

A beginner's guide to vanilla JavaScript

— FIELD GUIDE —

Ditching jQuery

Chris Ferdinandi

Ditching jQuery

A beginner's guide to vanilla JavaScript.

By Chris Ferdinandi

Go Make Things, LLC

v2.1.0

Copyright 2017 Chris Ferdinandi and Go Make Things, LLC. All Rights Reserved.

Intro

A few years ago, I was a front end developer who was completely overwhelmed by JavaScript.

I'd search StackOverflow for solutions, copy/paste code, and just hope that it would work. Most of the time it wouldn't. When it did, I had no idea how or why. Does this sound familiar?

Eventually, I started to feel comfortable hacking together some jQuery, but I never felt like I really understood how it worked.

I decided to convert a few of my jQuery scripts into vanilla JavaScript. After a few dozen side projects and a lot of trial **and** error, it finally clicked.

I want to help you learn vanilla JavaScript without all of the painful false starts and roadblocks that I encountered.

Who this is for

Ditching jQuery is for...

- Developers who can hack together some jQuery but don't feel like they really know JavaScript.
- People who want to write better jQuery (learning vanilla JS will help you code better, period).
- 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.

What you'll learn

This book covers...

- Modern vanilla JavaScript equivalents of common jQuery APIs.
- 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.

Along the way, there will be a ton of **labs** where you'll practice what you just learned. At the end, we'll work together to create Invisible Ink, a JavaScript accordion plugin.

Getting Setup

All of the source code for the lessons in this book are available on GitHub [1](#).

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 2 ², but Atom (from GitHub) ³ 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 that added a ton of useful APIs that hadn't previously existed.

Many modern web and ECMAScript 5 APIs wouldn't exist at all if jQuery hadn't grown so popular and pushed demand for them. It's made working on the web much easier. Thanks jQuery!

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 and jQuery.

As a result, we'll be focusing on ES5 APIs, which are incredibly well supported and are the perfect place to start for someone transitioning from jQuery to vanilla JavaScript.

Selectors

jQuery

In jQuery, you get elements in the DOM using `$ ()`. This returns an array of matching elements.

To get the first matching element on a page, you would grab the first item in the array.

```
// All divs  
var $elems = $( 'div' );  
  
// The first div  
var $firstElem = $elems[0];
```

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="c
arrots"]' );

// 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 nonexistent 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...
}

```

querySelectorAll()

Use `document.querySelectorAll()` to find all matching elements on a page:

```
// 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  
]' );
```

Loops

jQuery

In jQuery, you can iterate of arrays, objects, and node lists with the `.each ()` method.

```
// Arrays
```

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

```
$.each(sandwiches, function( index, sandwich ) {  
    console.log( index ) // index  
    console.log( sandwich ) // value  
});
```

```
// Objects
```

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

```
$.each(lunch, function( key, item ) {  
    console.log( key ) // key  
    console.log( item ) // item  
});
```

Arrays and Node Lists

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

- 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.
- 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`.

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

```
var sandwiches = [  
  'tuna',  
  'ham',  
  'turkey',  
  'pb&j'  
];  
  
for ( var i = 0; i < sandwiches.length; i++ ) {  
  console.log(i) // index  
  console.log( sandwiches[i] ) // value  
}
```

Any variables you set in the `for` part of the loop are global in scope, 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` 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...  
}
```

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] );  
  
}
```

Objects

You can also use a `for` loop for objects, though the structure is just a little different. The first part, `item`, 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 item in lunch ) {
  if ( Object.prototype.hasOwnProperty.call( lunch, item ) ) {
    console.log( item ); // key
    console.log( lunch[item] ); // value
  }
}

```

forEach Helper Method

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

It checks to see if 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(coll
ection, prop)) {
                callback.call(scope, collection[prop], pr
op, collection);
            }
        }
    } else {
        for (var i = 0, len = collection.length; i < len;
i++) {
            callback.call(scope, collection[i], i, collec
tion);
        }
    }
};

```

// 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

```



```
// 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.

String Transforms

In this section, we're going to look at a handful of helpful APIs for modifying strings. Most of them have no jQuery equivalent.

trim()

`.trim()` is used to remove whitespace from the beginning and end of a string. This is the only API that has a jQuery equivalent.

```
var text = '  This sentence has some whitespace at the beginning and end of it.  ';
```

```
// jQuery
var $trimmed = $.trim( text );
```

```
// Vanilla JS
var trimmed = text.trim();
```

toLowerCase()

Transform all text in a string to lowercase.

```
var text = 'This sentence has some MIXED CASE LeTTeRs in  
it.';  
var lower = text.toLowerCase();
```

toUpperCase()

Transform all text in a string to uppercase.

```
var text = 'This sentence has some MIXED CASE LeTTeRs in  
it.';  
var upper = text.toUpperCase();
```

parseInt()

Convert a string into an integer (number). That 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 );
```

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' );
```

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 negative, 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 );
```

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 );
```

Classes

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. [4](#)

`classList.add()`

Add a class to an element.

```
// jQuery  
var $elem = $( '#a' );  
$elem.addClass( 'jquery' );  
  
// Vanilla JS  
var elem = document.querySelector( '#a' );  
elem.classList.add( 'vanilla-js' );
```

`classList.remove()`

Remove a class from an element.

```
// jQuery
var $elem = $( '#a' );
$elem.removeClass( 'bg-navy' );

// Vanilla JS
var elem = document.querySelector( '#c' );
elem.classList.remove( 'bg-red' );
```

classList.toggle()

Add a class if the element does not currently have that class. Remove it if it does.

```
// jQuery
var $elem = $( '#a' );
$elem.toggleClass( 'bg-navy' );

// Vanilla JS
var elem = document.querySelector( '#c' );
elem.classList.toggle( 'bg-green' );
```

classList.contains()

Check if an element contains a class.

```
// jQuery
var $elem = $( '#a' );
if ( $elem.hasClass( 'bg-navy' ) ) {
    // Do something...
}

// Vanilla JS
var elem = document.querySelector( '#a' );
if ( elem.classList.contains( 'bg-navy' ) ) {
    // Do something...
}
```

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.

```
// Get all of the classes on an element
var elem = document.querySelector( 'div' );

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

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


Styles

jQuery

jQuery uses the `css()` method to get and set styles on an element.

```
var $elem = $( '#a' );

// Get styles
var bgColor = $elem.css( 'background-color' );

// Set styles
$elem.css( 'background-color', '#000000' );
```

Get Styles

There are a couple different ways to get styles in vanilla JavaScript. The Mozilla Developer Network provides a comprehensive list of available attributes. ⁵

You can use `style.{css property name}`. This only works for inline styles, however. If you're trying to get a value specified in a stylesheet, you'll get `null` instead.

```
var elem = document.querySelector( '#a' );  
var bgColor = elem.style.backgroundColor;
```

`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( '#a' );  
var bgColor = window.getComputedStyle( elem ).backgroundC  
olor;
```

Set Styles

`style.{css property name}` can also be used to set an inline style on an element.

```
var elem = document.querySelector( '#b' );  
elem.style.backgroundColor = '#000000';
```

Attributes

jQuery

jQuery uses the `data()` method to get and set attributes on an element.

```
var $elem = $( '#a' );

// Get attribute
var dataSandwich = $elem.data( 'sandwich' );
var elemId = $elem.data( 'id' );

// Set attribute
$elem.data( 'sandwich', 'ham' );
$elem.data( 'id', 'abc' );
```

getAttribute()

The `getAttribute()` API let's you get attributes from an element.

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

// Get the [data-sandwich] attribute value
var dataSandwich = elem.getAttribute( 'data-sandwich' );
```

It can be used to get attributes like ID, title, and so on as well, but this is better done using the attribute directly. You can view a full list of HTML attributes on the Mozilla Developer Network. ⁶

```
// These do the same thing
var elemId1 = elem.getAttribute( 'id' );
var elemId2 = elem.id;
```

setAttribute()

Use `setAttribute()` to set attributes on an element.

```
var elem = document.querySelector( '#a' );
elem.setAttribute( 'data-sandwich', 'turkey' );
```

As with `getAttribute()`, you can use `setAttribute()` to set things like ID, title, tabindex and more, though this is better done using the attribute directly.

```
// These do the same thing  
elem.setAttribute( 'title', 'Nope, turkey!' );  
elem.title = 'Nope, turkey!';
```

hasAttribute()

Use the `hasAttribute()` API to check if an element has a specific attribute.

```
var elem = document.querySelector( '#a' );  
if ( elem.hasAttribute( 'data-snack' ) ) {  
    // Do something...  
}
```

Event Listeners

jQuery

jQuery uses the `on()` method to listen for events, and also supplies some event specific methods (like `.click()`).

```
var $elem = $( '#click-me' );

$elem.on( 'click', function( event ) {
    console.log( 'clicked' );
});

$elem.click(function( event ) {
    console.log( 'also clicked' );
});
```

addEventListener

Listen for events on an element. You can find a full list of available events on the Mozilla Developer Network. [7](#)

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

Multiple Targets

jQuery's `.on()` method automatically listens for clicks on any element that matches your selector. The vanilla JavaScript `addEventListener()` API requires you to specify specific, individual elements to listen to.

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

jQuery's `.on()` method also allows you to pass in multiple events to listen to. With vanilla JavaScript, you need to add an event listener for each event individually.

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:⁸

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  
}, false);
```

DOM Ready

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.

jQuery

jQuery uses the `.ready()` method to wait until the DOM is ready before running code.

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

Vanilla JavaScript

Vanilla JavaScript does provide a native API to handle this: the `DOMContentLoaded` event for `addEventListener`.

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

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

1. Checking 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...
});

```

If you test these, you'll see that the vanilla JavaScript version actually runs earlier than the jQuery one, even if you put the vanilla JS one second in your code.

HTML

jQuery

jQuery uses the `.html()` method to get and set HTML.

```
var $elem = $( '#a' );

// Get HTML
var html = $elem.html();

// Set HTML
$elem.html( 'We can dynamically change the HTML. We can even include HTML elements like <a href="#">this link</a>.' );
```

Get HTML

In vanilla JavaScript you can get HTML for an element using the `innerHTML` API.

```
var elem = document.querySelector( '#a' );
var html = elem.innerHTML;
```

Set HTML

You can also use the `innerHTML` API to set HTML.

```
var elem = document.querySelector( '#a' );  
elem.innerHTML = 'We can dynamically change the HTML. We  
can even include HTML elements like <a href="#">this link  
</a>.';
```


Climbing Up the DOM

parentNode

You can use `parentNode` to get the parent of an element. This is the same as jQuery's `.parent()` method.

```
// jQuery  
var $elem = $( '#d3' );  
var parent = $elem.parent();
```

```
// Vanilla JS  
var elem = document.querySelector( '#d3' );  
var parent = elem.parentNode;
```

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

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

getClosest()

jQuery's `.closest()` method provides you with a way to get the closest parent up the DOM tree that matches against a selector.

```

var $elem = $( '#d3' );
var closestSandwich = $elem.closest( '[data-sandwich]' );

```

While there's no native vanilla JavaScript API to handle this, a simple helper method achieves the same result.

```

/**
 * 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) !== th

```

```

is) {}

        return i > -1;
    };

}

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

return null;

};

// Example
var elem = document.querySelector( '#d3' );
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?).

getParents()

jQuery's `.parents()` method returns an array of parent elements, optionally matching against a selector.

```

var $elem = $( '#d3' );
var parents = $elem.parents();
var parentsWithWrapper = $elem.parents( '.wrapper' );

```

The `getParents()` helper method provide below does the same thing. It starts with the element 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.owne

```

```

rDocument).querySelectorAll(s),
        i = matches.length;
        while (--i >= 0 && matches.item(i) !== th
is) {}

        return i > -1;
    };
}

// Setup parents array
var parents = [];

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

    // 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( '#d3' );
var parents = getParents( elem.parentNode );
var parentsWithWrapper = getParents( elem.parentNode, '.wrapper' );
```

Climbing Down the DOM

querySelector() and querySelectorAll()

You can use jQuery's `.find()` method to find elements that match a selector within another element.

```
var $elem = $( '.wrapper' );  
var divs = $elem.find( 'div' );
```

The `querySelector()` and `querySelectorAll()` APIs aren't limited to just running on the document. They can be run on any element, and provide the same functionality as jQuery's `.find()`.

```
var elem = document.querySelector( '.wrapper' );  
var snack = elem.querySelector( '[data-snack]' );  
var divs = elem.querySelectorAll( 'div' );
```

ChildNodes

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.

jQuery's `.children()` method does this.

```
var $elem = $( '#d' );  
var descendants = $wrapper.children();
```

The native `childNodes` API does the same thing.

```
var elem = document.querySelector( '#d' );  
var descendants = wrapper.childNodes;
```


Forms

There are some helpful native JavaScript and browser APIs that make working with forms a bit easier.

forms

Get all forms on a page.

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

elements

Get all elements in a form.

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

Form Element Attributes

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

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

var checkbox = document.querySelector( '#c' );
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( '#a' );
input.type = 'email';

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

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

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( '#a' );  
var isDisabled = input1.disabled;
```

```
var input2 = document.querySelector( '#b' );  
input2.readOnly = false;  
input2.required = true;
```

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

```
var radio = document.querySelector( '#d1' );  
radio.checked = true;
```

Cross-Browser Compatibility

The web is for everyone, but support is not the same as optimization. ⁹

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. ¹⁰

A simple browser test determines whether or not a browser supports modern JavaScript APIs. If it does, it 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)

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:^{[11](#)}



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 occurring, 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 jQuery and vanilla JS `ready()` methods aren't necessary.

They do actually serve a useful purpose, though: scoping.

They act as a wrapper around your code, and as a result keep 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 precedence.

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.

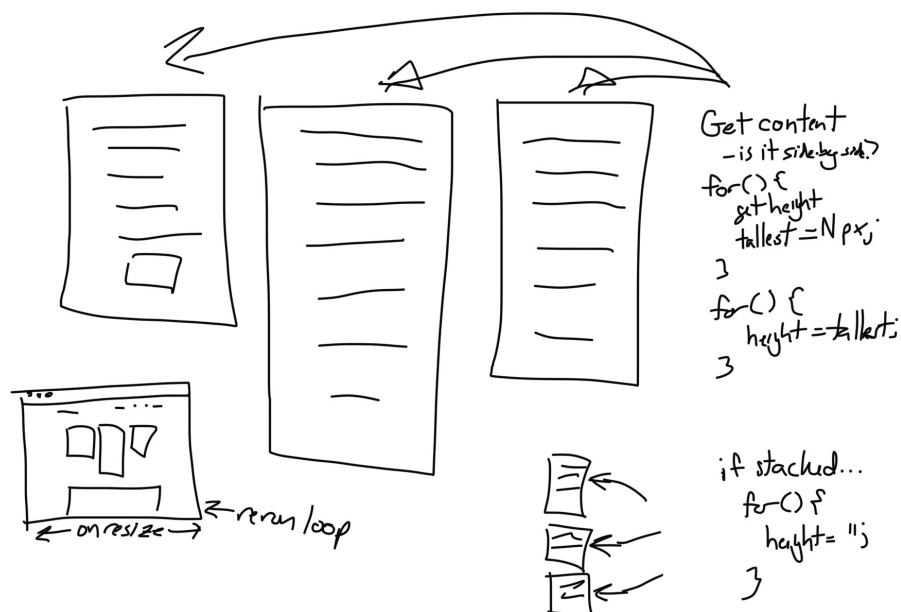
I find it helpful to identify what each of those steps should be ahead of time, which makes it easier for me to think through which APIs and methods I need to use, which elements on the page I need to target, and so on.

An example

Here's an actual example from when I created my Right Height plugin. [12](#)

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

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

Just like last time, let's add 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);
```

Just like in jQuery, 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);
```

In jQuery, you might use `toggle()` to toggle the content visibility. This method adds or removes `display: none` as an inline style on the element.

In vanilla JS, you *could* toggle `display: none` using the `style` attribute, but I think it's easier to maintain and gives you more flexibility to add these to an external stylesheet. 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 to show or hide  
when the link is clicked.</p>  
</div>
```

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

We use `event.target.hash` to get the hash from the clicked link.

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

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

    // Get the target content
    // We use .hash instead of .href because .href returns
    // the full URL, even for relative links
    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.

In jQuery, if an element isn't found, jQuery silently ignores it. `querySelector` does not, and we'll get an error if we try to run any methods against a `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 have multiple ones, like this?

```
<p>
  <a class="click-me" id="click-me-1" 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>

<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.ha
sh );
    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.ha
sh );
    if ( !content ) return;
    content.classList.toggle( 'active' );

```

```
}, false);
```

Seriously, don't ever, 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 basically identical in size to the jQuery version.

Cutting the Mustard

We want to make sure the browser supports modern JavaScript 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 && 'classList' 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);
```

```
    , , false , ,  
  
})(window, document);
```

Progressively Enhance

You may have visitors who use older browsers that don't support ES5 (think 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.

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.

```
;(function (window, document, undefined) {  
  
    'use strict';  
  
    // Feature test  
    var supports = 'querySelector' in document && 'addEventListener' in window && 'classList' in document.createElement('_');
```

```

—
if ( !supports ) return;

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

// 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.targe
t.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;  
}
```

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 to write than jQuery. In the not too distant future, vanilla JS will feel like second nature, and you'll need to pull up the jQuery 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 jQuery 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 how 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.^{[13](#)}
5. Read the Mozilla Developer Network's JavaScript documentation.^{[14](#)}

I would probably do this in conjunction with items #1 and #2.

Vanilla JS Project Ideas

Some ideas to get you started.

- **Toggle Tabs.** Clicking on a tab shows it's content and hides other tab content in that tab group.
- **Modals.** Clicking on a link or button opens a modal window. Clicking a close button (or outside of the modal) closes 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.
- **Password Visibility.** Clicking a button or a checkbox makes a password field visible. Clicking it again (or unchecking the box) masks 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, let me reset 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 ¹⁵. 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 jQuery, or offer both jQuery and
vanilla JS 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 JS alternatives when
possible.

Never stop learning.

About the Author



Hi, I'm Chris Ferdinandi. I help people learn vanilla 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](https://gomakethings.com).
- By email at chris@gomakethings.com.
- On Twitter at [@ChrisFerdinandi](https://twitter.com/ChrisFerdinandi).

-
1. <https://github.com/cferdinandi/ditching-jquery-source-code>↵
 2. <http://www.sublimetext.com/2>↵
 3. <https://atom.io>↵
 4. <https://github.com/eligrey/classList.js/>↵
 5. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference↵
 6. <https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes>↵
 7. <https://developer.mozilla.org/en-US/docs/Web/Events>↵
 8. <https://www.paulirish.com/2009/throttled-smartresize-jquery-event-handler/>↵
 9. <http://bradfrostweb.com/blog/mobile/support-vs-optimization/>↵
 10. <http://responsivenews.co.uk/post/18948466399/cutting-the-mustard>↵
 11. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode↵
 12. <https://github.com/cferdinandi/right-height>↵
 13. <http://stackoverflow.com/search?q=vanilla+js>↵
 14. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>↵
 15. <https://ditchingjquery.com/resources/>↵