Table of Contents

Introduction	1.1
Getting Setup	1.2
ECMA Script 5	1.3
Selectors	1.4
Loops	1.5
String Transforms	1.6
Classes	1.7
Styles	1.8
Attributes	1.9
Event Listeners	1.10
DOM Ready	1.11
HTML	1.12
DOM Injection	1.13
Forms	1.14
Traversing Up the DOM	1.15
Traversing Down the DOM	1.16
Traversing Sideways in the DOM	1.17
Merging Arrays & Objects	1.18
The Viewport	1.19
Distances	1.20
Query Strings	1.21
Ajax/HTTP Requests	1.22
Cookies	1.23
Cross-Browser Compatibility	1.24
Debugging	1.25
Planning	1.26
Putting It All Together	1.27
What Now?	1.28
AbouttheAuthor	1.29

The Vanilla JS Guidebook

Learn vanilla JavaScript and level-up as a web developer.

A step-by-step training guide to help you master vanilla JavaScript. Short, focused, and made for beginners.

If you're sick of hostile answers on StackOverflow, bad documentation, and tutorials that assume you have a Computer Science degree, this is for you.

What you'll learn

The Vanilla JS Guidebook covers everything you need to know to write vanilla JavaScript.

Modern vanilla JavaScript equivalents of common jQuery methods. The easy way to ensure cross-browser compatibility. How to write scripts that are accessible to everyone. How to debug your code when things go wrong. A simple trick to make writing JavaScript faster and easier. How to put it all together and create real, working scripts.

Who this is for

The Vanilla JS Guidebook is perfect for anyone who wants to level-up their core JavaScript skills:

Developers who can hack together some jQuery but don't feel like they really know JavaScript. People who want to learn other frameworks—like React or Angular—but don't feel like they even know where to start. Web developers who want to opt-out of the JavaScript framework rat race altogether. Front-end developers who want to build websites that are faster and more reliable. Anyone who wants a simpler, easier JavaScript experience.

Intro

This book is the missing manual for vanilla JavaScript. You'll learn...

- Modern vanilla JavaScript equivalents of common tasks you normally use a library or framework for.
- The easy way to ensure cross-browser compatibility.
- How to debug your code when things go wrong.
- A simple trick to make writing JavaScript faster and easier.
- How to put it all together and create real, working scripts.

Getting Setup

All of the source code for the lessons in this book are avilable on GitHub.

To make things easier, I've inlined everything. There's some basic CSS up in the `<head>`, some sample `<body>` content to work with, and all of your scripts are down at the bottom.

I make heavy use of `console.log()` in the source code to spit out the results of the lessons into the Console tab of Developer Tools. All modern browsers--Chrome, Firefox, Safari, and Microsoft Edge--have great browser tools baked right in.

I'd also recommend getting a good text editor. My text editor of choice is Sublime, but Atom (from GitHub) is a great free cross-platform alternative with most of the same features.

ECMAScript 5

ECMAScript 5, more commonly known as ES5, is a more modern iteration of JavaScript was approved in 2011 and added a ton of useful functions and APIs that hadn't previously existed.

At the time of writing, ES6 is the latest iteration, ES7 (now called ES2016) is in the works, and new versions are coming every year. ES6 and beyond lack strong backwards browser compatibility, however, and the resulting code looks very different from traditional JavaScript.

As a result, we'll be focusing on ES5 functions and APIs, which are incredibly well supported and provide a dependency-free development experience..

Selectors

Get elements in the DOM.

querySelectorAll()

Use `document.querySelectorAll()` to find all matching elements on a page. You can use any valid CSS selector.

```
// Get all elements with the .bg-red class
var elemsRed = document.querySelectorAll( '.bg-red' );

// Get all elements with the [data-snack] attribute
var elemsSnacks = document.querySelectorAll( '[data-snack]' );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. Can also be used in IE8 with CSS2.1 selectors (no CSS3 support).

querySelector()

Use `document.querySelector()` to find the first matching element on a page.

```
// The first div
var elem = document.querySelector( 'div' );

// The first div with the .bg-red class
var elemRed = document.querySelector( '.bg-red' );

// The first div with a data attribute of snack equal to carrots
var elemCarrots = document.querySelector( '[data-snack="carrots"]' );

// An element that doesn't exist
var elemNone = document.querySelector( '.bg-orange' );
```

If an element isn't found, `querySelector()` returns `null`. If you try to do something with the nonexistant element, an error will get thrown. You should check that a matching element was found before using it.

```
// Verify element exists before doing anything with it
if ( elemNone ) {
    // Do something...
}
```

If you find yourself doing this a lot, here's a helper method you can use that will fail gracefully by returning a dummy element rather than cause an error.

```
var getElem = function ( selector ) {
    return document.querySelector( selector ) || document.createElement( '_' );
};
getElem( '.does-not-exist' ).id = 'why-bother';
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. Can also be used in IE8 with CSS2.1 selectors (no CSS3 support).

getElementById()

Use `getElementById()` to get an element by its ID.

```
var elem = getElementById( '#some-selector' );
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

getElementsByClassName()

Use `getElementsByClassName()` to find all elements on a page that have a specific class or classes.

Note: *This returns a live HTMLCollection of elements. If an element is added or removed from the DOM after you set your variable, the list is automatically updated to reflect the current DOM.*

```
// Get elements with a class
var elemsByClass = document.getElementsByClassName( 'some-class' );

// Get elements that have multiple classes
var elemsWithMultipleClasses = document.getElementsByClassName( 'some-class another-cl
ass' );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

getElementsByTagName()

Use `getElementsByTagName()` to get all elements that have a specific tag name.

Note: *This returns a live HTMLCollection of elements. If an element is added or removed from the DOM after you set your variable, the list is automatically updated to reflect the current DOM.*

```
// Get all divs
var divs = document.getElementsByTagName( 'div' );

// Get all links
var links = document.getElementsByTagName( 'a' );
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

matches()

Use `matches()` to check if an element would be selected by a particular selector or set of selectors. Returns `true` if the element is a match, and `false` when it's not. This function is analogous to jQuery's `.is()` method.

```
var elem = document.querySelector( '#some-elem' );
if ( elem.matches( '.some-class' ) ) {
    console.log( 'It matches!' );
} else {
    console.log( 'Not a match... =(' );
}
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

But... several browser makes implemented it with nonstandard, prefixed naming. If you want to use it, you should include this polyfill to ensure consistent behavior across browsers.

```
if (!Element.prototype.matches) {
    Element.prototype.matches =
        Element.prototype.matchesSelector ||
        Element.prototype.mozMatchesSelector ||
        Element.prototype.msMatchesSelector ||
        Element.prototype.oMatchesSelector ||
        Element.prototype.webkitMatchesSelector ||
        function(s) {
            var matches = (this.document || this.ownerDocument).querySelectorAll(s),
                i = matches.length;
            while (--i >= 0 && matches.item(i) !== this) {}
            return i > -1;
        };
}
```

A word about selector performance

Selectors that target a specific element type, like `getElementById()` and `getElementsByClassName()`, are more than twice as fast as `querySelector()` and `querySelectorAll()`.

So, that's bad, right? I honestly don't think it matters.

`getElementById()` can run about 15 million operations a second, compared to *just* 7 million per second for `querySelector()` in the latest version of Chrome. But that also means that `querySelector()` runs 7,000 operations a millisecond. A millisecond. Let that sink in.

That's absurdly fast. Period.

Yes, `getElementById()` and `getElementsByClassName()` are faster. But the flexibility and consistency of `querySelector()` and `querySelectorAll()` make them the obvious muscle-memory choice for my projects.

They're not slow. They're just not *as* fast.

Loops

Loop through arrays, objects, and node lists.

Arrays and Node Lists

In vanilla JavaScript, you can use `for` to loop through array and node list items.

```
var sandwiches = [
    'tuna',
    'ham',
    'turkey',
    'pb&j'
];

for ( var i = 0; i < sandwiches.length; i++ ) {
    console.log(i) // index
    console.log( sandwiches[i] ) // value
}

// returns 0, tuna, 1, ham, 2, turkey, 3, pb&j
```

- In the first part of the loop, before the first semicolon, we set a counter variable (typically `i`, but it can be anything) to `0`.
- The second part, between the two semicolons, is the test we check against after each iteration of the loop. In this case, we want to make sure the counter value is less than the total number of items in our array. We do this by checking the `.length` of our array.
- Finally, after the second semicolon, we specify what to run after each loop. In this case, we're adding `1` to the value of `i` with `i++`.

We can then use `i` to grab the current item in the loop from our array.

Multiple loops on a page

Variables you set in the `for` part of the loop are not scoped to the loop, so if you tried to include a second loop with `var i` you would get an error. You can use a different variable, or define `var i` outside of the loop and set it's value in the loop.

```
for ( var n = 0; n < sandwiches.length; n++ ) {
    // Do stuff...
}

// Or...
var i;
for ( i = 0; i < sandwiches.length; i++ ) {
    // Do stuff...
}
```

Skip and End

You can skip to the next item in a loop using `continue`, or end the loop altogether with `break`.

```
for ( var n = 0; n < sandwiches.length; n++ ) {

    // Skip to the next in the loop
    if ( sandwiches[n] === 'ham' ) continue;

    // End the loop
    if ( sandwiches[n] === 'turkey' ) break;

    console.log( sandwiches[n] );

}
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

Objects

You can also use a `for` loop for objects, though the structure is just a little different. The first part, `key`, is a variable that gets assigned to the object key on each loop. The second part (in this case, `lunch`), is the object to loop over.

We also want to check that the property belongs to this object, and isn't inherited from further up the object chain (for nested or *deep* objects).

```
var lunch = {
    sandwich: 'ham',
    snack: 'chips',
    drink: 'soda',
    desert: 'cookie',
    guests: 3,
    alcohol: false,
};

for ( var key in lunch ) {
    if ( Object.prototype.hasOwnProperty.call( lunch, key ) ) {
        console.log( key ); // key
        console.log( lunch[key] ); // value
    }
}

// returns sandwich, ham, snack, chips, drink, soda, desert, cookie, guests, 3, alcoho
l, false
```

Browser Compatibility

Supported in all modern browsers, and IE6 and above.

forEach Helper Method

If you use loops a lot, you may want to use Todd Motto's helpful `forEach()` method.

It checks to see whether you've passed in an array or object and uses the correct `for` loop automatically. It also gives you a more jQuery-like syntax.

```
/*! foreach.js v1.1.0 | (c) 2014 @toddmotto | https://github.com/toddmotto/foreach */
var forEach = function (collection, callback, scope) {
    if (Object.prototype.toString.call(collection) === '[object Object]') {
        for (var prop in collection) {
            if (Object.prototype.hasOwnProperty.call(collection, prop)) {
                callback.call(scope, collection[prop], prop, collection);
            }
        }
    } else {
        for (var i = 0, len = collection.length; i < len; i++) {
            callback.call(scope, collection[i], i, collection);
        }
    }
};

// Arrays
forEach(sandwiches, function (sandwich, index) {
    console.log( sandwich );
    console.log( index );
});

// Objects
forEach(lunch, function (item, key) {

    // Skips to the next item.
    // No way to terminate the loop
    if ( item === 'soda' ) return;

    console.log( item );
    console.log( key );
});
```

It's worth mentioning that because the helper uses a function, you can only skip to the next item using `return`. There's no way to terminate the loop entirely.

Browser Compatibility

Supported in all modern browsers, and IE6 and above.

String Transforms

Helpful functions for modifying strings.

trim()

`.trim()` is used to remove whitespace from the beginning and end of a string.

```
var text = '  This sentence has some whitespace at the beginning and end of it.  ';
var trimmed = text.trim();
// returns 'This sentence has some whitespace at the beginning and end of it.'
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. The following polyfill can be used to push support back to IE6.

```
if (!String.prototype.trim) {
    String.prototype.trim = function () {
        return this.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, '');
    };
}
```

toLowerCase()

Transform all text in a string to lowercase.

```
var text = 'This sentence has some MIXED CASE LeTTeRs in it.';
var lower = text.toLowerCase();
// returns 'this sentence has some mixed case letters in it.'
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

toUpperCase()

Transform all text in a string to uppercase.

```
var text = 'This sentence has some MIXED CASE LeTTeRs in it.';
var upper = text.toUpperCase();
// returns 'THIS SENTENCE HAS SOME MIXED CASE LETTERS IN IT.'
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

parseInt()

Convert a string into an integer (a whole number). The second argument, `10`, is called the `radix`. This is the base number used in mathematical systems. For our use, it should always be `10`.

```
var text = '42px';
var integer = parseInt( text, 10 );
// returns 42
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

parseFloat()

Convert a string into a point number (a number with decimal points).

```
var text = '3.14someRandomStuff';
var pointNum = parseFloat( text );
// returns 3.14
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

replace()

Replace a portion of text in a string with something else.

```
var text = 'I love Cape Cod potato chips!';
var lays = text.replace( 'Cape Cod', 'Lays' );
var soda = text.replace( 'Cape Cod potato chips', 'soda' );
var extend = text.replace( 'Cape Cod', 'Cape Cod salt and vinegar' );

// lays: 'I love Lays potato chips!'
// soda: 'I love soda!'
// extend: 'I love Cape Cod salt and vinegar potato chips!'
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

slice()

Get a portion of a string starting (and optionally ending) at a particular character.

The first argument is where to start. Use `0` to include the first character. The second argument is where to end (and is optional).

If either argument is a negative integer, it will start at the end of the string and work backwards.

```
var text = 'Cape Cod potato chips';
var startAtFive = text.slice( 5 );
var startAndEnd = text.slice( 5, 8 );
var sliceFromTheEnd = text.slice( 0, -6 );

// startAtFive: 'Cod potato chips'
// startAndEnd: 'Code'
// sliceFromTheEnd: 'Cape Cod potato'
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

split()

Convert a string into an array by splitting it after a specific character (or characters).

The first argument, the `delimiter`, the character or characters to split by. As an optional second argument, you can stop splitting your string after a certain number of delimiter matches have been found.

```
var text = 'Soda, turkey sandwiches, potato chips, chocolate chip cookies' ;
var menu = text.split( ', ' );
var limitedMenu = text.split( ', ', 2 );

// menu: ["Soda", "turkey sandwiches", "potato chips", "chocolate chip cookies"]
// limitedMenu: ["Soda", "turkey sandwiches"]
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

Classes

Add, remove, toggle, and check for classes on an element.

Vanilla JavaScript provides the `classList` API, which works very similar to jQuery's class manipulation APIs.

All modern browsers support it, but IE9 does not. And new versions of IE don't implement all of its features. A small polyfill from Eli Grey provides IE9+ support if needed. [1]

classList

`classList` is modelled after jQuery's class manipulation APIs.

```
var elem = document.querySelector( '#some-elem' );

// Add a class
elem.classList.add( 'some-class' );

// Remove a class
elem.classList.remove( 'some-other-class' );

// Toggle a class
// (Add the class if it's not already on the element, remove it if it is.)
elem.classList.toggle( 'toggle-me' );

// Check if an element has a specfic class
if ( elem.classList.contains( 'yet-another-class' ) ) {
    // Do something...
}
```

Browser Compatibility

Works in all modern browsers, and IE10 and above. A polyfill from Eli Grey[2] extends support back to IE8.

className

You can use `className` to get all of the classes on an element as a string, add a class or classes, or completely replace or remove all classes.

```
var elem = document.querySelector( 'div' );

// Get all of the classes on an element
var elemClasses = elem.className;

// Add a class to an element
elem.className += ' vanilla-js';

// Completely replace all classes on an element
elem.className = 'new-class';
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

Styles

Get and set styles (ie. CSS) for an element.

Vanilla JavaScript uses camel cased versions of the attributes you would use in CSS. The Mozilla Developer Network provides a comprehensive list of available attributes and their JavaScript counterparts.^[https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference]

Inline Styles

Use `.style` to get and set inline styles for an element.

```
var elem = document.querySelector( '#some-elem' );

// Get a style
// If this style is not set as an inline style directly on the element, it returns an empty string
var bgColor = elem.style.backgroundColor;

// Set a style
elem.style.backgroundColor = 'purple';
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

Computed Styles

Use `window.getComputedStyle()` gets the actual computed style of an element. This factors in browser default stylesheets as well as external styles you've specified.

```
var elem = document.querySelector( '#some-elem' );
var bgColor = window.getComputedStyle( elem ).backgroundColor;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Attributes

Get and set attributes for an element.

getAttribute(), setAttribute(), and hasAttribute()

The `getAttribute()`, `setAttribute()`, and `hasAttribute()` methods let you get, set, and check for the existance of attributes (including data attributes) on an element.

```
var elem = document.querySelector( '#some-elem' );

// Get the value of an attribute
var sandwich = elem.getAttribute( 'data-sandwich' );

// Set an attribute value
elem.setAttribute( 'data-sandwich', 'turkey' );

// Check if an element has an attribute
if ( elem.hasAttribute( 'data-sandwich' ) ) {
    // do something...
}
```

These methods can also be used to manipulate other types of attributes (things like `id`, `tabindex`, `name`, and so on), but these are better done by calling the attribute on the element directly (see below).

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

Attribute Properties

You can get and set attributes directly on an element. View a full list of HTML attributes on the Mozilla Developer Network.^[https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes]

```
var elem = document.querySelector( '#some-elem' );

// Get attributes
var id = elem.id;
var name = elem.name;
var tabindex = elem.tabindex;

// Set attributes
elem.id = 'new-id';
elem.title = 'The title for this thing is awesome!';
elem.tabIndex = '-1';
```

Browser Compatibility

Supported in all modern browsers, and at least back to IE6.

Event Listeners

Use `addEventListener` to listen for events on an element. You can find a full list of available events on the Mozilla Developer Network. ^[https://developer.mozilla.org/en-US/docs/Web/Events]

```
var btn = document.querySelector( '#click-me' );
btn.addEventListener('click', function ( event ) {
    console.log( event ); // The event details
    console.log( event.target ); // The clicked element
}, false);
```

Multiple Targets

The vanilla JavaScript `addEventListener()` function requires you to pass in a specific, individual element to listen to. You cannot pass in an array or node list of matching elements like you might in jQuery or other frameworks.

To add the same event listener to multiple elements, you also **cannot** just use a `for` loop because of how the `i` variable is scoped (as in, it's not and changes with each loop).

Fortunately, there's a *really* easy way to get a jQuery-like experience: event bubbling.

Instead of listening to specific elements, we'll instead listen for *all* clicks on a page, and then check to see if the clicked item has a matching selector.

```
// Listen for clicks on the entire window
window.addEventListener('click', function ( event ) {

    // If the clicked element has the `.click-me` class, it's a match!
    if ( event.target.classList.contains( 'click-me' ) ) {
        // Do something...
    }

}, false);
```

Multiple Events

In vanilla JavaScript, each event type requires it's own event listener. Unfortunately, you *can't* pass in multiple events to a single listener like you might in jQuery and other frameworks.

But... by using a named function and passing that into your event listener, you can avoid having to write the same code over and over again.

```
// Setup our function to run on various events
var someFunction = function ( event ) {
    // Do something...
};

// Add our event listeners
window.addEventListener( 'click', someFunction, false );
window.addEventListener( 'scroll', someFunction, false );
```

Event Debouncing

Events like `scroll` and `resize` can cause huge performance issues on certain browsers. Paul Irish explains:^[https://www.paulirish.com/2009/throttled-smartresize-jquery-event-handler/]

> If you've ever attached an event handler to the window's resize event, you have probably noticed that while Firefox fires the event slow and sensibly, IE and Webkit go totally spastic.

Debouncing is a way of forcing an event listener to wait a certain period of time before firing again. To use this approach, we'll setup a `timeout` element. This is used as a counter to tell us how long it's been since the event was last run.

When our event fires, if `timeout` has no value, we'll assign a `setTimeout` function that expires after 66ms and contains our the methods we want to run on the event.

If it's been less than 66ms from when the last event ran, nothing else will happen.

```
// Setup a timer
var timeout;

// Listen for resize events
window.addEventListener('resize', function ( event ) {
    console.log( 'no debounce' );

    // If timer is null, reset it to 66ms and run your functions.
    // Otherwise, wait until timer is cleared
    if ( !timeout ) {
        timeout = setTimeout(function() {

            // Reset timeout
            timeout = null;

            // Run our resize functions
            console.log( 'debounced' );

        }, 66);
    }
}, false);
```

Use Capture

The last argument in `addEventListener()` is `useCapture`, and it specifies whether or not you want to "capture" the event. For most event types, this should be set to `false`. But certain events, like `focus`, don't bubble.

Setting `useCapture` to `true` allows you to take advantage of event bubbling for events that otherwise don't support it.

```
// Listen for all focus events in the document
document.addEventListener('focus', function (event) {
    // Run functions whenever an element in the document comes into focus
}, true);
```

Browser Compatibility

`addEventListener` works in all modern browsers, and IE9 and above.

DOM Ready

Wait until the DOM is ready before running code.

Note: *If you're loading your scripts in the footer (which you should be for performance reasons), the* `ready()` *method isn't really needed. It's just a habit from the "load everything in the header" days.*

Vanilla JavaScript provides a native way to do this: the `DOMContentLoaded` event for `addEventListener`.

But... if the DOM is already loaded by the time you call your event listener, the event never happens and your function never runs.

Below is a super lightweight helper method that does the same thing as jQuery's `ready()` method. This helper method does two things:

1. Check to see if the document is already `interactive` or `complete`. If so, it runs your function immediately.
2. Otherwise, it adds a listener for the `DOMContentLoaded` event.

```
/**
 * Run event after DOM is ready
  * @param  {Function} fn Callback function
 */
var ready = function ( fn ) {

    // Sanity check
    if ( typeof fn !== 'function' ) return;

    // If document is already loaded, run method
    if ( document.readyState === 'interactive' || document.readyState === 'complete' )
 {
        return fn();
    }

    // Otherwise, wait until document is loaded
    document.addEventListener( 'DOMContentLoaded', fn, false );

};

// Example
ready(function() {
    // Do stuff...
});
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

HTML

Use `.innerHTML` to get and set HTML content.

```
var elem = document.querySelector( '#some-elem' );

// Get HTML content
var html = elem.innerHTML;

// Set HTML content
elem.innerHTML = 'We can dynamically change the HTML. We can even include HTML element
s like <a href="#">this link</a>.';

// Add HTML to the end of an element's existing content
elem.innerHTML += ' Add this after what is already there.';

// Add HTML to the beginning of an element's existing content
elem.innerHTML = 'We can add this to the beginning. ' + elem.innerHTML;

// You can inject entire elements into other ones, too
elem.innerHTML += '<p>A new paragraph</p>';
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. **IE9 Exception:** Tables and selects require IE10 and above.^[http://quirksmode.org/dom/html/]

DOM Injection

How to add and remove elements in the DOM.

Injecting an element into the DOM

Injecting an element into the DOM requires us to combine a few JavaScript methods.

1. Get the element you want to add our new element before or after.
2. Create our new element using the `createElement()` method.
3. Add content to our element with `innerHTML`.
4. Add any other attributes we want to our element (an ID, classes, etc.).
5. Insert the element using the `insertBefore()` method.

```
// Get the element you want to add your new element before or after
var target = document.querySelector( '#some-element' );

// Create the new element
// This can be any valid HTML element: p, article, span, etc...
var div = document.createElement( 'div' );

// Add content to the new element
div.innerHTML = 'Your content, markup, etc.';

// You could also add classes, IDs, and so on
// div is a fully manipulatable DOM Node

// Insert the element before our target element
// The first argument is our new element.
// The second argument is our target element.
target.parentNode.insertBefore( div, target );

// Insert the element after our target element
// Instead of passing in the target, you pass in target.nextSibling
target.parentNode.insertBefore( div, target.nextSibling );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. **IE9 Exception:** Adding content to tables and selects with `innerHTML` require IE10 and above.^[http://quirksmode.org/dom/html/]

Removing an element from the DOM

You can easily hide an element in the DOM by setting it's `style.display` to `none`.

```
var elem = document.querySelector('#some-element');
elem.style.display = 'none';
```

To *really* remove an element from the DOM, we can use the `removeChild()` method. This method is called against our target element's parent, which we can get with `parentNode`.

```
var elem = document.querySelector('#some-element');
elem.parentNode.removeChild( elem );
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

Forms

There are some helpful element properties that make working with forms in JavaScript a bit easier.

forms

Get all forms on a page.

```
var forms = document.forms;
```

Browser Compatibility

Works in all modern browsers, and at least back to IE6.

elements

Get all elements in a form.

```
var form = document.querySelector( 'form' );
var elements = form.elements;
```

Browser Compatibility

Works in all modern browsers, and at least back to IE6.

Form Element Attributes

You can check the value of form element attributes by calling them directly on the element.

```
var input = document.querySelector( '#input-1' );
var type = input.type;

var checkbox = document.querySelector( '#checkbox' );
var name = checkbox.name;

var radio = document.querySelector( '[name="radiogroup"]:checked' );
var value = radio.value;
```

You can also change and set attributes using the same approach.

```
var input = document.querySelector( '#input-1' );
input.type = 'email';

var checkbox = document.querySelector( '#checkbox' );
checkbox.name = 'thisForm';

var textarea = document.querySelector( '#textarea' );
textarea.value = 'Hello, world!';
```

Browser Compatibility

Works in all modern browsers, and at least back to IE6.

Boolean Element Attributes

Certain attributes, like whether or not an input is `disabled`, `readonly`, or `checked`, use simple `true` / `false` boolean values.

```
var input1 = document.querySelector( '#input-1' );
var isDisabled = input1.disabled;

var input2 = document.querySelector( '#input-2' );
input2.readOnly = false;
input2.required = true;

var checkbox = document.querySelector( '#checkbox' );
isChecked = checkbox.checked;

var radio = document.querySelector( '#radio-1' );
radio.checked = true;
```

Browser Compatibility

Works in all modern browsers, and at least back to IE6.

Traversing Up the DOM

parentNode

Use `parentNode` to get the parent of an element.

```
var elem = document.querySelector( '#some-elem' );
var parent = elem.parentNode;
```

You can also string them together to go several levels up.

```
var levelUpParent = elem.parentNode.parentNode;
```

Browser Compatibility

Works in all modern browsers, and at least back to IE6.

getClosest()

`getClosest()` is a helper method I wrote to get the closest parent up the DOM tree that matches against a selector. It's a vanilla JavaScript equivalent to jQuery's `.closest()` method.

```
/**
 * Get the closest matching element up the DOM tree.
 * @private
 * @param  {Element} elem     Starting element
 * @param  {String}  selector Selector to match against
 * @return {Boolean|Element}  Returns null if not match found
 */
var getClosest = function ( elem, selector ) {

    // Element.matches() polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
            Element.prototype.webkitMatchesSelector ||
            function(s) {
                var matches = (this.document || this.ownerDocument).querySelectorAll(s
),
                    i = matches.length;
                while (--i >= 0 && matches.item(i) !== this) {}
                return i > -1;
            };
    }

    // Get closest match
    for ( ; elem && elem !== document; elem = elem.parentNode ) {
        if ( elem.matches( selector ) ) return elem;
    }

    return null;

};

// Example
var elem = document.querySelector( '#some-elem' );
var closestSandwich = getClosest( elem, '[data-sandwich]' );
```

Note: *This method can also be used in event listeners to determine if the* `event.target` *is inside of a particular element or not (for example, did a click happen inside of a dropdown menu?).*

Browser Compatibility

Works in all modern browsers, and IE9 and above.

getParents()

`getParents()` is a helper method I wrote that returns an array of parent elements, optionally matching against a selector. It's a vanilla JavaScript equivalent to jQuery's `.parents()` method.

It starts with the element you've passed in itself, so pass in `elem.parentNode` to skip to the first parent element instead.

```
/**
 * Get all of an element's parent elements up the DOM tree
 * @param  {Node}   elem     The element
 * @param  {String} selector Selector to match against [optional]
 * @return {Array}           The parent elements
 */
var getParents = function ( elem, selector ) {

    // Element.matches() polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
            Element.prototype.webkitMatchesSelector ||
            function(s) {
                var matches = (this.document || this.ownerDocument).querySelectorAll(s
),
                    i = matches.length;
                while (--i >= 0 && matches.item(i) !== this) {}
                return i > -1;
            };
    }

    // Setup parents array
    var parents = [];

    // Get matching parent elements
    for ( ; elem && elem !== document; elem = elem.parentNode ) {

        // Add matching parents to array
        if ( selector ) {
            if ( elem.matches( selector ) ) {
                parents.push( elem );
            }
        } else {
            parents.push( elem );
        }

    }

    return parents;

};

// Example
var elem = document.querySelector( '#some-elem' );
var parents = getParents( elem.parentNode );
var parentsWithWrapper = getParents( elem.parentNode, '.wrapper' );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

getParentsUntil()

`getParentsUntil()` is a helper method I wrote that returns an array of parent elements until a matching parent is found, optionally matching against a selector. It's a vanilla JavaScript equivalent to jQuery's `.parentsUntil()` method.

It starts with the element you've passed in itself, so pass in `elem.parentNode` to skip to the first parent element instead.

```
/**
 * Get all of an element's parent elements up the DOM tree until a matching parent is
found
 * @param  {Node}   elem     The element
 * @param  {String} parent   The selector for the parent to stop at
 * @param  {String} selector The selector to filter against [optionals]
 * @return {Array}           The parent elements
 */
var getParentsUntil = function ( elem, parent, selector ) {

    // Element.matches() polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
            Element.prototype.webkitMatchesSelector ||
            function(s) {
                var matches = (this.document || this.ownerDocument).querySelectorAll(s
),
                    i = matches.length;
                while (--i >= 0 && matches.item(i) !== this) {}
                return i > -1;
            };
    }

    // Setup parents array
    var parents = [];

    // Get matching parent elements
    for ( ; elem && elem !== document; elem = elem.parentNode ) {

        if ( parent ) {
            if ( elem.matches( parent ) ) break;
        }
```

```
        if ( selector ) {
            if ( elem.matches( selector ) ) {
                parents.push( elem );
            }
            break;
        }

        parents.push( elem );

    }

    return parents;

};

// Examples
var elem = document.querySelector( '#some-element' );
var parentsUntil = getParentsUntil( elem, '.some-class' );
var parentsUntilByFilter = getParentsUntil( elem, '.some-class', '[data-something]' );
var allParentsUntil = getParentsUntil( elem );
var allParentsExcludingElem = getParentsUntil( elem.parentNode );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Traversing Down the DOM

querySelector() and querySelectorAll()

The `querySelector()` and `querySelectorAll()` APIs aren't limited to just running on the `document`. They can be run on any element to search only for elements inside of it.

```
var elem = document.querySelector( '#some-elem' );

// Find the first element inside `#some-elem` that has a `[data-snack]` attribute
var snack = elem.querySelector( '[data-snack]' );

// Get all divs inside `#some-elem`
var divs = elem.querySelectorAll( 'div' );
```

Browser Compatibility

Same as `querySelectorAll` and `querySelector`.

children

While `querySelector()` and `querySelectorAll()` search through all levels within a nested DOM/HTML structure, you may want to just get immediate decedants of a particular element. Use `children` for this.

```
var elem = document.querySelector( '#some-elem' );
var decendants = wrapper.children;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Traversing Sideways in the DOM

`getSiblings` is a helper method I wrote that gets the siblings of an element in the DOM. For example: if you had a list item (`<li>`) and wanted to grab all of the other items in the list.

```
/**
 * Get all siblings of an element
 * @param  {Node}  elem The element
 * @return {Array}      The siblings
 */
var getSiblings = function ( elem ) {
    var siblings = [];
    var sibling = elem.parentNode.firstChild;
    for ( ; sibling; sibling = sibling.nextSibling ) {
        if ( sibling.nodeType === 1 && sibling !== elem ) {
            siblings.push( sibling );
        }
    }
    return siblings;
};

// Example
var elem = document.querySelector( '#some-element' );
var siblings = getSiblings( elem );
```

Browser Compatibility

Works in all modern browsers, and IE6 and above.

Merging Arrays and Objects

Add items to an array

Use `push()` to add items to an array.

```
var sandwiches = ['turkey', 'tuna', 'blt'];
sandwiches.push('chicken', 'pb&j');
// sandwiches: ['turkey', 'tuna', 'blt', 'chicken', 'pb&j']
```

Browser Compatibility

Works in all modern browsers, and IE6 and above.

Merge two or more arrays together

Use `Array.prototype.push.apply()` to merge two or more arrays together. Merges all subsequent arrays into the first.

```
var sandwiches1 = ['turkey', 'tuna', 'blt'];
var sandwiches2 = ['chicken', 'pb&j'];
Array.prototype.push.apply(sandwiches1, sandwiches2);
// sandwiches1: ['turkey', 'tuna', 'blt', 'chicken', 'pb&j']
// sandwiches2: ['chicken', 'pb&j']
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

Add items to an object

Use the dot notation (`obj.something`) or bracket notation (`obj['something']`) to add key/value pairs to an object.

```
var lunch = {
    sandwich: 'turkey',
    chips: 'cape cod',
    drink: 'soda'
}

// Add items to the object
lunch.alcohol = false;
lunch["dessert"] = 'cookies';

// return: {sandwich: "turkey", chips: "cape cod", drink: "soda", alcohol: false, dess
ert: "cookies"}
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

Merge two or more objects together

`extend` is a helper method I wrote to merge two or more objects together. It works a lot like jQuery's `.extend()` function, except that it returns a new object, preserving all of the original objects and their properties. For deep (or recursive) merges, pass in `true` as the first argument. Otherwise, just pass in your objects.

```
/**
 * Merge two or more objects. Returns a new object.
 * Set the first argument to `true` for a deep or recursive merge
 * @param {Boolean}  deep     If true, do a deep (or recursive) merge [optional]
 * @param {Object}   objects  The objects to merge together
 * @returns {Object}          Merged values of defaults and options
 */
var extend = function () {

    // Variables
    var extended = {};
    var deep = false;
    var i = 0;
    var length = arguments.length;

    // Check if a deep merge
    if ( Object.prototype.toString.call( arguments[0] ) === '[object Boolean]' ) {
        deep = arguments[0];
        i++;
    }

    // Merge the object into the extended object
    var merge = function ( obj ) {
```

```
        for ( var prop in obj ) {
            if ( Object.prototype.hasOwnProperty.call( obj, prop ) ) {
                // If deep merge and property is an object, merge properties
                if ( deep && Object.prototype.toString.call(obj[prop]) === '[object Ob
ject]' ) {
                    extended[prop] = extend( true, extended[prop], obj[prop] );
                } else {
                    extended[prop] = obj[prop];
                }
            }
        }
    };

    // Loop through each object and conduct a merge
    for ( ; i < length; i++ ) {
        var obj = arguments[i];
        merge(obj);
    }

    return extended;

};

// Example objects
var object1 = {
    apple: 0,
    banana: { weight: 52, price: 100 },
    cherry: 97
};
var object2 = {
    banana: { price: 200 },
    durian: 100
};
var object3 = {
    apple: 'yum',
    pie: 3.214,
    applePie: true
}

// Create a new object by combining two or more objects
var newObjectShallow = extend( object1, object2, object3 );
var newObjectDeep = extend( true, object1, object2, object3 );
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

The Viewport

Get the viewport height

There are two methods to get the viewport height: `window.innerHeight` and `document.documentElement.clientHeight`. The former is more accurate. The latter has better browser support.

To get the best of both worlds, try `innerHeight` first, and fallback to `clientHeight` if not supported.

```
var viewportHeight = window.innerHeight || document.documentElement.clientHeight;
```

Browser Compatibility

`innerHeight` works in all modern browsers, and IE9 and above. `clientHeight` works in all modern browsers, and IE6 and above.

Get the viewport width

There are two methods to get the viewport width: `window.innerWidth` and `document.documentElement.clientWidth`. The former is more accurate. The latter has better browser support.

To get the best of both worlds, try `innerWidth` first, and fallback to `clientWidth` if not supported.

```
var viewportWidth = window.innerWidth || document.documentElement.clientWidth;
```

Browser Compatibility

`innerWidth` works in all modern browsers, and IE9 and above. `clientWidth` works in all modern browsers, and IE6 and above.

Check if an element is in the viewport or not

`isInViewport` is a helper method I wrote to check if an element is in the viewport or not. It returns `true` if the element is in the viewport, and `false` if it's not.

```
/**
 * Determine if an element is in the viewport
 * @param  {Node}    elem The element
 * @return {Boolean}      Returns true if element is in the viewport
 */
var isInViewport = function ( elem ) {
    var distance = elem.getBoundingClientRect();
    return (
        distance.top >= 0 &&
        distance.left >= 0 &&
        distance.bottom <= (window.innerHeight || document.documentElement.clientHeigh
t) &&
        distance.right <= (window.innerWidth || document.documentElement.clientWidth)
    );
};

// Example
var elem = document.querySelector( '#some-element' );
isInViewport( elem ); // Boolean: returns true/false
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Distances

Get the currently scrolled distance from the top of the page

Use `pageYOffset` to get the distance the user has scrolled from the top of the page.

```
var distance = window.pageYOffset;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Get an element's distance from the top of the page

`getOffsetTop` is a helper method I wrote to get an element's distance from the top of the document.

```
/**
 * Get an element's distance from the top of the Document.
 * @private
 * @param  {Node} elem The element
 * @return {Number}    Distance from the top in pixels
 */
var getOffsetTop = function ( elem ) {
    var location = 0;
    if (elem.offsetParent) {
        do {
            location += elem.offsetTop;
            elem = elem.offsetParent;
        } while (elem);
    }
    return location >= 0 ? location : 0;
};

// Example
var elem = document.querySelector( '#some-element' );
var distance = getOffsetTop( elem );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Query Strings

`getQueryString` is a helper method I wrote to get the value of a query string from a URL. Pass in the key to get the value of. You can optionally pass in a URL as a second argument. The function will use the window URL by default.

```
/**
 * Get the value of a query string from a URL
 * @param  {String} field The field to get the value of
 * @param  {String} url   The URL to get the value from [optional]
 * @return {String}       The value
 */
var getQueryString = function ( field, url ) {
    var href = url ? url : window.location.href;
    var reg = new RegExp( '[?&]' + field + '=([^&#]*)', 'i' );
    var string = reg.exec(href);
    return string ? string[1] : null;
};

// Examples
// http://example.com?sandwich=turkey&snack=cookies
var sandwich = getQueryString( 'sandwich' ); // returns 'turkey'
var snack = getQueryString( 'snack' ); // returns 'cookies'
var dessert = getQueryString( 'dessert' ); // returns null
var drink = getQueryString( 'drink', 'http://another-example.com?drink=soda' ); // ret
urns 'soda'
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

Ajax/HTTP Requests

Standard Ajax/HTTP Requests

Ajax requests have really good browser support, but require a few steps and can be a bit tedious to write. Later in this lesson, we'll look at a lightweight, super helpful micro-library. But first, let's look at how to make Ajax requests without any help.

1. We setup our request with `XMLHttpRequest()`.
2. We create an `onreadystatechange` callback to run when the request completes.
3. We open and send our request.

For testing purposes, we'll pull data from JSON Placeholder.^[https://jsonplaceholder.typicode.com/]

```
// Set up our HTTP request
var xhr = new XMLHttpRequest();

// Setup our listener to process compeleted requests
xhr.onreadystatechange = function () {
    // Only run if the request is complete
    if ( xhr.readyState !== 4 ) return;

    // Process our return data
    if ( xhr.status === 200 ) {
        // What do when the request is successful
        console.log( xhr );
    } else {
        // What do when the request fails
        console.log('The request failed!');
    }

    // Code that should run regardless of the request status
    console.log('This always runs...');
};

// Create and send a GET request
// The first argument is the post type
// The second argument is the endpoint URL
xhr.open( 'GET', 'https://jsonplaceholder.typicode.com/posts' );
xhr.send();
```

Browser Compatibility

Works in all modern browsers, and IE7 and above.

JSONP

For security reasons, you cannot load JSON files that reside on a different server. JSONP is a way to get around this issue. Sitepoint explains:^[https://www.sitepoint.com/jsonp-examples/]

> JSONP (which stands for JSON with Padding) builds on this technique and provides us with a way to access the returned data. It does this by having the server return JSON data wrapped in a function call (the “padding”) which can then be interpreted by the browser.

With JSONP, you need to use a global callback function to actually do something with the data you get. When you request your JSON data, you pass in that callback as an argument on the URL. When the data loads, it runs the callback function.

`getJSONP` is a helper function I wrote to handle JSONP requests. Pass in your URL endpoint as the first argument, and your callback function (as a string) as the second argument.

For testing purposes, we're using JSFiddle's Echo service^[http://doc.jsfiddle.net/use/echo.html] in the example below.

```
/**
 * Get JSONP data
 * @param  {String}   url      The JSON URL
 * @param  {Function} callback The function to run after JSONP data loaded
 */
var getJSONP = function ( url, callback ) {

    // Create script with url and callback (if specified)
    var ref = window.document.getElementsByTagName( 'script' )[ 0 ];
    var script = window.document.createElement( 'script' );
    script.src = url + (url.indexOf( '?' ) + 1 ? '&' : '?') + 'callback=' + callback;

    // Insert script tag into the DOM (append to <head>)
    ref.parentNode.insertBefore( script, ref );

    // After the script is loaded (and executed), remove it
    script.onload = function () {
        this.remove();
    };

};

// Example
var logAPI = function ( data ) {
    console.log( data );
}

getJSONP( 'http://jsfiddle.net/echo/jsonp/?text=something&par1=another&par2=one-more' ,
'logAPI' );
```

Browser Compatibility

Works in all modern browsers, and IE6 and above.

Atomic

Atomic^[https://github.com/toddmotto/atomic] is an insanely useful library from Todd Motto. It weighs just 1.5kb minified, and makes working with Ajax/HTTP and JSONP absurdly easy.

```
// After you've included Atomic with your scripts...

// GET
atomic.get('https://jsonplaceholder.typicode.com/posts')
    .success(function (data, xhr) {
        // What do when the request is successful
        console.log(data);
        console.log(xhr);
```

```
    })
    .error(function () {
        // What do when the request fails
        console.log( 'The request failed!' );
    })
    .always(function (data, xhr) {
        // Code that should run regardless of the request status
    });


// POST
atomic.post('http://jsonplaceholder.typicode.com/posts', {
    title: 'foo',
    body: 'bar',
    userId: 1
})
    .success(function (data, xhr) {
        // What do when the request is successful
        console.log(data);
        console.log(xhr);
    })
    .error(function () {
        // What do when the request fails
        console.log( 'The request failed!' );
    })
    .always(function (data, xhr) {
        // Code that should run regardless of the request status
    });


// JSONP

// Callback method
var logAPI = function ( data ) {
    console.log( data );
}

// Get JSONP
atomic.jsonp('http://jsfiddle.net/echo/jsonp/', 'logAPI', {
    text: 'something',
    par1: 'another',
    par2: 'one-more',
    bool: true
});
```

Browser Compatibility

Works in all modern browsers, and IE8 and above.

Cookies

How to get, set, and remove cookies.

Setting a cookie

You can use `document.cookie` to set a cookie.

```
document.cookie = 'sandwich=turkey; expires=Fri, 31 Dec 2024 23:59:59 GMT';
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

Getting a cookie value

Getting a cookie value involves parsing a string, and can be a bit awkward to work with. `getCookie()` ^[https://gist.github.com/wpsmith/6cf23551dd140fb72ae7] is a super lightweight helper method to make this easier.

```
var getCookie = function (name) {
    var value = "; " + document.cookie;
    var parts = value.split("; " + name + "=");
    if (parts.length == 2) return parts.pop().split(";").shift();
};

// Example
var cookieVal = getCookie( 'sandwich' ); // returns "turkey"

### Browser Compatibility

Works in all modern browsers, and at least IE6.


## More complex cookies

If you're doing more complex work with cookies, I would strongly recommend the simple
cookie library provided by MDN.^[[https://developer.mozilla.org/en-US/docs/Web/API/Doc
ument/cookie/Simple_document.cookie_framework](https://developer.mozilla.org/en-US/doc
s/Web/API/Document/cookie/Simple_document.cookie_framework)]. It let's you easily set,
 get, and remove cookies.

```javascript
// Set a cookie
docCookies.setItem( 'sandwich', 'turkey with tomato and mayo', new Date(2020, 5, 12) )
;

// Get a cookie
var cookieVal = docCookies.getItem('sandwich');

// Remove a coookie
docCookies.removeItem( 'sandwich' );
```

Browser Compatibility

Works in all modern browsers, and at least IE6.
```

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 section, 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.^[http://bradfrostweb.com/blog/mobile/support-vs-optimization/]

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.^[http://responsivenews.co.uk/post/18948466399/cutting-the-mustard]

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 && 'addEventListener' 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, or check for `classList` support. Without the polyfill, your IE support starts at IE10 instead of IE9.

```
var supports = 'querySelector' in document && 'addEventListener' in window && 'classLi
st' in document.createElement('_');
```

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.

Debugging

Your code 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.

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:^[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode]

> 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';
```

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, unfortunately.

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

In the previous lesson, I mentioned that when you load your JavaScript in the footer (as you should for performance reasons) the `ready()` method isn't neccessary.

It does actually serve a useful purpose, though: scoping.

It acts as a wrapper around your code, and as a result keeps your variables and functions out of the global scope, where they're more likely to conflict with other scripts.

Since you typically won't need to use the `ready()` method, you should wrap your scripts in what's called an IIFE, an Immediately Invoked Function Expression. An IIFE is simply an anonymous (unnamed) function that runs immediately.

```
;(function (window, document, undefined) {
    // Do stuff...
})(window, document);
```

Because your code is wrapped in a function, all of the variables and functions inside it are local to the IIFE. They won't override variables and functions set by other scripts, nor can they be accessed by other scripts.

And if you set a variable with the same name as one in the global scope the local one takes precendence.

```
// Default variables
var sandwich = 'tuna';
var chips = 'lays';

;(function (window, document, undefined) {

    // This changes the chips variable globally
    // Without `var`, it's modifying the global variable
    chips = 'cape cod';

   // This creates a new local variable that doesn't affect or get affected by the glo
bal scope
   var sandwich = 'turkey';

})(window, document);

console.log( sandwich );
// returns 'tuna'
// the `sandwich` variable is completely unaffected by why happens in the IIFE
```

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), and there are other ways to keep your code out of the global scope.

But generally speaking, for the types of scripts we will be covering in this book, you should always wrap your code in an IIFE.

Caveat: *If you use* `ready()` *, you don't need to use an IIFE. Your scripts are already scoped.*

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 APIs and methods 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.^[https://github.com/cferdinandi/right-height]

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.

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...
        i. Get the height of the tallest content area.
        ii. Set each content area to the height of the tallest one.
    - If they're one-on-top-of-the-other...
        i. 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...
        i. 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.
        ii. 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...
        i. 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

Let's put everything we've learned together with a small, working script.

When someone clicks a link with the `.click-me` class, we want to show the content in the element whose ID matches the anchor link.

```
<p>
    <a class="click-me" href="#hide-me">
        Click Me
    </a>
</p>

<div class="hide-me" id="hide-me">
    <p>Here's some content that I'd like to show or hide when the link is clicked. </p>
</div>
```

IIFE and Strict Mode

Let's start by creating an IIFE to keep our code out of the global scope. Let's also turn on strict mode so that we can aggressively debug.

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

    'use strict';

    // Code goes here...

})(window, document);
```

Core Functionality

We can use `addEventListener` to detect clicks on the `.click-me` button.

```
var link = document.querySelector( '.click-me' );
link.addEventListener('click', function (event) {
    // Do stuff when the link is clicked
}, false);
```

Putting It All Together

We can prevent the default click behavior with `event.preventDefault()`. This stops the link from changing the page URL.

```
var link = document.querySelector( '.click-me' );
link.addEventListener('click', function (event) {

    // Prevent the default click behavior
    event.preventDefault();

}, false);
```

To show and hide the content, we *could* toggle `display: none` using the `style` attribute, but I think using CSS in an external stylesheet is easier to maintain and gives you more flexibility. We'll add a style that hides our content areas by default, and shows them a second class is present.

```
.hide-me {
    display: none;
}

.hide-me.active {
    display: block;
}
```

That `.active` class can be anything you want. Some people prefer more action or state-oriented class names like `.is-open`. Do whatever fits your approach to CSS.

The nice thing with using a class to control visibility is that if you want a content area to be visible by default, you can just add the `.active` class to it as the starting state in your markup.

```
<div class="hide-me active" id="hide-me">
    <p>Here's some content that I'd like open by default, and hidden when the link is
clicked.</p>
</div>
```

Next, we'll toggle that class with JavaScript when the link is clicked using `classList`. The `classList.toggle()` method adds the class if it's missing, and removes it if it's present (just like jQuery's `.toggleClass()` method).

We'll use `event.target.hash` to get the hash from the clicked link, and `querySelector` to get the content it points to.

```
var link = document.querySelector( '.click-me' );
link.addEventListener('click', function (event) {

    // Prevent the default click behavior
    event.preventDefault();

    // Get the target content
    var content = document.querySelector( event.target.hash );

    // If the content area doesn't exist, bail
    if ( !content ) return;

    // Show or hide the content
    content.classList.toggle( 'active' );

}, false);
```

A few notes about the script above. First, we're using `.hash` instead of `.href`. Even with a relative URL like `#some-id`, `.href` will return the full URL for the page.

If an element is not found with `querySelector`, our script will throw an error if we try to run any methods against the `null` element. As a result, we want to make sure the element exists before doing anything with it.

What if there's more than one content area?

The code we've got so far is great when there's just a single expand-and-collapse area, but what if you multiple ones, like this?

```
<p>
    <a class="click-me" id="click-me-1" href="#hide-me-1">
        Click Me
    </a>
</p>
<div class="hide-me" id="hide-me-1">
    <p>Here's some content that I'd like to show or hide when the link is clicked. </p>
</div>

<p>
    <a class="click-me" id="click-me-2" href="#hide-me-2">
        Click Me, Too
    </a>
</p>
<div class="hide-me" id="hide-me-2">
    <p>Here's some more content that I'd like to show or hide when the link is clicked.
</p>
</div>
```

You *could* write add an event listener for every content area, but that's bloated and annoying to maintain.

```
/**
 * DON'T DO THIS!
 */
var link = document.querySelector( '#click-me-1' );
var link2 = document.querySelector( '#click-me-2' );

link.addEventListener('click', function (event) {

    // Prevent the default click behavior
    event.preventDefault();

    // Show or hide the content
    var content = document.querySelector( event.target.hash );
    if ( !content ) return;
    content.classList.toggle( 'active' );

}, false);

link2.addEventListener('click', function (event) {

    // Prevent the default click behavior
    event.preventDefault();

    // Show or hide the content
    var content = document.querySelector( event.target.hash );
    if ( !content ) return;
    content.classList.toggle( 'active' );

}, false);
```

Seriously, don't ever do that. Instead, let's take advantage of event bubbling.

Event Bubbling

Instead of identifying the individual elements to listen to, you can listen to all events within a parent element, and check to see if they were made on the elements you care about.

In our case, let's watch all clicks on the `document`, and then check to see if the clicked element has the `.click-me` class using the `.contains` method for `classList`. If it does, we'll run our script. Otherwise, we'll bail.

```
document.addEventListener('click', function (event) {

    // Make sure a .click-me link was clicked
    if ( !event.target.classList.contains( 'click-me' ) ) return;

    // Prevent default
    event.preventDefault();

    // Show or hide the content
    var content = document.querySelector( event.target.hash );
    if ( !content ) return;
    content.classList.toggle( 'active' );

}, false);
```

Congratulations! You now have a working expand-and-collapse script in vanilla JavaScript. And, it's more-or-less identical in size to the jQuery version.

Cutting the Mustard

We want to make sure the browser supports our JavaScript functions and browser APIs before attempting to run our code. Let's include a feature test before running our event listener.

We're checking for `querySelector`, `addEventListener`, and `classList`. I'm comfortable with IE10+ support for this, so I've decided to skip the polyfill for this project.

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

    'use strict';

    // Feature test
    var supports = 'querySelector' in document && 'addEventListener' in window && 'cla
ssList' in document.createElement('_');
    if ( !supports ) return;

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

        // Make sure a .click-me link was clicked
        if ( !event.target.classList.contains( 'click-me' ) ) return;

        // Prevent default
        event.preventDefault();

        // Show or hide the content
        var content = document.querySelector( event.target.hash );
        if ( !content ) return;
        content.classList.toggle( 'active' );

    }, false);

})(window, document);
```

Progressively Enhance

You may have visitors who use older browsers that don't support ES5 (like corporate users stuck on IE8). You may also have some visitors who have modern browsers but spotty connections that fail to download your JavaScript file (like a commuter on a train headed into a tunnel).

You could have a bug in another script that cause your script to break, too. JavaScript is incredibly fragile.

Whatever the reason, you don't want your visitors stuck without access to your content.

We should conditionally hide our content areas only after the script has loaded. To do that, we'll add a unique class to our `<html>` element when the script loads.

Putting It All Together

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

    'use strict';

    // Feature test
    var supports = 'querySelector' in document && 'addEventListener' in window && 'cla
ssList' in document.createElement('_');
    if ( !supports ) return;

    // Add a class when the script loads
    document.documentElement.classList.add( 'invisible-ink' );

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

        // Make sure a .click-me link was clicked
        if ( !event.target.classList.contains( 'click-me' ) ) return;

        // Prevent default
        event.preventDefault();

        // Show or hide the content
        var content = document.querySelector( event.target.hash );
        if ( !content ) return;
        content.classList.toggle( 'active' );

    }, false);

})(window, document);
```

Then we'll hook into that class in our CSS file to conditionally hide content.

```
.invisible-ink .hide-me {
    display: none;
}

.hide-me.active {
    display: block;
}
```

And now we have a lightweight, vanilla JavaScript expand-and-collapse widget that supports any browser that can access the web.

What now?

It's my hope that by this point, you're feeling a lot more comfortable writing vanilla JavaScript.

If you stick with it, soon vanilla JS will feel more natural than working with a framework. In the not too distant future, vanilla JS will feel like second nature, and you'll need to pull up the jQuery/React/[Your Favorite Framework Here] docs every time you need to work with it.

How to keep learning

Here are some thoughts on how to keep building your vanilla JavaScript skills:

1. Convert a few of your framework-dependant scripts into vanilla JS. It's often easier to get started when you're refactoring existing code.
2. Pick a random project from the list below and start coding. You'll mess up a lot and learn a ton.
3. Look for vanilla JS open source projects. Look at thow their authors structure their code. Contribute to them if you can or want to. Ask questions.
4. Browse the Q&A for `vanilla JS` on Stack Overflow.^[http://stackoverflow.com/search?q=vanilla+js]
5. Read the Mozilla Developer Network's JavaScript documentation.^[https://developer.mozilla.org/en-US/docs/Web/JavaScript] I would probably do this in conjunction with items #1 and #2.

Vanilla JS Project Ideas

Some ideas to get you started.

- **Toggle Tabs.** Click on a tab to show it's content and hide other tab content in that tab group.
- **Modals.** Click on a link or button to open a modal window. Click a close button (or anywhere outside of the modal) to close it.
- **Save Form Data.** Click a "Save this Form" button to save all of the data in a form. Reloading the page causes the form to repopulate with the already completed data. Alternatively, automatically save form data whenever the user types.
- **Password Visibility.** Click a button or a checkbox to make a password field visible. Click it again (or uncheck the box) to mask the password in the field again.

- **Same Height.** Set all content areas in a group to the same height. If they're stacked one-on-top-of-the-other, reset them to their default height.
- **Notifications.** Click a button to make a notification message appear on the page. Click an "X" in the notification to remove it. Let the notification text get set dynamically via some data on the button that toggles it.

How to find vanilla JS open source projects

I'm maintaining a growing list of third-party vanilla JavaScript plugins on the Ditching jQuery Resources page ^[https://github.com/cferdinandi/vanilla-js-plugins/]. That's a great place to start.

You can also just google, `{type of script you're looking for} vanilla js`. A lot of impressive plugins either don't require any libraries or frameworks, or offer both jQuery and vanilla JavaScript versions.

Don't feel like you have to go 100% vanilla JS immediately

You'll go crazy. I tried doing this. It's *really* hard.

Migrate your projects over time. Look for vanilla JavaScript alternatives when possible.

Never stop learning.