Cookies, DOM manipulation

Browser windows

Two main kinds of windows:

- Regular browser window with web content. Could be a tab or a separate window. (Each browser tab is represented by a different Window object.)
- Modal dialog windows: `alert(message)`, `confirm(message)` returns true if user clicks OK and false otherwise, `prompt(message)` returns string entered by user
  - Not used much anymore in modern apps; annoying; ugly; limited
  - Generated by browser; do not contain HTML or CSS
  - *modal* means window stays in foreground until user resolves it; prevents user from doing anything else

Some informative window properties you can examine:

- `window.screenX` and `screenY` will tell you where top-left corner of the browser window is on the screen. The (0,0) coordinates are the top-left corner of the screen. (screenLeft and screenTop on IE)
- `window.innerWidth` and `innerHeight` tell you the size of the *visible content*, including scrollbars (but not including window frame, toolbars, menus, etc.). use `outerWidth` and `outerHeight` for full size of window. Note that none of these dimensions reflect a web documents full, scrollable width or height, only the part that is visible.
- `window.screen` gives you access to various properties of the physical screen that the document is being viewed on, like `window.screen.height`.

Opening and closing tabs, windows

(I'm leaving out many details in this section because this isn't a focus of the course.)

The most reliable way to open a new window or tab is using an HTML link:

```
<a href=”photos.html” id=”photosLink” target="photosTab">
View photos. (Will open new window)</a>
```

A tab is considered a different Window object in this context. Browser settings determine whether clicking on such a link will display content in a new tab or a new window.

If you open windows using JavaScript, you can set up your HTML and JS code to use the `a` tag (as above) as a fallback in case JavaScript is disabled or user preferences disable JS pop-ups. (Browsers block new JavaScript-triggered windows and tabs by default, so your user will have to allow the action in their preferences.)

> We generally avoid opening and closing windows in JavaScript since it's usually blocked, it's annoying to the user, content displayed that way can't be indexed by search engines, and the JS can be disabled anyway, so, as usual, you can't count the content to be displayed properly in the first place.

You can open a new tab with code like `newTab = window.open("absolute or relative URL")`. (Again, whether it's a new tab or a separate window is up to user preferences, but the default preference is a tab.) If you don't provide a URL, an empty tab will open. *Note that the new window will not necessarily load its URL immediately; the loading may be deferred to a later time, possibly after your script is done executing.*

In the previous example, `newTab` is a reference to a Window object that represents the tab you opened. If the browser blocked the creation of the new Window, `newTab` will be `null`, *even if the user allowed the new tab to be opened later by unblocking it in settings*. However, if the tab wasn't blocked and `newTab` refers to it, then you can close it: `newTab.close()`. You can also check that it closed successfully by checking the boolean value of `newTab.closed`. You can't close a window unless that window was opened using JavaScript.

If there are many windows or tabs open, you can give a particular window focus.

```
// here `popup` is a Window object
// assume it represents a second window
if ((popup !== null) && !popup.closed) {
 popup.focus();
}
```

Here's an example of creating a new window object with additional properties:

```
var popup = window.open("somepage.html", "NameForTheWindowWithoutSpaces",
  "height=200,width=200,location=no,resizable=yes,scrollbars=yes");
```

Many of these properties aren't standardized across browsers.

Each browser window/tab has its own execution context and therefore its own Window object. Let's say your main app is running in Window A and you open a pop-up which we'll call Window B. You can access B's `window` object from A and vice versa.

```
// code running in Window A
// `popup` represents Window B
console.log(popup.window.screenX);
```

Browsers have security restrictions for how JS interacts with other windows. In particular, reading or writing window properties is usually limited to content that
has the same origin, meaning the same domain and protocol (e.g. https). That is, your JS can't mess with window objects that belong to other web properties or subdomains and vice versa.

HTML DOM Manipulation

DOM manipulation refers to dynamically altering a document's content. We usually deal with HTML documents, but a DOM can represent other kinds of documents like SVG or XML. We're already been performing DOM manipulation with methods such as `createElement` and `appendChild`. This section will explain DOM manipulation in more detail.

Recall that the HTML DOM is a tree data structure whose root element is the HTML element. With that in mind, we might say that a DOM element has a *parent*, *children*, *ancestors*, *siblings*, and so on.

The Document Object, Nodes and Elements

`window.document`, or just `document`, represents the content loaded in the window. We've already used a few methods on the Document object such as `getElementById`.

- You can check or modify the document's title with `document.title`
- You can check the doctype with `document.compatMode`, which can be useful for debugging or for handling cross-browser issues.
- If the browser is using quirks mode, the `compatMode` will be `'BackCompat'`
- If the browser is using standard rendering mode, the `compatMode` will be `'CSS1Compat'`

Our preceding definition of the DOM only considers HTML elements, but the DOM actually contains other kinds of objects as well. If we consider all DOM nodes, not just elements, then the root *element* `<html>` has one parent: the `document` object. The `document` object is the root *node* of the DOM.

The DOM also includes nodes to represent text content, whitespace, comments.

You can see this difference in the JS console: try `document.firstChild` versus `document.firstElementChild`, or `document.firstElementChild.parentNode` versus `document.firstElementChild.parentElement`.

Similarly, if `el` is any Element:

- `el.children` is a list (HTMLCollection) of child elements only
- `el.childNodes` is a list (NodeList) of all child nodes, including both elements and other kinds of nodes like Text, which represents text content of an HTML tag, for example.

Both Document and Element inherit from an interface called Node, which provides some useful properties:

- `el.nodeName` can give you the name of the HTML tag like "DIV" or "TABLE"
- `el.nodeType` tells you whether `el` is an element, a text node, a comment, etc.

There are also some shortcut properties on the `document` object to allow easy access to commonly-needed nodes:

- `document.body` gets the body element
- `document.forms` gets an HTMLCollection of all form elements
- Same idea for `document.images` and `document.links` for image elements and anchor elements
- `document.styleSheets` get a list of all CSS style sheets. You can disable and enable styles this way, for example: `document.styleSheets[0].disabled = true;`

Finding Elements

HTMLCollection versus NodeList

We've used many methods like `getElementsByTagName` that return an HTMLCollection, which has some special characteristics.

1. An HTMLCollection is always *live*, which means that it always lists what is in the DOM *right now*. For example, let's say `x = getElementsByTagName("p")`; if I later add more paragraphs to the document with `appendChild` and then look inside `x` again, `x` will list the newly-added paragraphs as well. *Be careful when iterating over an HTMLCollection, since its length may change when the DOM changes.*

2. You can refer to an item in the HTMLCollection either by its index in the list or by its id. For example, if I have a paragraph with id "instructions", and `x` is an HTMLCollection of all paragraphs, then `x.instructions` or `x["instructions"]` both refer to that particular paragraph, and `x[5]` is the 6th paragraph.
3. HTMLCollection only ever contains objects of type Element.

We've also seen that some methods and properties return a NodeList, like `el.childNodes`. `el.getElementsByName` also returns a NodeList.

- NodeLists contain Node objects, which can be elements, text nodes, etc.
- NodeList objects are sometimes *live* like HTMLCollection, but sometimes they are *static*. It depends on the situation, so you have to check the documentation for your particular case. The return value of `el.childNodes` is always live, and we'll see an example later of a static NodeList.

In addition to methods like `getElementsByTagName`, `getElementsByClassName` we also have methods that collect elements by CSS selector. For example:

```
// get NodeList of all paragraphs in the document
// that belong to the "important" CSS class
var items = document.querySelectorAll("p.important");
// get one Element that is descendant of el and
// is a an anchor inside a div
var link = el.querySelector("div a");
// get one Element anywhere in whole document
// that is a an anchor inside a div
var link = document.querySelector("div a");
```

It's important to remember that `querySelectorAll` returns a *static* NodeList, which means it always stays the same, even when the DOM changes. On the other hand, the `getElements...` methods return an HTMLCollection.

Specifying a root element

When you search for elements with `querySelector`, `querySelectorAll`, `getElementsByClassName` and `getElementsByTagName`, you can search in the whole document:

```
document.getElementsByTagName("p");
```

This means the root element of your search is the DOM's document object.

Or you can search in a subtree of the DOM, that is, starting with some child element.

```
someEl.getElementsByTagName("p");
```

The above example will return all paragraphs that are descendents of `someEl`. The root of the search is `someEl`. Note that `someEl` is not included in the search result, even if it matches: we are search for paragraphs under `someEl`, excluding `someEl`.

Things to do with elements

We've already seen that you can access element attributes as properties of the corresponding Element object. For example, if our document contains `<a href="b.html" id="b">Click <span>this</span></a></p>` then we can do the following:

```
var link = document.getElementById("b");
// print b
console.log(link.id);
// print b.html
console.log(link.href);
```

We've also used element properties that do not correspond to attributes like `link.textContent`, which in our example is `"Click this"`. There is also `innerHTML`, which allows you to get or set the HTML code inside an element. Here, `link.innerHTML` is `"Click <span>this</span>"`.

Summary of methods to manipulate elements:

- `par = document.createElement("p")` creates a new HTMLElement object (a paragraph in this case)
- `el.appendChild(par)` will put `par` at the end of `el`'s children. If `par` is already a child of `el`, it will get moved to become the last child.
- Similarly `el.removeChild(par)` will remove the paragraph.
- `el.insertBefore(par, childEl)` will put `par` right before `childEl` in `el`'s children
- `el.replaceChild(par, childEl)` will remove `childEl` from `el`'s children and put `par` in its place.

Here's an example of removing all children from an element. The loop removes the first child one at a time until the firstChild property is undefined, which means there are no children left.

```
while (el.firstChild) {
  el.removeChild(el.firstChild);
}
```

Methods continued:

- `someText = document.createTextNode("More text")` creates a new Text object, which is a kind of DOM node. You can insert the `someText` node to become a child of an element with `appendChild`, etc. This is useful if you want to add text to a node that already has other children.
- `parClone = par.cloneNode()` is another way to create new elements: it creates an independent copy of `par` with all the same properties. Cloning an element does not clone any event handlers attached to the original element.

Let's go back to our previous example, where `link` refers to the anchor element:
`<a href="b.html" id="b">Click <span>this</span></a></p>`

If I write `link.textContent = "foo"` it will replace the anchor's text content entirely. If instead I do `link.appendChild(document.createTextNode("foo"))`, then I'm keeping the original child content of the anchor intact. The result will be:
`<a href="b.html" id="b">Click <span>this</span>foo</a></p>`

A better way to change element style

When you get or set the `style` property on an element object, you are interacting with its *inline style*. The `style` property doesn't reflect external styles applied to an element.

If you want to know all the styles that apply to an element rather than just the inline style, use `allStyles = window.getComputedStyle(myElement)`, which returns a read-only CSS2Properties object. Then you could look at CSS properties like `allStyles.backgroundColor`. (Note that this won't work on old IE.)

Updating CSS properties in JavaScript one by one is tedious and repetitive. So, rather than changing the inline style of an element with JavaScript, consider changing its CSS class instead. That is, pre-define a CSS class in your external CSS, then when you want to change the look of an element, just change its class with JavaScript.

```
/* some css file loaded in head tag */
.selected {
  color: red;
  border: solid 1px;
  background-color: yellow;
}
```

```
someElement.className = "selected";
// instead of
// someElement.style.backgroundColor = red;
// etc...
```

Cookies

The HTTP protocol is designed to be stateless: when you navigate from one page to another, or you refresh a page, all information about you and your actions is lost. To compensate for this, browser cookies were invented to *persist* user data across many visits. (There are also *sessions*, but that's a server-side thing.) Cookies help implement web apps with user accounts, preferences, targeted ads, etc.

A cookie is a named collection of information associated with a server, and it is stored by your browser. A web app can save a cookie in your browser. The cookie can then be accessed later by the web app and processed on the client side or passed to the server that originally created it.

Cookie data includes:

- name
- value
- expiration time (in a format like `Mon, 03 Jul 2006 21:44:38 GMT`)
- path (like /store)
- domain (like example.com)

A cookie can only be read when the user is at the specified domain and path, like example.com/store. (If you're in a subdirectory of store, that counts as being in store.)

Cookies are designed to be used together with server-side code[1], but we'll use them in some client-side only examples to practice interacting with them. More precisely, server-side code can set cookies (a server can send a cookies to a browser), but we can also set cookies using JavaScript alone. Once a cookie is set, *it is sent with every request to the server that it's associated with* as part of the request headers. (HTTP headers are explained in previous notes; look for HTTP GET and POST in the notes.) Since browsers expect cookies to be used with a server, you need to set up a local development server to be able to run and test JS code that uses cookies.

Cookies have limitations:

- Cookies can be blocked, modified (so apps must always validate cookie data), or deleted by the user.
- Cookies have a max size.
- Cookies increase the size of each HTTP request.

- There is a limit on how many cookies can be sent to a user.
- Never store sensitive information in cookies, since they can easily be examined on a user's computer.

Here's an example of creating two cookies named `fontSize` and `theme` with the path, domain and expiration set to the default values:

```
// you can only set one cookie at a time
document.cookie = "fontSize=14";
// each assignment to document.cookie appends to the list of cookies
document.cookie = "theme=dark;domain=*.example.com";
// you can replace an existing cookie
document.cookie = "fontSize=12";
// reading from document.cookie shows a string of all valid cookies
console.log(document.cookie)
```

When you access `document.cookies`, you get a string that shows only name and value pairs like `fontSize=14;theme=dark`, but not any other properties like expiration date.

To set an expiration date the correct format, use Date's `toUTCString` method.

```
// today's date
var expire = new Date();
// increment by one day
expire.setDate(expire.getDate() + 1);
// looks like Mon, 03 Jul 2006 21:44:38 GMT
var dateInUTCFormat = expire.toUTCString();
document.cookie = "someName=someValue;expires=" + dateInUTCFormat;
```

To delete a cookie, create a cookie with the same name and an expiration date in the past:

```
document.cookie = "fontSize=;expires=Thu, 01-Jan-1970 00:00:01 GMT";
```

---

1. The new recommended way to persist client-side data is with the Web Storage API but it's not officially in the web standard yet and it's beyond the scope of this course. See also Storing Data Client-Side in the Eloquent JavaScript book.↩

Last update: 2017-03-28 10:23:57

Source: _javascript_423/lectures/09_cookies_dom.md