
    var supports = 'querySelector' in document && 'addEve
ntListener' in window;
    if (!supports) return;

    // Listen for click events
    document.addEventListener('click', function (event) {
        // ...
    }, false);

})(window, document);
```

Access

Next, let's add add a class when the script loads, and use it as a hook in our CSS to make sure we don't hide any content until the JavaScript is available and our browser passes the feature test.

```
;(function (window, document, undefined) {

    'strict mode';

    // Feature test
    var supports = 'querySelector' in document && 'addEve
ntListener' in window;
    if (!supports) return;

    // Listen for click events
    document.addEventListener('click', function (event) {
        // ...
    }, false);

    // Add an activation class
    document.documentElement.className += ' js-accordion'
;

})(window, document);


.js-accordion .accordion {
    display: none;
}
```

Congratulations! You just cleaned up a project and helped avoid some common bugs and browser quirks.

About the Author

Hi, I'm Chris Ferdinandi. I help people learn JavaScript.

I love pirates, puppies, and Pixar movies, and live near horse farms in rural Massachusetts. I run Go Make Things with Bailey Puppy, a lab-mix from Tennessee.

You can find me:

- On my website at GoMakeThings.com.
- By email at chris@gomakethings.com.
- On Twitter at @ChrisFerdinandi.

1. http://bradfrostweb.com/blog/mobile/support-vs-optimization/↩
2. http://responsivenews.co.uk/post/18948466399/cutting-the-mustard↩
3. https://github.com/eligrey/classList.js/↩
4. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode↩
5. http://www.sublimelinter.com↩
6. https://atom.io/packages/atom-lint↩
7. https://github.com/cferdinandi/right-height↩
8. https://github.com/cferdinandi/bugs-and-browser-quirks-source-code/↩