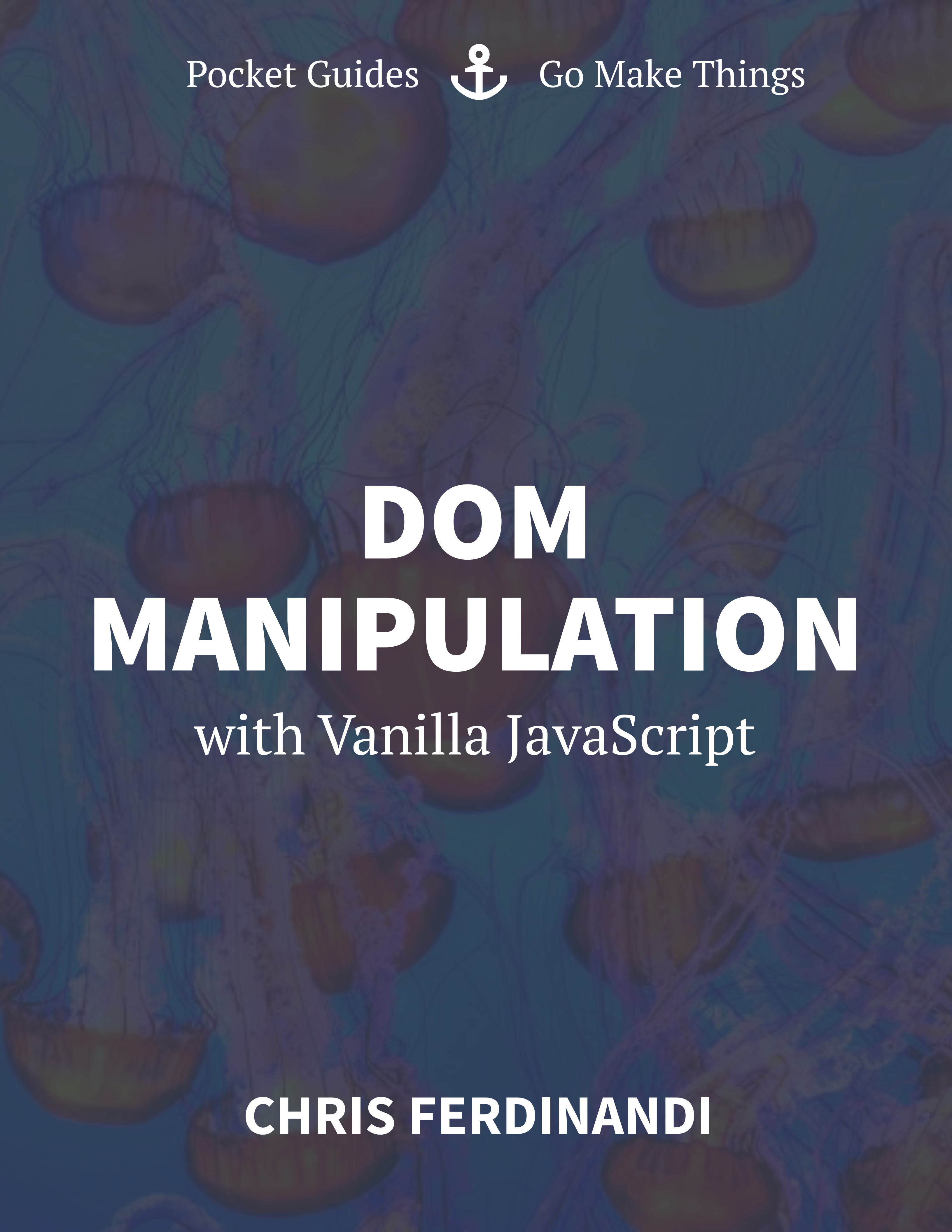

Pocket Guides Go Make Things

DOM MANIPULATION

with Vanilla JavaScript

CHRIS FERDINANDI

DOM Manipulation

By Chris Ferdinandi

Go Make Things, LLC

v2.0.1

Table of Contents

Intro

In this guide, you'll learn:

- How to get elements in the DOM.
- How to loop through arrays and objects.
- How to get, set, and remove classes.
- How to manipulate styles.
- How to get and set attributes.
- How to listen for events in the DOM.

A quick word about browser compatibility

This guide makes heavy use of ECMAScript 5 (more commonly known as ES5) methods and APIs.

That generally means browser support begins with IE9 and above. Each function or technique mentioned in this guide includes specific browser support information, as some do provide further backwards compatibility.

Let's get started...

Selectors

How to 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="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 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.getElementsByClas
sName( 'some-class another-class' );
```

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[1] 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[2] 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

How to loop through arrays, objects, and node lists.

Arrays and Node Lists

In vanilla JavaScript, you 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, dese
rt, cookie, guests, 3, alcohol, 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[3].

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://git
hub.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
    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.

Classes

How to add, remove, toggle, and check for classes on an element.

classList

The `classList` API works very similar to jQuery's class manipulation functions.

```
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, rem
ove 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[4] 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

How to get and set styles (as in, 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.[5]

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 o
n 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 ).backgroundC
olor;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Attributes

How to get, set, and remove attributes for an element.

getAttribute(), setAttribute(), removeAttribute(), and hasAttribute().

The `getAttribute()`, `setAttribute()`, `removeAttribute()`, and `hasAttribute()` methods let you get, set, remove, 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' );

// Remove an attribute
elem.removeAttribute( 'data-sandwich' );

// 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.[6]

```
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';

// Remove attributes
elem.id = '';
elem.title = '';
elem.name = '';
```

Browser Compatibility

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

Event Listeners

How to listen for browser events and run callback functions when they happen.

addEventListener()

Use `addEventListener` to 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 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:[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 fu
nctions.
    // 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.

Putting it all together

To make this all tangible, let’s work on a project together. We’ll build a simple accordion script we can use to show and hide the visibility of content.

The starter template and complete project code are included in the source code[9] on GitHub.

Getting Setup

I’ve dropped some placeholder markup into the template to help you get started.

Each content area is shown or hidden using a simple anchor link. Each link has an `.accordion-toggle` class on it that we can use to target the links for styling and with JavaScript. The link’s `href` points to the target content area.

Similarly, each content area is wrapped in a `<div>` with the `.accordion-content` class, and has a unique ID that matches the `href` of the toggle that shows or hides it.

```html
<a class="accordion-toggle" href="#content-1">Show More 1
</a>

<div class="accordion-content" id="content-1">
    Content goes here...
</div>
```

Showing and hiding our content

To make things really easy and gives us a ton of flexibility, we'll use some simple CSS to hide and show our content.

By default, we'll hide elements with the `.accordion-content` class using `display: none`. When someone clicks one of our toggle links, we'll add the `.active` class to the content, overriding it with `display: block` to make it visible.

Note: *There are a ton of different naming conventions you can use for classes like this. Some prefer more state-driven classes like `.is-open` or `.is-active`. Use whatever works best for you.*

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

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

An advantage of this approach is that if you have content you want to be open by default, you can add the `.active` class to it directly in the markup.

```
<div class="accordion-content active" id="content-1">
    This content is visible on page load...
</div>
```

Toggling content visibility

To toggle the visibility of our content, we need to detect when our `.accordion-toggle` links are clicked.

We *could* add an event listener to each link, but for maximum flexibility and performance, let's listen for all click events on the `document`, and filter out any clicked elements that don't have the `.accordion-toggle` class.

To do this, we'll use `classList.contains()` to check the `event.target` in our event listener.

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

    // Log the clicked element in the console
    console.log(event.target);

    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if (!event.target.classList.contains('accordion-toggl
e')) return;

    // Log the clicked element in the console
    console.log('matches');

});
```

The code above logs every clicked element in the console, but only displays `matches` when an accordion toggle is clicked. Open up the console tab in developer tools to see it in action.

Next, we want to get the element that our accordion toggle points to. The `event.target.hash` gives us the hash value of our anchor link, which matches the ID of our target content area. We can use that with `querySelector` to get our content. If no matching content is found, we'll end our function.

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

    // Bail if our clicked element doesn't have the .accordion-toggle class
    if (!event.target.classList.contains('accordion-toggle')) return;

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

});
```

Finally, let’s change the visibility of our content. If it already has the `.active` class on it, we want to remove it. And if it doesn’t have the class, we want to add it.

For that, we’ll use `classList.toggle()`.

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

    // Bail if our clicked element doesn't have the .accordion-toggle class
    if (!event.target.classList.contains('accordion-toggle')) return;

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

    // Toggle our content
    content.classList.toggle('active');

});
```

And with that little bit of code, we can now show and hide content when our toggle link is clicked.

Prevent the page from jumping

Because we’re using anchor links, every time a visitor clicks a toggle, it updates the URL and causes the page to jump to the anchored element.

If your monitor is tall enough, you might not notice it because the content in our demo project is pretty short. On real webpages, this would create a jarring visual jump.

We can prevent the URL update from happening (and the subsequent page jump) by using `event.preventDefault()` to prevent the default link behavior. We want to do this after we make sure the link is an accordion toggle and that matching content exists.

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

    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if (!event.target.classList.contains('accordion-toggl
e')) return;

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

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

    // Toggle our content
    content.classList.toggle('active');

});
```

Adding accordion functionality

At the moment, we have a collection of show-and-hide links.

To really make this an accordion, we only want one content are to be open at a time. We need to find any open accordion content areas and close them.

We’ll use `querySelectorAll()` to get any elements with both the `.accordion-content` and `.active` classes. Then, we’ll loop through them and remove the `.active` class with `classList.remove()`.

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

    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if (!event.target.classList.contains('accordion-toggl
e')) return;

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

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

    // Get all open accordion content, loop through it, a
nd close it
    var accordions = document.querySelectorAll('.accordio
n-content.active');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }

    // Toggle our content
    content.classList.toggle('active');

});
```

Now, only one content area can be open a time.

There’s a small problem with our code, though. If you click a toggle for an open content area, you might want to close that content area. With our current code, that area remains open.

In our `for` loop, we’re removing the existing active class on that content, and just adding it back again with `classList.toggle()`.

Instead, let’s first check to see if our content is open using the `classList.contains()` method. If it’s open, we’ll close it by removing the `.active` class and end our function. Otherwise, we’ll close all currently open content and open it by using `classList.add()` instead of `classList.toggle()`.

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

    // Bail if our clicked element doesn't have the .accordion-toggle class
    if (!event.target.classList.contains('accordion-toggle')) return;

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

    // Prevent default link behavior
    event.preventDefault();
    // If the content is already expanded, collapse it and quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        return;
    }

    // Get all open accordion content, loop through it, and close it
    var accordions = document.querySelectorAll('.accordion-content.active');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }

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

});
```

Congratulations! You just created a hide-and-show/accordion script using a variety of DOM manipulation techniques.

About the Author

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

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

You can find me:

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

1. https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill↩
2. https://jsperf.com/getelementbyid-vs-queryselector/25↩
3. https://github.com/toddmotto/foreach↩
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. https://github.com/cferdinandi/dom-manipulation-source-code/↩