Pocket Guides  Go Make Things

1

DOM MANIPULATION

with Vanilla JavaScript

CHRIS FERDINANDI

DOM Manipulation

By Chris Ferdinandi
Go Make Things, LLC
v2.2.2

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

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) and ECMA 6 (ES6) methods and APIs.

My goal for browser support is IE9 and above. Each function or technique mentioned in this guide includes specific browser support information. For methods and APIs that don't meet that standard, I also include information about polyfills—snippets of code that add support for features to browsers that don't natively offer it.

You'll never have to run a command line prompt, compile code, or learn a weird pseudo language (though you certain can if you want to).

Note: *You can extend support all the way back to IE7 with a polyfill service like polyfill.io*[1].

Using the code in this guide

Unless otherwise noted, all of the code in this book is free to use under the MIT license. You can view of copy of the license at https://gomakethings.com/mit.

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="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 nonexistent element, an error will get thrown. You should check that a matching element was found before using it.

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

Browser Compatibility

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

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('#sandwich
');
if (elem.matches('.turkey')) {
    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[2] to ensure consistent behavior across browsers.

```
/**
 * Element.matches() polyfill (simple versio
n)
 * https://developer.mozilla.org/en-US/docs/
Web/API/Element/matches#Polyfill
 */
if (!Element.prototype.matches) {
    Element.prototype.matches = Element.prot
otype.msMatchesSelector || Element.prototype
.webkitMatchesSelector;
}
```

Type-specific selector methods

There are other selector methods that target elements by specific type.

The `getElementById()` method gets elements by their ID, predates IE6. The `getElementsByName()` method returns a NodeList of elements with matching `name` attributes. It also has deep backwards compatibility.

If you wanted to get all elements of a certain type, you could use `getElementsByTagName()`, which works back to IE6. And the new kid on the block, `getElementsByClassName()`, gets all elements that match a specific class. It works in IE9 and up.

I don’t recommend using any of them.

I’m lazy. I don’t like to think about which selector is the right one to use. The `querySelector()` and `querySelectorAll()` methods do everything those other methods do and more.

The toughest decision I have to make is whether I need all matching elements or just the first one.

Loops

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

`for` loops

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.

Browser Compatibility

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

Scoping and `for` loops

Variables you set in the `for` part of the loop are not scoped to the loop, and this creates some interesting issues.

Changing variable values

If you write code that uses `i` but executes at a later time, `i` will equal the last value it was set to instead of the value it was at the

time you setup the code.

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

for (var i = 0; i < sandwiches.length; i++)
{
    window.setTimeout(function () {
        // This will always log "4"
        console.log(i);
    }, 1000)
}
```

Nested Loops

If you try to nest a loop, you cannot use the same counter variable for both loops.

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

    // This would reset i on the parent loop
, too
    for (var i = 0; i < drinks.length; i++)
{
        // Do stuff...
    }

}
```

To use nested loops, you need to use a different counter variable for each loop.

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

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

}
```

for...in

A `for...in` loop is similar to a `for` loop, but used to loop through objects.

The first part, `key`, is a variable that gets assigned to the object key on each loop. The second part (in the example below, `lunch`), is the object to loop over.

It's considered a best practice to use `obj.hasOwnProperty(key)` to check that the property in each loop belongs to the current object, and isn't inherited from further up the object chain (for nested or *deep* objects).

Here's a working example.

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

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

// returns sandwich, ham, snack, chips, drin
k, soda, desert, cookie, guests, 3, alcohol,
 false
```

Browser Compatibility

Supported in all modern browsers, and IE6 and above.

Skipping and ending `for` and `for...in` loops

You can skip to the next item in a loop using `continue`, or end the loop altogether with `break`. These work with both `for` and `for...in` loops.

```
/**
 * Skipping a loop
 */
var sandwiches = ['turkey', 'tuna', 'ham', '
chicken salad', 'pb&j'];

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

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

    console.log(sandwiches[i]);

}

/**
 * Breaking a loop
 */
var lunch = {
    sandwich: 'ham',
    snack: 'chips',
```

```
    drink: 'soda',
    desert: 'cookie',
    guests: 3,
    alcohol: false,
};

for (var key in lunch) {
    if (lunch.hasOwnProperty(key)) {
        if (key === 'drink') break;
        console.log(lunch[key]);
    }
}
```

`Array.forEach()`

ES5 introduced a new method for loops over arrays: `Array.forEach()`.

You pass a callback function into `forEach()`. The first argument is the current item in the loop. The second is the current index in the array. You can name these two variables anything you want.

Unlike with a `for` loop, you can't terminate the `forEach()` function before it's completed. You can `return` to end the current loop (like you would with `continue` in a `for` loop), but you can't call `break`.

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

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

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

Browser Compatibility

Supported in all modern browsers, and IE9 and above. You can push support back to IE6 with a polyfill.[3]

```
/**
 * Array.prototype.forEach() polyfill
 * @author Chris Ferdinandi
 * @license MIT
 */
if (window.Array && !Array.prototype.forEach) {
    Array.prototype.forEach = function (callback, thisArg) {
        thisArg = thisArg || window;
        for (var i = 0; i < this.length; i++) {
            callback.call(thisArg, this[i], i, this);
        }
    };
}
```

NodeLists and `Array.forEach()`

The `Array.forEach()` method only works with arrays, not NodeLists (like those returned from `querySelectorAll()`) and other array-like collections. While there is a `NodeList.forEach()` method, it has poor browser support at this time and does not let you use any of the newer `Array` methods.

You can convert NodeLists into Arrays with the `Array.from()` method and use `Array.forEach()` instead.

```
Array.from(document.querySelectorAll('.sandw
iches')).forEach(function (sandwich, index)
{
    console.log(sandwich.textContent);
});
```

Browser Compatibility

Works in all modern browsers, including MS Edge, but requires a polyfill for IE support[4].

Objects and `Array.forEach()`

Strangely, there is no `Object.forEach()` method. The `Object.keys()` method returns an array of the keys in an object, and you can call the `Array.forEach()` method on that.

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

Object.keys(lunch).forEach(function (item) {
    console.log(item); // key
    console.log(lunch[item]); // value
});

// returns sandwich, ham, snack, chips, drin
k, soda, desert, cookie, guests, 3, alcohol,
 false
```

Browser Compatibility

`Object.keys()` works in all modern browsers, and IE9 and above. You can push support back to IE7 by adding this polyfill[5], as well as the one for `Array.forEach()`.

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('#sandwich
');

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

// Remove a class
elem.classList.remove('tuna');

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

// Check if an element has a specific class
if (elem.classList.contains('mayo')) {
    console.log('add mayo!');
}
```

Browser Compatibility

Works in all modern browsers, and IE10 and above. A polyfill from Eli Grey[6] 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 elem
ent
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.[7]

Inline Styles

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

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

// Get a style
// If this style is not set as an inline sty
le directly on the element, it returns an em
pty string
// ex. <div id="sandwich" style="background-
color: green"></div>
var bgColor = elem.style.backgroundColor; //
 this will return "green"
var fontWeight = elem.style.fontWeight; // t
his will return ""

// 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

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 existence of attributes (including data attributes) on an element.

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

// 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-chips');

// Check if an element has an attribute
if (elem.hasAttribute('data-drink')) {
    console.log('Add a drink!');
}
```

These methods can also be used to manipulate other types of attributes—things like `id`, `tabindex`, `name`, and so on.

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. [8]

```
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);
```

Browser Compatibility

This works in all modern browsers, and IE9 and above.

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.

```
// This won't work!

var btns = document.querySelectorAll('.click-me');

btns.addEventListener('click', function (event) {
    console.log(event); // The event details
    console.log(event.target); // The clicked element
}, false);
```

For performance reasons, you also **should not** loop over each element and attach an even listener to it.

```
/**
 * This works, but it's bad for performance
 * DON'T DO IT!
 */
var btns = document.querySelectorAll('.click
-me');

Array.from(btns).forEach(function (btn) {
    btn.addEventListener('click', function (
event) {
        console.log(event); // The event det
ails
        console.log(event.target); // The cl
icked element
    }, false);
});
```

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

Instead of listening for an event on specific elements, you attach your listener to the `window` or `document`. Events that happens on elements inside it *bubble up*. We can then check to see if the item that triggered the event has a matching selector.

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

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

}, false);
```

Note: *don't forget to include a polyfill for* `matches()` *if you use it.*

Yes, it is actually better for performance to listen to all clicks on the document than have a bunch of individual event listeners.

As a side benefit, you can dynamically load matching elements to the DOM after the event listener is already set up and it will still work.

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.

```
/**
 * This won't work!
 */
var btns = document.querySelectorAll('.click
-me');

btns.addEventListener('click, scroll', funct
ion (event) {
    console.log(event); // The event details
    console.log(event.target); // The clicke
d element
}, false);
```

Instead, create a named function and pass that into your event listener. This lets you avoid writing the same code over and over again, and keeps your code more DRY.

You don't need to include the parentheses (`()`) on the function. The `event` object is automatically passed in as an argument.

```
// Setup our function to run on various even
ts
var logTheEvent = function (event) {
    console.log('The following event happene
d: ' + event.type);
};

// Add our event listeners
document.addEventListener('click', logTheEve
nt, false);
window.addEventListener('scroll', logTheEven
t, 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 docume
nt
document.addEventListener('focus', function
(event) {
    // Run functions whenever an element in
the document comes into focus
}, true);
```

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.

```
<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="co
ntent-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 consol
e
    console.log(event.target);

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

    // Log the clicked element in the consol
e
    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 h
ave the .accordion-toggle class
    if (!event.target.classList.contains('ac
cordion-toggle')) return;

    // Get the target content
    var content = document.querySelector(eve
nt.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 h
ave the .accordion-toggle class
    if (!event.target.classList.contains('ac
cordion-toggle')) return;

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

We'll pass the returned NodeList through `Array.from()` to convert it into an array. Then, we'll use `Array.forEach()` to loop through each item 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 h
ave the .accordion-toggle class
    if (!event.target.classList.contains('ac
cordion-toggle')) return;

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

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

    // Get all open accordion content, loop
through it, and close it
```

```
    var accordions = Array.from(document.que
rySelectorAll('.accordion-content.active'));
    accordions.forEach(function (accordion)
{
        accordion.classList.remove('active')
;
    });

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

});
```

Note: *Don't forget to add a polyfill for* `Array.from()` *to support IE.*

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 `forEach()` 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
```

```
through it, and close it
    var accordions = document.querySelectorAll('.accordion-content.active');
    accordions.forEach(function (accordion) {
        accordion.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.