**JAVASCRIPT WEEK 6 SUMMARY**

**Day 1 – Frames and windows**

**Popups and window methods –** A popup window is one of the oldest methods to show the additional documents to the user. Just run: window.open(‘https://javascript.info/’) and it will open a new window with a given URL. Most modern browsers are configured to open new tabs instead of separate windows. The initial idea for popups was to show another content without closing the main window. As of now, there are other ways to do that: you can load content dynamically with fetch and show it in a dynamically generated <div>. So, popups aren’t something you use every day.

Still, there are tasks where popups are still used, e.g. for OAuth authorization (login with Google/Facebook, etc.), because:

* A popup is a separate window with its own independent JavaScript environment. So, opening a popup with a third-party non-trusted site is safe.
* It’s very easy to open a popup.
* A popup can navigate (change URL) and send messages to the opener window.

**window.open –** The syntax to open a window is: window.open(url, name, params):

url – A URL to load into the new window.

name – The name of the new window. Each window has a window.name, and here you can specify which window to use for the popup. If there’s already a window with such a name – the given URL opens in it, other wise a new window is opened.

params – The configuration string for the new window. It contains settings, delimited by a comma. There must be no spaces in params, for instance: width:200, height=100.

Settings for params:

Position:

* left/top (numeric) – coordinates of the window top-left corner on the screen. There is a limitation: a new window cannot be positioned offscreen.
* width/height (numeric) – width and height of a new window. There is a limit on minimal width/height, so it’s impossible to create an invisible window.

Window features:

* menubar (yes/no) – shows or hides the browser menu on the new window.
* toolbar (yes/no) – shows or hides the browser navigation bar (back, forward, reload etc) on the new window.
* location (yes/no) – shows or hides the URL field in the new window. FF and IE don’t allow to hide it by default.
* status (yes/no) – shows or hides the status bar. Again, most browsers force it to show.
* resizable (yes/no) – allows to disable the resize for the new window. Not recommended.
* scrollbars (yes/no) – allows to disable the scrollbars for the new window. Not recommended.
* There is also a number of less supported browser-specific features, which are usually not used. Check window.open in MDN for examples.

**Accessing popup from window –** The open call returns a reference to the new window. It can be used to manipulate it’s properties, change location and even more. In this example, you generate popup content form JavaScript:

let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(“Hello, world!”);

And here you modify the contents after loading:

let newWindow = open('/', 'example', 'width=300,height=300')

newWindow.focus();

alert(newWin.location.href); // (\*) about:blank, loading hasn't started yet

newWindow.onload = function() {

let html = `<div style="font-size:30px">Welcome!</div>`;

newWindow.document.body.insertAdjacentHTML('afterbegin', html);

};

Please note: immediately after window.open, the new window isn’t loaded yet. That’s demonstrated by alert in line(\*). So you wait for onload to modify it. We could also use DOMContentLoaded handler for newWin.document.

Same origin policy – Windows may freely access content of each other only if they come from the same origin(the same protocol://domain:port). Otherwise, e.g. if the main window is from site.com, and the popup from gmail.com, that’s impossible for user safety reasons.

**Accessing window from popup –** A popup may access the “opener” window as well using window.opener reference. It is null for all windows except popups. If you run this code, it replaces the opener (current) window content with “Test”:

let newWin = window.open(“about:blank”, “hello”, “width=200,height=200”);

newWin.document.write(

“<script>window.opener.document.body.innerHTML = ‘Test’ <Vscript>”

);

So the connection between the windows is bidirectional: the main window and the popup have a reference to each other.

**Closing a popup –** To close a window: win.close(). To check if a window is closed: win.closed.

Technically, the close() method is available for any window, but window.close() is ignored by most browsers if window is not created with window.open(). So it’ll only work on a popup. The closed property is true if the window is closed. That’s useful to check if the popup (or the main window) is still open or not. A user can close it anytime, and your code should take that possibility into account. This code loads and then closes the window:

let newWindow = open('/', 'example', 'width=300,height=300');

newWindow.onload = function() {

newWindow.close();

alert(newWindow.closed); // true

};

**Scrolling and resizing –** There are methods to move/resize a window:

* win.moveBy(x,y) - Move the window relative to current position x pixels to the right and y pixels down. Negative values are allowed (to move left/up).
* Win.moveTo(x,y) – Move the window to coordinates (x,y) on the screen.
* Win.resizeBy(width,height) - Resize the window by given width/height relative to the current size. Negative values are allowed.
* Win.resizeTo(width,height) – Resize the window to the given size. There’s also window.onresize event.

Only popups – To prevent abuse, the browser usually blocks these methods. They only work reliably on popups that we opened, that have no additional tabs.

No minification/maximization - JavaScript has no way to minify or maximize a window. These OS-level functions are hidden from Frontend-developers. Move/resize methods do not work for maximized/minimized windows.

**Scrolling a window**

* win.scrollBy(x,y) – Scroll the window x pixels right and y down relative the current scroll. Negative values are allowed.
* win.scrollTo(x,y) – Scroll the window to the given coordinates (x,y).
* elem.scrollIntoView(top = true) – Scroll the window to make elem show up at the top (the default) or at the bottom for elem.scrollIntoView(false). There’s also window.onscroll event.

**Focus/blur on a window -** Theoretically, there are window.focus() and window.blur() methods to focus/unfocus on a window. Also there are focus/blur events that allow to focus a window and catch the moment when the visitor switches elsewhere.

In the past evil pages abused those. For instance, look at this code:

window.onblur = () => window.focus();

When a user attempts to switch out of the window (blur), it brings it back to focus. The intention is to “lock” the user within the window.

So, there are limitations that forbid the code like that. There are many limitations to protect the user from ads and evils pages. They depend on the browser.

For instance, a mobile browser usually ignores that call completely. Also focusing doesn’t work when a popup opens in a separate tab rather than a new window. Still, there are some things that can be done. For instance:

* When you open a popup, it’s might be a good idea to run a newWindow.focus() on it. Just in case, for some OS/browser combinations it ensures that the user is in the new window now.
* If you want to track when a visitor actually uses our web-app, you can track window.onfocus/onblur. That allows you to suspend/resume in-page activities, animations etc. But please note that the blur event means that the visitor switched out from the window, but they still may observe it. The window is in the background, but still may be visible.

**Day 2 – Cross-window communication**

**Same origin -** Two URLs are said to have the “same origin” if they have the same protocol, domain, and port. These URLs all share the same origin:

* http://site.com
* http://site.com/
* <http://site.com/my/page.html>

These ones do not:

* http://www.site.com (another domain: www.matters)
* http://site.org (another domain: .orgmatters)
* https://site.com (another protocol: https)
* http://site.com:8080 (another port: 8080)

The “Same Origin” policy states that:

If you have a reference to another window, e.g. a popup created by window.open or a window inside <iframe>, and that window comes from the same origin, then you have full access to that window. Otherwise, if it comes from another origin, then you can’t access the content of that window: variables, document, anything. The only exception is location: you can change it (thus redirecting the user). But you cannot read the location (so you can’t see where the user is now, no information leak).

**In action: iframe –** An <iframe> tag hosts a separate embedded window, with its own separate document and window objects. You can access them using properties:

* iframe.contentWindow to get the window inside the <iframe>.
* iframe.contentDocument to get the document inside the <iframe>, iframe.contentWindow.document.

When you access something inside the embedded window, the browser checks if the iframe has the same origin. If that’s not so then the access is denied (writing to location is an exception, it’s still permitted).

Iframe.onload vs iframe.contentWindow.onLoad - The iframe.onload event (on the <iframe>tag) is essentially the same as iframe.contentWindow.onload (on the embedded window object). It triggers when the embedded window fully loads with all resources. But you can’t access iframe.contentWindow.onload for an iframe from another origin, so use iframe.onload.

**Windows on subdomains: document.domain –** By definition, two URLs with different domains have different origins. But if windows share the same second-level domain, for instance, john.site.com, peter.site.com, and site.com (so that their common second-level domain is site.com), you can make the browser ignore that difference, so that they can be treated as coming from the “same origin” for the purposes of cross-window communication. To make it work, each such window should run the code: document.domain = ‘site.com’;

Now they can interact without limitations. Again, that’s only possible pages with the same second-level domain.

**Iframe: wrong document pitfall –** When an iframe comes from the same origin, and you may access its document, there’s a pitfall. It’s not related to cross-domain things, but important to know. Upon its creation, an iframe immediately has a document. But that document is different from the one that loads into it. So if you do something with the document immediately, that will probably be lost.

You shouldn’t work with the document of a not-yet-loaded iframe, because that’s the wrong document. If you set any event handlers on it, they will be ignored. The right document is definitely at the place when iframe.onload triggers. But it only triggers when the whole iframe with all resources is loaded.

**Collection: window.frames –** An alternative way to get a window object for <iframe> - is to get it from the named collectionwindow.frames:

* By number: window.frames[0] – the window object for the first frame in the document.
* By name: window.frames.iframeName – the window object for the frame withname="iframeName".

An iframe may have other iframes inside. The corresponding window objects form a hierarchy. Navigation links are:

* window.frames – the collection of “children” windows (for nested frames).
* window.parent – the reference to the “parent” (outer) window.
* window.top – the reference to the topmost parent window.

**The “sandbox” iframe attribute -** The sandbox attribute allows for the exclusion of certain actions inside an <iframe> in order to prevent it from executing untrusted code. It “sandboxes” the iframe by treating it as coming from another origin and/or applying other limitations. There’s a “default set” of restrictions applied for <iframe sandbox src="...">. But it can be relaxed if you provide a space-separated list of restrictions that should not be applied as a value of the attribute, like this: <iframe sandbox="allow-forms allow-popups">. In other words, an empty “sandbox” attribute puts the strictest limitations possible, but we can put a space-delimited list of those that we want to lift. Here’s a list of limitations:

* allow-same-origin - By default "sandbox" forces the “different origin” policy for the iframe. In other words, it makes the browser to treat the iframe as coming from another origin, even if its src points to the same site. With all implied restrictions for scripts. This option removes that feature.
* allow-top-navigation – Allows the iframe to change parent.location.
* allow-forms – Allows to submit forms from iframe.
* allow-scripts – Allows to run scripts form the iframe.
* allow-popups – Allows to window.open popups from the iframe.

**Cross Window Messaging -** The postMessage interface allows windows to talk to each other no matter which origin they are from. So, it’s a way around the “Same Origin” policy. It allows a window from john-smith.com to talk to gmail.com and exchange information, but only if they both agree and call corresponding JavaScript functions. That makes it safe for users. The interface has two parts.

* postMessage - The window that wants to send a message calls postMessage method of the receiving window. In other words, if we want to send the message to win, we should call win.postMessage(data, targetOrigin).

Arguments:

* data - The data to send. Can be any object, the data is cloned using the “structured cloning algorithm”. IE supports only strings, so we should JSON.stringify complex objects to support that browser.
* targetOrigin - Specifies the origin for the target window, so that only a window from the given origin will get the message. The targetOrigin is a safety measure. Remember, if the target window comes from another origin, we can’t read it’s location in the sender window. So we can’t be sure which site is open in the intended window right now: the user could navigate away, and the sender window has no idea about it. Specifying targetOrigin ensures that the window only receives the data if it’s still at the right site. Important when the data is sensitive.
* onmessage - To receive a message, the target window should have a handler on the message event. It triggers when postMessage is called (and targetOrigin check is successful).

The event object has special properties:

* data – The data from postMessage.
* origin – The origin of the sender, for instance httpp://javascript.info.
* source – The reference to the sender window. You can immediately source.postMessage(…) back if you want. To assign that handler, you should use addEventListener, a short syntax window.onmessage does not work.

**Day 3 – The clickjacking attack**

**Introduction to clickjacking –** Here’s how clickjacking was done with Facebook:

1. A visitor is lured to the evil page. It doesn’t matter how.
2. The page has a harmless-looking link on it (like “get rich now” or “click here, very funny”).
3. Over that link the evil page positions a transparent <iframe> with src from facebook.com, in such a way that the “Like” button is right above that link. Usually that’s done with z-index.
4. In attempting to click the link, the visitor in fact clicks the button.

Clickjacking is for clicks, not for keyboard. The attack only affects mouse actions (or similar, like taps on mobile). Keyboard input is much more difficult to redirect. Technically, if you have a text field to hack, then you can position an iframe in such a way that text fields overlap each other. So, when a visitor tries to focus on the input they see on the page, they actually focus on the input inside the iframe. But then there’s a problem. Everything that the visitor types will be hidden, because the iframe is not visible. People will usually stop typing when they can’t see their new characters printing on the screen.

**Old school defenses (weak) –** The oldest defense is a bit of JavaScript which forbids opening the page in a frame (so-called “framebusting”).

That looks like this:

If (top != window) {  
 top.location = window.location;

}

That is: if the window finds out that it’s not on top, then it automatically makes itself the top. This is not a reliable defense, because there are many ways to hack around it. Let’s cover a few.

Blocking top-navigation – You can block the transition caused by changing top.location in beforeunload event handler. The top page (enclosing one, belonging to the hacker) sets a preventing handler to it, like this:

Window.onbeforeunload = function() {

Return false;

};

When the iframe tries to change top.location, the visitor gets a message asking them whether they want to leave. In most cases the visitor would answer negatively because they don’t know about the iframe – all they can see is the top page, there’s no reason to leave. So top.location won’t change.

Sandbox attribute - One of the things restricted by the sandbox attribute is navigation. A sandboxed iframe may not change top.location. So you can add the iframe with sandbox="allow-scripts allow-forms". That would relax the restrictions, permitting scripts and forms. But you omit allow-top-navigation so that changing top.location is forbidden. Here’s the code:

<iframe sandbox="allow-scripts allow-forms" src="facebook.html"></iframe>

**X-frame-options -** The server-side header X-Frame-Options can permit or forbid displaying the page inside a frame. It must be sent exactly as HTTP-header: the browser will ignore it if found in HTML <meta> tag. So, <meta http-equiv="X-Frame-Options"...> won’t do anything. The header may have 3 values:

* DENY – Never ever show the page inside a frame.
* SAMEORIGIN – allow inside a frame if the parent document comes from the same origin.
* ALLOW-FROM domain – allow inside a frame if the parent document is from the given domain. For instance Twitter uses X-Frame-Options: SAMEORIGIN

Showing with disabled functionality - The X-Frame-Options header has a side-effect. Other sites won’t be able to show our page in a frame, even if they have good reasons to do so.

**Samesite cookie attribute –** The samesite cookie attribute can also prevent clickjacking attacks. A cookie with such attribute is only sent to a website if it’s opened directly, not via a frame, or otherwise. More information can be accessed using this link Cookies, document.cookie. If the site, such as Facebook, had samesite attribute on its authentication cookie, like this:

Set-Cookie: authorization=secret; samesite;

Then such cookie wouldn’t be sent when Facebook is open in iframe from another site. So the attack would fail.

The samesite cookie attribute will not have an effect when cookies are not used. This may allow other websites to easily show our public, unauthenticated pages in iframes.

However, this may also allow clickjacking attacks to work in a few limited cases. An anonymous polling website that prevents duplicate voting by checking IP addresses, for example, would still be vulnerable to clickjacking because it does not authenticate users using cookies.

**Day 4 – Arraybuffer and binary arrays**

**ArrayBuffer –** The basic binary object is ArrayBuffer – a reference to a fixed-length contiguous memory are. You create it like this:

let buffer = new ArrayBuffer(16); // create a buffer of length 16

alert(buffer.byteLength); // 16

This allocated a contiguous memory areaq of 16 bytes and pre-fills it with zeroes.

ArrayBuffer is not an array of something. ArrayBuffer has nothing in with Array:

* It has a fixed length, you can’t increase or decrease it.
* It takes exactly that much space in the memory.
* To access individual bytes, another “view” object is needed, not buffer[index].

ArrayBuffer is a memory area. What’s stored in it? It has no clue. Just a raw sequence of bytes. To manipulate an ArrayBuffer, you need to use a “view” object. A view object doesn’t store anything on its own. It’s the “eyeglasses” that give an interpretation of the bytes stored in the ArrayBuffer. For instance:

* Uint8Array – treats each byte in ArrayBuffer as a separate number, with possible values are from 0 to 255 (a byte is 8-bit, so it can hold only that much). Such value is called a “8-bit unsigned integer”.
* Uint16Array – treats every 2 bytes as an integer, with possible values from 0 to 65535. That’s called a “16-bit unsigned integer”.
* Uint32Array – treats every 4 bytes as an integer, with possible values from 0 to 4294967295. That’s called a “32-bit unsigned integer”.
* Float64Array – treats every 8 bytes as a floating point number with possible values from 5.0x10-324to 1.8x10308.

So, the binary data in an ArrayBuffer of 16 bytes can be interpreted as 16 “tiny numbers”, or 8 bigger numbers (2 bytes each), or 4 even bigger (4 bytes each), or 2 floating-point values with high precision (8 bytes each).

ArrayBuffer is the core object, the root of everything, the raw binary data.

**TypedArray -** The common term for all these views (Uint8Array, Uint32Array, etc) is TypedArray. They share the same set of methods and properties. They are much more like regular arrays: have indexes and iterable. A typed array constructor (be it Int8Array or Float64Array, doesn’t matter) behaves differently depending on argument types. There are 5 types of arguments:

* new TypedArray(buffer, [byteOffset], [length]);
* new TypedArray(object);
* new TypedArray(typedArray);
* new TypedArray(length);
* new TypedArray();

If an ArrayBuffer argument is supplied, the view is created over it. We used that syntax already. Optionally we can provide byteOffset to start from (0 by default) and the length (till the end of the buffer by default), then the view will cover only a part of the buffer.

If an Array, or any array-like object is given, it creates a typed array of the same length and copies the content. You can use it to pre-fill the array with the data:

let arr = new Uint8Array([0, 1, 2, 3]);

alert( arr.length ); // 4, created binary array of the same length

alert( arr[1] ); // 1, filled with 4 bytes (unsigned 8-bit integers) with given values

If another TypedArray is supplied, it does the same: creates a typed array of the same length and copies values. Values are converted to the new type in the process, if needed.

let arr16 = new Uint16Array([1, 1000]);

let arr8 = new Uint8Array(arr16);

alert( arr8[0] ); // 1

alert( arr8[1] ); // 232, tried to copy 1000, but can't fit 1000 into 8 bits (explanations below)

For a numeric argument length – creates the typed array to contain that many elements. Its byte length will be length multiplied by the number of bytes in a single item TypedArray.BYTES\_PER\_ELEMENT:

let arr = new Uint16Array(4); // create typed array for 4 integers

alert( Uint16Array.BYTES\_PER\_ELEMENT ); // 2 bytes per integer

alert( arr.byteLength ); // 8 (size in bytes)

Without arguments, creates an zero-length typed array.

You can create a TypedArray directly, without mentioning ArrayBuffer. But a view cannot exist without an underlying ArrayBuffer, so gets created automatically in all these cases except the first one (when provided). To access the ArrayBuffer, there are properties:

* arr.buffer - references the ArrayBuffer.
* arr.byteLength – the length of the ArrayBuffer.

So you can always move from one view to another:

let arr8 = new Uint8Array([0, 1, 2, 3]);

// another view on the same data

let arr16 = new Uint16Array(arr8.buffer);

Here’s the list of typed arrays:

* Uint8Array, Uint16Array, Uint32Array – for integer numbers of 8, 16 and 32 bits.
* Uint8ClampedArray – for 8-bit integers, “clamps” them on assignment (see below).
* Int8Array, Int16Array, Int32Array – for signed integer numbers (can be negative).
* Float32Array, Float64Array – for signed floating-point numbers of 32 and 64 bits.

No int8 or similar single-valued types. Please note, despite of the names like Int8Array, there’s no single-value type like int, or int8 in JavaScript. That’s logical, as Int8Array is not an array of these individual values, but rather a view on ArrayBuffer.

**Out-of-bounds behaviour -** What if you attempt to write an out-of-bounds value into a typed array? There will be no error. But extra bits are cut-off.

For instance, let’s try to put 256 into Uint8Array. In binary form, 256 is 100000000 (9 bits), but Uint8Array only provides 8 bits per value, which makes the available range from 0 to 255.

**TypedArray methods –** TypedArray has regular Array methods, with notable exceptions. We can iterate, map, slice, find, reduce, etc. There are a few things you can’t do though:

* No splice – we can’t “delete” a value because typed arrays are views on a buffer, and these are fixed, contiguous areas of memory. All we can do is to assign a zero.
* No concat method.

There are two additional methods:

* arr.set(fromArr, [offset]) copies all elements from fromArr to the arr, starting at position offset (0 by default).
* arr.subarray([begin, end]) creates a new view of the same type from begin to end (exclusive). That’s similar to slice method (that’s also supported), but doesn’t copy anything – just creates a new view, to operate on the given piece of data.

These methods allow you to copy typed arrays, mix them, create new arrays from existing ones, and so on.

**DataView -** DataView is a special super-flexible “untyped” view over ArrayBuffer. It allows accessing the data on any offset in any format.

* For typed arrays, the constructor dictates what the format is. The whole array is supposed to be uniform. The i-th number is arr[i].
* With DataView we access the data with methods like .getUint8(i) or .getUint16(i). We choose the format at method call time instead of the construction time.

The syntax: new DataView(buffer, [byteOffset], [byteLength])

* buffer – the underlying ArrayBuffer. Unlike typed arrays, DataView doesn’t create a buffer on its own. We need to have it ready.
* byteOffset – the starting byte position of the view (by default 0).
* byteLength – the byte length of the view (by default till the end of buffer).

DataView is great when you store mixed-format data in the same buffer. So, for example you store a sequence of pairs (16-bit integer, 32-bit float). Then DataView allows you to access them easily.