Notes - JavaScript - Working with the DOM &c.

- Dr Nick Hayward

A collection of notes &c. on plain JavaScript and working with the DOM.

Contents

Intro *DOM* or document object model, is a tree data structure used for multiple markup languages, including HTML and XML.

Move through the DOM tree DOM nodes contain a wealth of links to other nearby nodes.

Every node has a `parentNode` property, which points to the containing, parent node.

Every element node (node type 1) has a `childNodes` property that points to an array-like object holding its children.

Additional convenience links for basic traversal include,

- `firstChild` and `lastChild` - returns `null` for nodes without children
- `previousSibling` and `nextSibling` - find nodes with the same parent
  - first and last child elements will return `null` for the appropriate call…
- `children` property returns nodes - only element nodes, and not others such as text nodes…

Each DOM node includes various properties, such as `nodeValue` . For a text node, we may return the string of text contained by the node.

tree recursion For nested data structures, such as the tree-based DOM, recursion is a useful option for traversal.

Recursion may be used to scan a document for various specified nodes, e.g. text nodes.

```
function talksAbout(node, string) {
  if (node.nodeType == document.ELEMENT_NODE) {
    for (let i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string)) {
        return true;
      }
    }
  return false;
  } else if (node.nodeType == document.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}
console.log(talksAbout(document.body, "book"));
// → true
```

Find elements Traversal is a useful option for tree-based data structures, such as the DOM.

However, we may also need to find a specific node, or collection of similar nodes.

Methods include,

- `getElementsByTagName()`
- `getElementById()`
- `getElementsByClassName()`

e.g. find the `href` attribute of a link element in the DOM

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

This method, `getElementsByTagName()`, may be used with all element nodes. It collects all descendant elements with the specified element tag, and returns them as an array-like object.

Single, unique nodes may be found using the method `getElementById()`. e.g.

```
let dog = document.getElementById("daisy");
console.log(dog.src);
```

A similar method, `getElementsByClassName()`, will search the DOM for all element nodes with the specifed selector class name.

Modifying the DOM We may modify most of the DOM's structure, effectively by changing various parent-child node relationships.

Add a *child* node to the DOM,

- `appendChild()` - adds node to the end of the list of children
- `replaceChild()` - replace a chid node another specified node…
- `insertBefore()` - insert node before another specified node, e.g.

```
let paragraphs = document.body.getElementsByTagName("p");
document.body.insertBefore(paragraphs[2], paragraphs[0]);
```

n.b. a specific node may exist in only a single place in the DOM. If it is moved it will first be removed from the DOM, and then inserted to the new position.

Create nodes in the DOM We may combine some of the above methods and options to enable node creation and insertion.

e.g. replace all link nodes with text nodes

```
function removeLink() {
  let links = document.body.getElementsByTagName('a');
  for (let i = links.length - 1; i >= 0; i--) {
    let link = links[i];
    if (link.href) {
      let text = document.createTextNode(link.href);
      link.parentNode.replaceChild(text, link);
    }
  }
}
```

n.b. the `for` start posn uses the end of the returned `links` object. Then, we can decrement the index for the loop's iterator.

Why start at the end? `links` node list is *live* relative to the current state of the DOM. If we started at the front of the index, as usual, and then removed the first link, the iterator would be out by one count for the second iteration of the `for` loop.

n.b. if a frozen set of nodes, from a live DOM, is required, as a representation of a given point in time, we may use `Array.from()` to create a standard array.

create element nodes Use `document.createElement()` to create element nodes.

e.g.

```
function elemBuild(type, ...children) {
  let node = document.createElement(type);
  for (let child of children) {
    if (typeof child != 'string') {
      node.appendChild(child);
    } else {
      node.appendChild(document.createTextNode(child));
    }
  }
  return node;
}
document.getElementById('berryhead').appendChild(
  elemBuild('footer', '-----', elemBuild('aside', 'unknown author, Berry Head'))
);
```

Work with node attributes Default attributes, such as `href` , may be accessed using a property of the same name on the element's DOM object. This is built into JavaScript as a global attribute property.

Custom attribute names may also be accessed using

- `getAttribute()`
- `setAttribute()`

e.g.

```
// get example blockquote nodes
let quotes = document.body.getElementsByTagName('blockquote');
// loop through quotes - freeze quotes object using Array.from
for (let quote of Array.from(quotes)) {
  if (quote.getAttribute('data-visible') === 'true') {
    quote.setAttribute('data-visible', 'false');
  }
}
```

Custom attributes may be prepended with `data-` to easily denote a non-default, custom attribute. It also helps with DOM lookup and traversal.

DOM layout and JavaScript Standard layout is commonly represented as either

- block-level - `p` , headings…
- inline - `span` , `a` …

Block-level will be rendered on a new line, by default, and inline will continue in the same rendered line &c.

Size and position of a HTML element may be accessed using Javascript.

- `offsetWidth` and `offsetHeight` properties - return space an element *occupies* in pixels
- `clientWidth` and `clientHeight` properties - return space *inside* an element, ignoring border width

e.g.

```
// binding for element to size...
let quote = document.getElementById('berryhead');

// get size of space *inside* the element (ignores border width)...
console.log('clientHeight = ', quote.clientHeight);
// get size of space element occupies in pixels
console.log('offsetHeight = ', quote.offsetHeight);
```

We may also check scroll position, and update the DOM to reflect scroll to a specified selector,

```
// lorem iframe content body
const lorem = document.getElementById('lorem');
const loremDoc = lorem.contentDocument;
// populate lorem iframe
loremDoc.body.innerHTML = contentHTML;
// get *overview* section from lorem
const overview = loremDoc.getElementById('overview');
// scroll iframe to posn of element with *overview* id...
loremDoc.scrollingElement.scrollTop = overview.offsetTop;
```

DOM styling We may also style DOM node elements using JavaScript, modifying the Style object.

e.g.

```
const quote = document.getElementById('berryhead');
quote.style.color = '#779eab';
quote.style.width = '200px';
quote.style.padding = '10px';
quote.style.boxSizing = 'border-box';
quote.style.border = "1px solid #779eab";
quote.style.margin = "10px";
console.log('styles = ', quote.style);
```

Further details of the Style object,

- https://www.w3schools.com/jsref/dom_obj_style.asp

n.b. `boxSizing` may prove useful for organizing content - any specified properties for padding, border are calculated as part of the overall width - they don't add to the width, which is useful for grids, parallel boxes &c.

DOM positioning and animating We may also use styling &c. to manipulate positioning and create animated effects.

Positioning of an element will be set to a value of `static` by default. However, we may update this value to manipulate the positioning and flow of an element in a document.

Position values include,

- static - element sits in normal position in the document
- relative - element still occupies space in the document, but top/left may be used to reposition *relative* to default position
- absolute - element removed from normal document flow, no longer occupying space, and may overlap other elements…

We may also animate a DOM element by manipulating position and various properties of the style object

e.g.

```
let start = 100;

function domAnimate(time, lastTime) {

  let animated = document.getElementById('right');

  if (start > window.innerWidth / 2) {
    return;
  }

  start += 0.5;
  animated.style.left = start + 'px';
  requestAnimationFrame(newTime => domAnimate(newTime, time));
}
```

In this example, we use recursion to call the `domAnimate` function until the specified condition is met.

The `requestAnimationFrame()` method is available on the global `window` object. This method tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint. The method takes a callback as an argument to be invoked before the repaint.

- MDN - https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

Load event As a page finishes loading, the `load` event will fire on the global window and document body objects.

This is often used to schedule app initialisation &c., in particular for actions that require a loaded full document.

Remote images and scripts will also use a `load` event, which indicates that the file &c. has been loaded.

If a page is closed (or navigated from…), a `beforeunload` event will fire. This permits options to ensure a user may not lose work by closing the document.

We may use this option to check with the user, to ensure they wish to navigate away from a page. We may return a `non-null` value from the handler to perform this check.

However, there is debate as to the merit of such UX practices…

n.b. `load` event do not propagate…

DOM events and handlers Modern web browsers actively notify application scripts when an event occurs, e.g. a user click in the UI.

Browsers do this by allowing a script to register functions as handlers for specified events.

add event to DOM node An event handler is registerd in a context.

As such, event listeners will only be called for a given event, in the context of the registered containing object.

e.g. a listener may be added to the following `button` node in the DOM

```
<button>click the button</button>
```

which is then handled as follows,

```
let button = document.querySelector('button');
button.addEventListener('click', () => {
  console.log('button has just been clicked...');
})
```

This event handler has been bound to the context of the `button` node.

`onclick` vs `addEventListener` Attaching an `onclick` handler to a node has a similar effect as the above `click` with `addEventListener` .

However, a node may only accept one `onclick` handler, whilst we may add multiple `addEventListener` handlers to the same node.

remove a handler As we attach a handler, we may also dynamically remove, or unbind, an event handler.

e.g.

```
let button = document.querySelector('button');
function singleTime() {
  console('button has now been used...');
  button.removeEventListener('click', singleTime);
}
button.addEventListener('click', singleTime);
```

event objects Event handlers are passed an argument, which is the `event` object.

The `event` object includes further information about the event. e.g. properties for mouse button clicked.

We also find that such properties will slightly vary from one event to another. However, a property identifying the event itself should always be available, e.g. click, mousedown, mouseup…

propagation Handlers registered for a given node with children will, commonly, also receive data on child propagated events.

Event handlers for a paragraph node element will see a click event for a child button element.

However, if both the paragraph and button have a handler, the more specific one will be executed first.

Propagation will start at the most specific handler, then move to the parent node, then the root of the document. After all options have been tested, the event will be passed to the window object if no other resolution.

This propagation may be stopped at any point by a handler calling `stopPropagation` on the event object.

e.g. inner button and out clickable element - by explicitly stopping propagation for at the inner node, the event cannot inadvertently activate any actions &c. on the parent clickable element.

```
let para = document.querySelector("p");
let quote = document.querySelector("blockquote");
para.addEventListener("mousedown", () => {
  console.log("Handler for paragraph...");
});
quote.addEventListener("mousedown", event => {
  console.log("Handler for quote...");
if (event.button == 2) event.stopPropagation();
});
```

Most event handlers will also include a `target` property, which refers to the originating node.

e.g.

```
let para = document.querySelector('p#quote-container');
let quote = document.querySelector("blockquote");
para.addEventListener("click", event => {
  console.log("Handler for paragraph as quote container...");
  event.stopPropagation();
});
quote.addEventListener("mousedown", event => {
  console.log("Handler for quote...");
  if (event.button == 2) event.stopPropagation();
});

// check target for a passed node/s
document.body.addEventListener('click', event => {
  if (event.target.nodeName === 'P') {
    console.log('clicked', event.target.textContent);
  }
});
```

In this example, we added a specific handler for a paragraph event with a unique ID reference. Then, we add a generic event listener for all paragraphs. The specific handler will be executed first, and then the generic.

default actions DOM events &c. will, customarily, include default behaviours. For most events, JS event handlers will be called before such default behaviour is actioned.

However, we may also explicitly prevent such default actions and behaviour using the `preventDefault()` method on the event object.

e.g. we may prevent the default actions for a standard anchor link, thereby implementing a custom context menu.

```
// prevent default event handling for links - use standard function to gain access to 'this'
// arrow function restricts access to 'this'...
document.querySelector('a').addEventListener('click', function(event) {
  //console.log('event = ', event);
  if (event.which === 1) {
    event.preventDefault();
    this.style.color = '#cd0603';
    console.log('link deactivated');
    console.log();
  }
});
```

In the above example, we're stopping anchor links, and then updating the text colour for the link clicked.

n.b. an arrow function cannot be used if we need access to `this` .

mouse motion `mousemove` event is fired each time the mouse is moved - use this event to simply track the position of the mouse.

e.g. we might implement tracking for a mouse drag event

```
let updateBar = document.getElementById('update-bar');
updateBar.style.width = '60px';
updateBar.style.height = '20px';
updateBar.style.backgroundColor = '#779eab';
let lastMouseX;

// check for mouse button pressed down...
updateBar.addEventListener('mousedown', function(event) {
  if (event.button == 0) {
    lastMouseX = event.clientX;
    console.log('mouse x in update bar = ', lastMouseX);
    // check for mouse move, and callback for draw on drag...
    this.addEventListener('mousemove', barDrag);
    event.preventDefault();
  }
});

function barDrag(event) {
  if (event.buttons == 0) {
    console.log('event buttons...0');
  }
  // calculate distance from last x to current clicked x
  let xDist = event.clientX - lastMouseX;
  let xWidth = Math.max(10, updateBar.offsetWidth + xDist);
  updateBar.style.width = xWidth + 'px';
  lastMouseX = event.clientX;
}
```

scroll events Scroll events are useful for tracking position of a DOM element, or simply following a user as they scroll page content.

As an element is scrolled, a `scroll` event is fired.

e.g. add a progress bar to the top of the page, which records % of page scroll by the user.

```
const progress = document.getElementById('progress');
// define default styles for progress bar...
progress.style.borderBottom = '10px solid #d282ed';
progress.style.width = 0;
progress.style.position = 'fixed';
progress.style.top = 0;
progress.style.left = 0;
// get content element
const poppins = document.getElementById('poppins');
// add some content for Poppins
poppins.appendChild(document.createTextNode('supercalifragilisticexpialidocious '
       .repeat(2500)));
// add event handler for page scroll
window.addEventListener('scroll', () => {
  // define max scroll point...
  let max = document.body.scrollHeight - innerHeight;
  // update progress width in % - pageYOffset (current scroll) divided by max scroll posn...
  progress.style.width = `${(pageYOffset / max) * 100}%`;
```

```
});
```

See articles example in the source repository for a working example of scroll manipulation and usage.

focus events We may also add events for focus and blur within elements such as an `input` field,

```
// get node for rendering output
let output = document.querySelector('#output');
// click in input fields
let fields = document.querySelectorAll("input");
// loop through each input field
for (let field of Array.from(fields)) {
  // add event listener for focus
  field.addEventListener("focus", event => {
    // clear input value on focus...
    field.value = "";
    // get value of field attr
    let inputText = event.target.getAttribute('data-help');
    // render instructions to user...
    output.textContent = inputText;
  });
  // add event listener for remove focus - blur event...
  field.addEventListener("blur", event => {
    // reset output text...
    output.textContent = '';
  });
}
```

In this example, we add an event listener for clicking in an input field, `focus` , and a complementary listener for leaving an input field, `blur` .

As a user clicks inside a given input field, we may perform various actions such as outputting text, cleaning up the input field, and so on.

n.b. focus events do not propagate…

Events and the event loop Handlers for browser events are scheduled as they happen, which means they must wait for other running scripts finish before execution.

Therefore, such queue based events need to be considered relative to performance. Too many events, or a single long-running event, may cause the browser and app to become slow and unresponsive.

web workers For apps that require time-consuming execution, including calculations &c., we may use *web workers* in the browser.

A *worker* is a JavaScript process designed to run alongside the main app script, and in its own timeline.

Such workers are often used to allow us to perform heavy or time consuming computation.

Web workers also benefit from their scope, isolated from an app's global scope. Instead, messages are used to enable communication with workers. We're able to send bi-directional messages to a web worker.

Basic web worker pattern is as follows,

1. create a separate file for the web worker, `worker.js`

e.g.

```
addEventListener('message', function(e) {
  postMessage(e.data);
}, false);
```

`postMessage()` will return a message to the caller, in this case the main file,

2. create a main file to send messages to the web worker,

```
// instantiate a worker object
const worker = new Worker('worker.js');

// add a listener to the worker object
worker.addEventListener('message', function(e) {
  // output test return data for message from worker...
  document.getElementById('worker-output').innerHTML = `Worker said: ${e.data}`;
}, false);

// send some data to the worker
worker.postMessage('We send greetings from Planet Earth...');
```

Debouncing Some DOM events may fire regularly, and in rapid succession. e.g. scroll and mouse events.

This may cause unresponsive apps, or missed events by a user, due to the large number of successive calls.

The other issue is that we may also need to perform time sensitive computation for each successive event. To ensure such computation, for example, is not executed too often, we may use `setTimeout` to regulate such calls.

e.g. we may add a listener to a `textarea` for key events. We may not wish to call a handler for each and every keypress, instead relying on a slight pause in typing to then execute such code.

e.g.

```
// define test text area...
let textarea = document.getElementById('debounce-text');
// define timeout for debounce
let timeout;

// add listener to test text area
textarea.addEventListener('input', () => {
  // ensure timer is reset to maintain debounce...
  clearTimeout(timeout);
  // update timeout - every 500ms get value from textarea and log...
  timeout = setTimeout(() => console.log('typed...', textarea.value), 500);
});
```

Another fun example is for checking mouse coordinates on `mousemove` .

e.g.

```
// define initial schedule
let scheduled = null;
// add window event listener for mousemove...
window.addEventListener("mousemove", event => {
  //console.log('part1');
  // check and update scheduled
  if (!scheduled) {
    //console.log('part2');
    // call timeout
    setTimeout(() => {
      //console.log('part3');
      // update dom node every 500ms - output mouse coordinates...
      document.getElementById('debounce-output').innerHTML =
                              `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
      // clear schedule to restart 500ms check...
      scheduled = null;
      //console.log('part4')
    }, 250);
  }

//console.log('part5');

// stire each mouse move event in scheduled variable...
scheduled = event;
//console.log('part6');
});
```

Execution of this example uses the following pattern,

- execution pattern and order - depends on timer e.g. 250ms compared to 500ms
- part1, part5, part6, part1, part5, oart6, part1, part5, part6, part3, part4
- scheduled is set to null
- event listener added to window for mousemove
- !scheduled is initially true - setTimeout is started with custom interval
- timeout keeps running and executing every 250ms
- schedule is updated for event (mousemove coords) for each pass through listener outside of timer execution