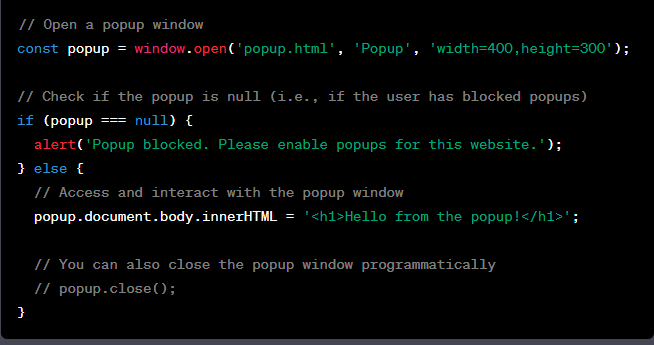
**Week 1: Events in JavaScript Day 1- Popups and window methods.**

**Popups:**

* **Alert**: The **alert()** method displays a simple dialog box with a message and an "OK" button. It is often used for displaying important messages to the user.
* javascriptCopy code
* alert("This is an alert message.");
* **Confirm**: The **confirm()** method shows a dialog box with a message and "OK" and "Cancel" buttons. It is used to ask the user for confirmation.
* javascriptCopy code
* if (confirm("Do you want to proceed?")) {  
   // User clicked OK  
  } else {  
   // User clicked Cancel  
  }
* **Prompt**: The **prompt()** method displays a dialog box with a message, an input field, and "OK" and "Cancel" buttons. It is used to collect user input.
* javascriptCopy code
* const userInput = prompt("Please enter your name:", "John Doe");
* **Window Methods**:
* **Open**: The **window.open()** method is used to open a new browser window or tab. It takes parameters for the URL to be loaded and window properties like size and position.
* javascriptCopy code
* const newWindow = window.open("https://www.example.com", "Example", "width=500,height=300");
* **Close**: The **window.close()** method is used to close the current browser window or tab. However, it can only be used to close windows that were opened using JavaScript.
* javascriptCopy code
* window.close();
* **Navigate**: You can use methods like **window.location** to navigate the current window to a new URL.
* javascriptCopy code
* window.location.href = "<https://www.example.com>";
* **Reload**: You can use **window.location.reload()** to reload the current page.
* javascriptCopy code
* window.location.reload();
* **Scroll**: Methods like **window.scrollTo()** and **window.scrollBy()** are used to scroll the content of the current window.
* javascriptCopy code
* window.scrollTo(0, 200); // Scroll to the top of the page

To access a popup window from the parent window in JavaScript, you can use the **window.open()** method to create the popup window, and then you can reference and interact with that popup window using the variable that holds a reference to it. Here's an



In the code above:

* We use **window.open()** to open a new popup window. The first argument is the URL of the popup, the second argument is the name of the popup window (which can be used as a reference), and the third argument is a string containing window features like width and height.
* We check if **popup** is **null**, which would happen if the user has blocked popups. If it's not **null**, we can interact with the popup window using the **popup** variable.
* Inside the **if** block, we set the content of the popup window's **document.body** to display a message.
* Optionally, you can close the popup window programmatically using **popup.close()**.

**To access the parent window from a popup window in JavaScript:**

you can use the **window.opener** property. This property provides a reference to the parent window that opened the popup. Here's an example of how to do this:

In the parent window (**parent.html**):

**In the parent window (parent.html):**

htmlCopy code

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Parent Window</title>  
 <script>  
 function openPopup() {  
 const popup = window.open('popup.html', 'Popup', 'width=400,height=300');  
 }  
  
 function showMessageFromPopup(message) {  
 alert(`Message from popup: ${message}`);  
 }  
 </script>  
</head>  
<body>  
 <button onclick="openPopup()">Open Popup</button>  
</body>  
</html>

**In the popup window (popup.html):**

htmlCopy code

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Popup Window</title>  
 <script>  
 function sendMessageToParent() {  
 const message = 'Hello from the popup!';  
   
 // Access the parent window and call a function in it  
 if (window.opener) {  
 window.opener.showMessageFromPopup(message);  
 }  
 }  
 </script>  
</head>  
<body>  
 <button onclick="sendMessageToParent()">Send Message to Parent</button>  
</body>  
</html>

In this example:

* The parent window (**parent.html**) contains a button that opens the popup window using the **window.open()** method.
* The popup window (**popup.html**) contains a button that, when clicked, sends a message to the parent window using the **window.opener** property. It calls the **showMessageFromPopup** function defined in the parent window.
* When the button in the popup is clicked, the **sendMessageToParent** function is called, which accesses the parent window through **window.opener** and calls the **showMessageFromPopup** function in the parent window to display the message in an alert.

This demonstrates how you can establish communication from a popup window to its parent window in JavaScript.

## **Closing a popup**

Use open() to open a window and close() to close the window:

<!DOCTYPE html>

<html>

<body>

<h1>The Window Object</h1>

<h2>The open() and close() Methods</h2>

<button onclick="openWin()">Open "myWindow"</button>

<button onclick="closeWin()">Close "myWindow"</button>

<script>

let myWindow;

function openWin() {

myWindow = window.open("", "", "width=200,height=100");

}

function closeWin() {

myWindow.close();

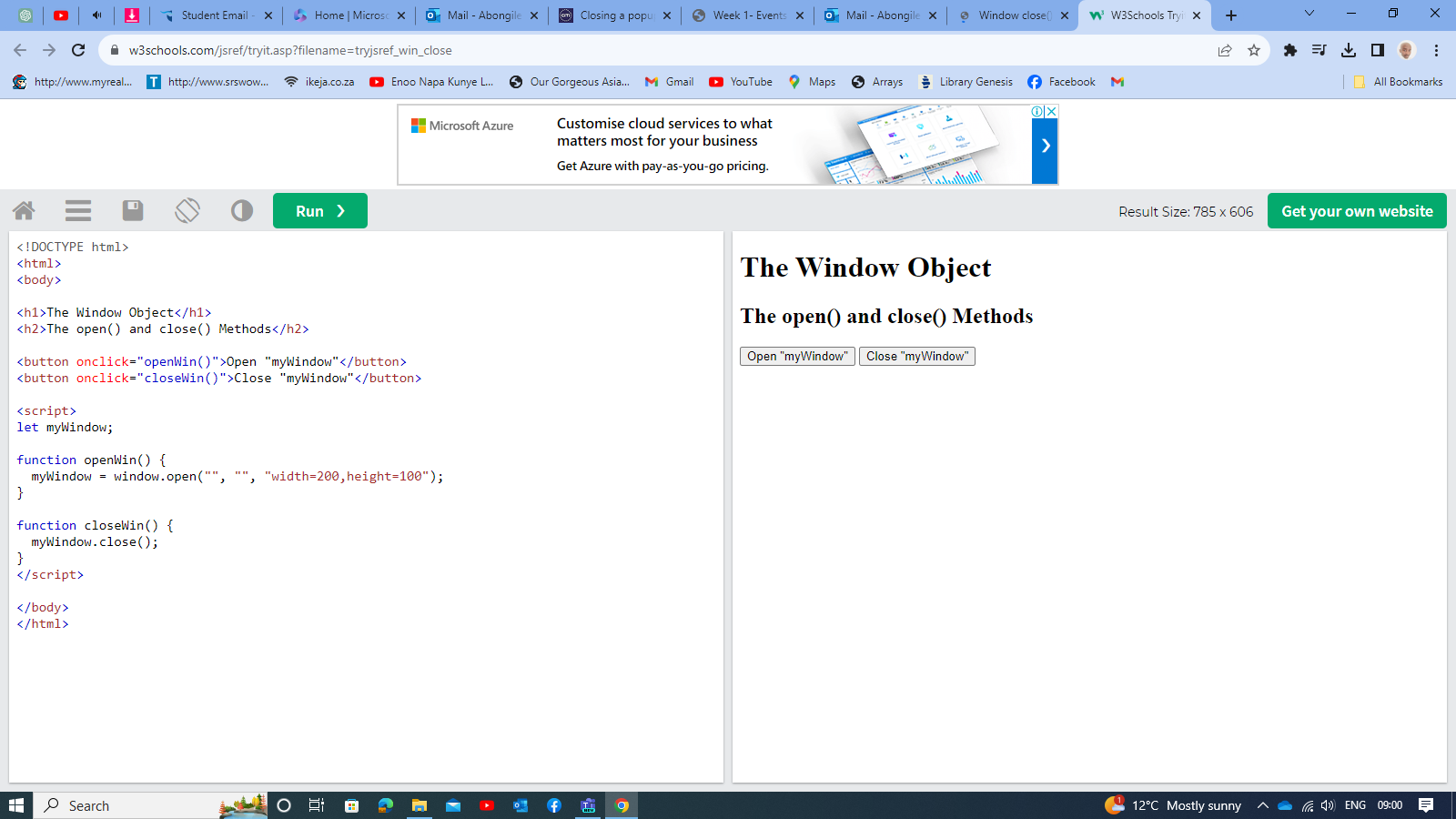
}

</script>

</body>

</html>

**Code for opening and closing a window:**



## **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.

We already talked about scrolling a window in the chapter [Window sizes and scrolling](https://javascript.info/size-and-scroll-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.

**Javascript Window Blur():**

Javascript Window Blur() method is used to remove focus from the current window

# **Cross-window communication**

The “Same Origin” (same site) policy limits access of windows and frames to each other.

The idea is that if a user has two pages open: one from john-smith.com, and another one is gmail.com, then they wouldn’t want a script from john-smith.com to read our mail from gmail.com. So, the purpose of the “Same Origin” policy is to protect users from information theft.

## [Same Origin](https://javascript.info/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](http://www.site.com) (another domain: www. matters)
* [http://**site.org**](http://site.org) (another domain: .org matters)
* [**https://**site.com](https://site.com) (another protocol: https)
* [http://site.com:**8080**](http://site.com:8080) (another port: 8080)

**The “Same Origin” policy states that:**

* if we 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 we have full access to that window.
* otherwise, if it comes from another origin, then we can’t access the content of that window: variables, document, anything. The only exception is location: we can change it (thus redirecting the user). But we cannot *read* location (so we can’t see where the user is now, no information leak).

### [**In action: iframe**](https://javascript.info/cross-window-communication#in-action-iframe)

An <iframe> tag hosts a separate embedded window, with its own separate document and window objects.

We can access them using properties:

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

When we 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).

For instance, let’s try reading and writing to <iframe> from another origin:

<iframe src="https://example.com" id="iframe"></iframe>

<script>  
 iframe.onload = function() {  
 // we can get the reference to the inner window  
 let iframeWindow = iframe.contentWindow; // OK  
 try {  
 // ...but not to the document inside it  
 let doc = iframe.contentDocument; // ERROR  
 } catch(e) {  
 alert(e); // Security Error (another origin)  
 }  
  
 // also we can't READ the URL of the page in iframe  
 try {  
 // Can't read URL from the Location object  
 let href = iframe.contentWindow.location.href; // ERROR  
 } catch(e) {  
 alert(e); // Security Error  
 }  
 // ...we can WRITE into location (and thus load something else into the iframe)!  
 iframe.contentWindow.location = '/'; // OK  
  
 iframe.onload = null; // clear the handler, not to run it after the location change  
 };  
</script>

The code above shows errors for any operations except:

* Getting the reference to the inner window iframe.contentWindow – that’s allowed.
* Writing to location.

Contrary to that, if the <iframe> has the same origin, we can do anything with it:

<!-- iframe from the same site -->  
<iframe src="/" id="iframe"></iframe>  
  
<script>  
 iframe.onload = function() {  
 // just do anything  
 iframe.contentDocument.body.prepend("Hello, world!");  
 };  
</script>

**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 we can’t access iframe.contentWindow.onload for an iframe from another origin, so using iframe.onload.

## [Windows on subdomains: document.domain](https://javascript.info/cross-window-communication#windows-on-subdomains-document-domain)

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), we 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';

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

**Deprecated, but still working**

The document.domain property is in the process of being removed from the [specification](https://html.spec.whatwg.org/multipage/origin.html#relaxing-the-same-origin-restriction). The cross-window messaging (explained soon below) is the suggested replacement.

That said, as of now all browsers support it. And the support will be kept for the future, not to break old code that relies on document.domain.

## [Iframe: wrong document pitfall](https://javascript.info/cross-window-communication#iframe-wrong-document-pitfall)

When an iframe comes from the same origin, and we may access its document, there’s a pitfall. It’s not related to cross-origin 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 we do something with the document immediately, that will probably be lost.

Here, look:

<iframe src="/" id="iframe"></iframe>  
  
<script>  
 let oldDoc = iframe.contentDocument;  
 iframe.onload = function() {  
 let newDoc = iframe.contentDocument;  
 // the loaded document is not the same as initial!  
 alert(oldDoc == newDoc); // false  
 };  
</script>

We shouldn’t work with the document of a not-yet-loaded iframe, because that’s the *wrong document*. If we set any event handlers on it, they will be ignored.

How to detect the moment when the document is there?

The right document is definitely at place when iframe.onload triggers. But it only triggers when the whole iframe with all resources is loaded.

We can try to catch the moment earlier using checks in setInterval:

<iframe src="/" id="iframe"></iframe>  
  
<script>  
 let oldDoc = iframe.contentDocument;  
  
 // every 100 ms check if the document is the new one  
 let timer = setInterval(() => {  
 let newDoc = iframe.contentDocument;  
 if (newDoc == oldDoc) return;  
  
 alert("New document is here!");  
  
 clearInterval(timer); // cancel setInterval, don't need it any more  
 }, 100);  
</script>

## [Collection: window.frames](https://javascript.info/cross-window-communication#collection-window-frames)

An alternative way to get a window object for <iframe> – is to get it from the named collection window.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 with name="iframeName".

For instance:

<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>  
  
<script>  
 alert(iframe.contentWindow == frames[0]); // true  
 alert(iframe.contentWindow == frames.win); // true  
</script>

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.

For instance:

window.frames[0].parent === window; // true

We can use the top property to check if the current document is open inside a frame or not:

if (window == top) { // current window == window.top?  
 alert('The script is in the topmost window, not in a frame');  
} else {  
 alert('The script runs in a frame!');  
}

## [The “sandbox” iframe attribute](https://javascript.info/cross-window-communication#the-sandbox-iframe-attribute)

The sandbox attribute allows for the exclusion of certain actions inside an <iframe> in order to prevent it 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 we 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 from the iframe.

**allow-popups**

Allows to window.open popups from the iframe

See [the manual](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe) for more.

The example below demonstrates a sandboxed iframe with the default set of restrictions: <iframe sandbox src="...">. It has some JavaScript and a form.

Please note that nothing works. So, the default set is harsh:

**The purpose of the "sandbox"** attribute is only to *add more* restrictions. It cannot remove them. It cannot relax same-origin restrictions if the iframe comes from another origin.

## [Cross-window messaging](https://javascript.info/cross-window-communication#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](https://javascript.info/cross-window-communication#postmessage)

The window that wants to send a message calls [postMessage](https://developer.mozilla.org/en-US/docs/Web/API/Window.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 serialization 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 its 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.

For instance, here win will only receive the message if it has a document from the origin <http://example.com>:

<iframe src="http://example.com" name="example">  
  
<script>  
 let win = window.frames.example;  
  
 win.postMessage("message", "<http://example.com>");  
</script>

If we don’t want that check, we can set targetOrigin to \*.

<iframe src="http://example.com" name="example">  
  
<script>  
 let win = window.frames.example;  
  
 win.postMessage("message", "\*");  
</script>

### [onmessage:](https://javascript.info/cross-window-communication#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 <http://javascript.info>.

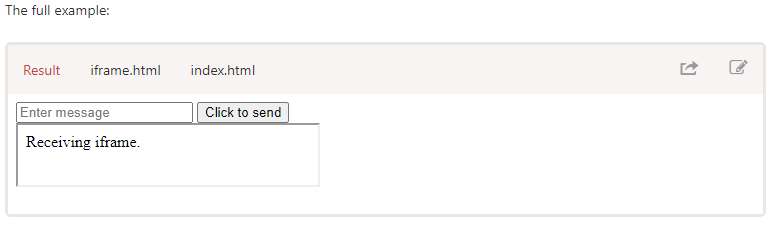
**source**

The reference to the sender window. We can immediately source.postMessage(...) back if we want.

To assign that handler, we should use addEventListener, a short syntax window.onmessage does not work.

Here’s an example:

window.addEventListener("message", function(event) {  
 if (event.origin != '<http://javascript.info>') {  
 // something from an unknown domain, let's ignore it  
 return;  
 }  
  
 alert( "received: " + event.data );  
  
 // can message back using event.source.postMessage(...)  
});



# **The clickjacking attacks**

The “clickjacking” attack allows an evil page to click on a “victim site” *on behalf of the visitor*.

Many sites were hacked this way, including Twitter, Facebook, Paypal and other sites. They have all been fixed, of course.

## [The idea](https://javascript.info/clickjacking#the-idea)

The idea is very simple.

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.

Example:



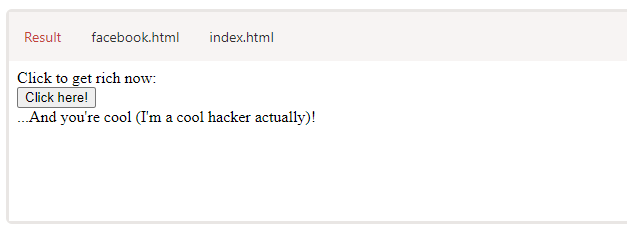
The full demo of the attack:



Here we have a half-transparent <iframe src="facebook.html">, and in the example we can see it hovering over the button. A click on the button actually clicks on the iframe, but that’s not visible to the user, because the iframe is transparent.

As a result, if the visitor is authorized on Facebook (“remember me” is usually turned on), then it adds a “Like”. On Twitter that would be a “Follow” button.

Here’s the same example, but closer to reality, with opacity:0 for <iframe>:



All we need to attack – is to position the <iframe> on the evil page in such a way that the button is right over the link. So that when a user clicks the link, they actually click the button. That’s usually doable with CSS (Cascading Style Sheets).

**Clickjacking is for clicks, not for keyboard**

The attack only affects mouse actions (or similar, like taps on mobile).

Keyboard input is much difficult to redirect. Technically, if we have a text field to hack, then we 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.

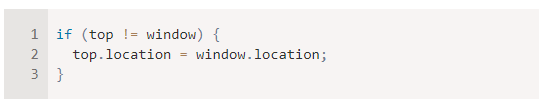
But then there’s a problem. Everything that the visitor types will be hidden, because the iframe is not visible.

People will usually stop typing when they can’t see their new characters printing on the screen.

**Old school defences (weak)**

The oldest defence is a bit of JavaScript which forbids opening the page in a frame (so-called “framebusting”).

That looks like this:



That is: if the window finds out that it’s not on top, then it automatically makes itself the top.

This not a reliable defence, because there are many ways to hack around it. Let’s cover a few.

### [Blocking top-navigation](https://javascript.info/clickjacking#blocking-top-navigation)

We can block the transition caused by changing top.location in [beforeunload](https://javascript.info/onload-ondomcontentloaded#window.onbeforeunload) event handler.

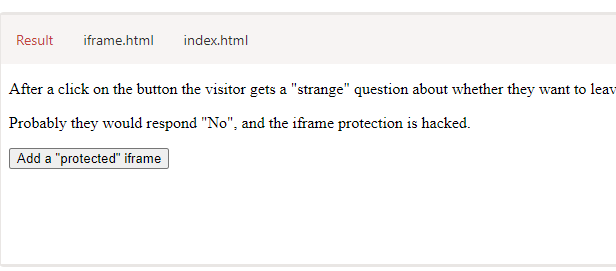
The top page (enclosing one, belonging to the hacker) sets a preventing handler to it, like this:

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!

In action:



### [Sandbox attribute](https://javascript.info/clickjacking#sandbox-attribute)

One of the things restricted by the sandbox attribute is navigation. A sandboxed iframe may not change top.location.

So we can add the iframe with sandbox="allow-scripts allow-forms". That would relax the restrictions, permitting scripts and forms. But we 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>

There are other ways to work around that simple protection too.

## [X-Frame-Options](https://javascript.info/clickjacking#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.

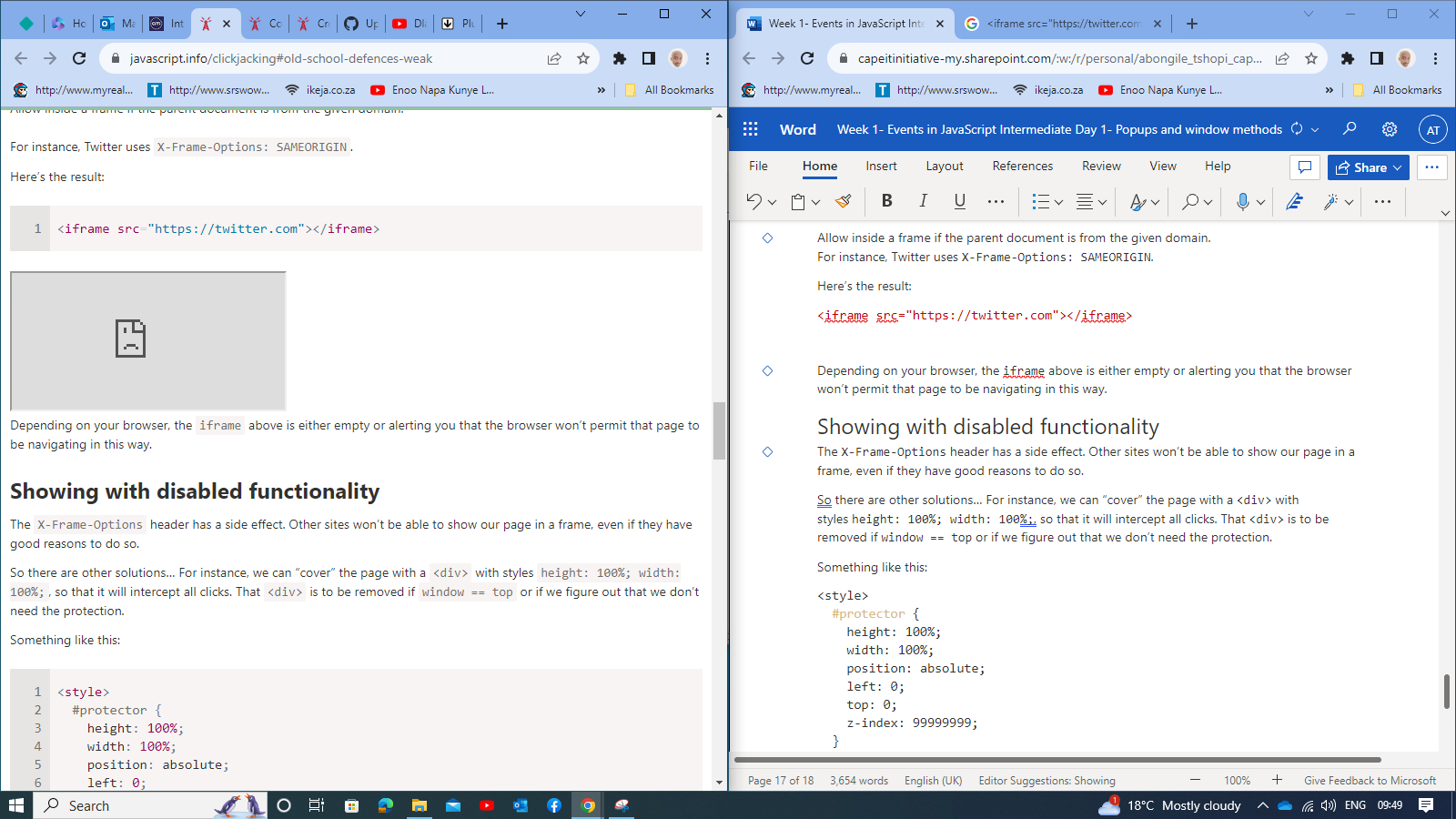
**ALLOW-FROM domain**

Allow inside a frame if the parent document is from the given domain.

For instance, Twitter uses X-Frame-Options: SAMEORIGIN.

Here’s the result:

<iframe src="https://twitter.com"></iframe>



Depending on your browser, the iframe above is either empty or alerting you that the browser won’t permit that page to be navigating in this way.

## [Showing with disabled functionality](https://javascript.info/clickjacking#showing-with-disabled-functionality)

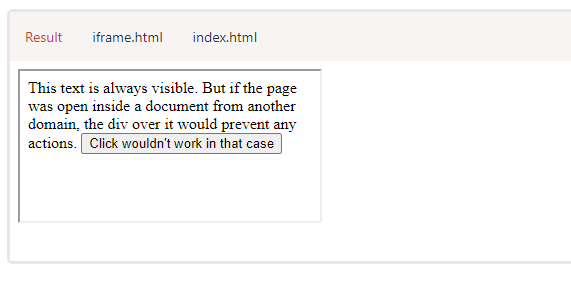
The X-Frame-Options header has a side effect. Other sites won’t be able to show our page in a frame, even if they have good reasons to do so.

So there are other solutions… For instance, we can “cover” the page with a <div> with styles height: 100%; width: 100%;, so that it will intercept all clicks. That <div> is to be removed if window == top or if we figure out that we don’t need the protection.

Something like this:

<style>  
 #protector {  
 height: 100%;  
 width: 100%;  
 position: absolute;  
 left: 0;  
 top: 0;  
 z-index: 99999999;  
 }  
</style>  
  
<div id="protector">  
 <a href="/" target="\_blank">Go to the site</a>  
</div>  
  
<script>  
 // there will be an error if top window is from the different origin  
 // but that's ok here  
 if (top.document.domain == document.domain) {  
 protector.remove();  
 }  
</script>

**The demo:**



## [Samesite cookie attribute](https://javascript.info/clickjacking#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 in the chapter [Cookies, document.cookie](https://javascript.info/cookie#samesite).

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.

# **ArrayBuffer, binary arrays**

In web-development we meet binary data mostly while dealing with files (create, upload, download). Another typical use case is image processing.

That’s all possible in JavaScript, and binary operations are high-performant.

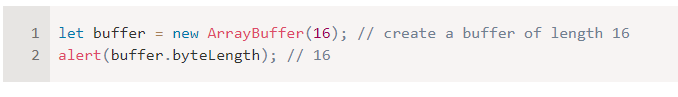
Although, there’s a bit of confusion, because there are many classes. To name a few:

* ArrayBuffer, Uint8Array, DataView, Blob, File, etc.

Binary data in JavaScript is implemented in a non-standard way, compared to other languages. But when we sort things out, everything becomes fairly simple.

**The basic binary object is ArrayBuffer – a reference to a fixed-length contiguous memory area.**

We create it like this:



**ArrayBuffer is not an array of something**

Let’s eliminate a possible source of confusion. ArrayBuffer has nothing in common with Array:

* It has a fixed length, we 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.

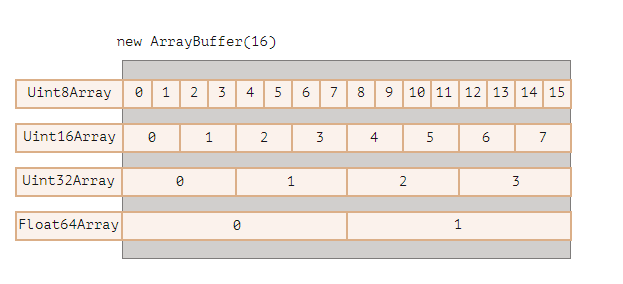
**To manipulate an ArrayBuffer, we need to use a “view” object.**

A view object does not 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 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 5.0x10-324 to 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.

But if we’re going to write into it, or iterate over it, basically for almost any operation – we must use a view, e.g:

let buffer = new ArrayBuffer(16); // create a buffer of length 16  
  
let view = new Uint32Array(buffer); // treat buffer as a sequence of 32-bit integers  
  
alert(Uint32Array.BYTES\_PER\_ELEMENT); // 4 bytes per integer  
  
alert(view.length); // 4, it stores that many integers  
alert(view.byteLength); // 16, the size in bytes  
  
// let's write a value  
view[0] = 123456;  
  
// iterate over values  
for(let num of view) {  
 alert(num); // 123456, then 0, 0, 0 (4 values total)  
}

## [TypedArray](https://javascript.info/arraybuffer-binary-arrays#typedarray)

The common term for all these views (Uint8Array, Uint32Array, etc) is [TypedArray](https://tc39.github.io/ecma262/#sec-typedarray-objects). They share the same set of methods and properties.

Please note, there’s no constructor called TypedArray, it’s just a common “umbrella” term to represent one of views over ArrayBuffer: Int8Array, Uint8Array and so on, the full list will soon follow.

When you see something like new TypedArray, it means any of new Int8Array, new Uint8Array, etc.

Typed arrays behave like regular arrays: have indexes and are iterable.

A typed array constructor (be it Int8Array or Float64Array, doesn’t matter) behaves differently depending on argument types.

There are 5 variants of arguments:

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

1. If an ArrayBuffer argument is supplied, the view is created over it. We used that syntax already.
2. 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.
3. If an Array, or any array-like object is given, it creates a typed array of the same length and copies the content.
4. We can use it to pre-fill the array with the data:
5. 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
6. 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.
7. 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)
8. 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:
9. 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)
10. Without arguments, creates an zero-length typed array.

We 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 underlying ArrayBuffer, there are following properties in TypedArray:

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

So, we 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 behavior](https://javascript.info/arraybuffer-binary-arrays#out-of-bounds-behavior)

What if we 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, that makes the available range from 0 to 255.

For bigger numbers, only the rightmost (less significant) 8 bits are stored, and the rest is cut off:

So we’ll get zero.

For 257, the binary form is 100000001 (9 bits), the rightmost 8 get stored, so we’ll have 1 in the array:

In other words, the number modulo 28 is saved.

Here’s the demo:

let uint8array = new Uint8Array(16);  
  
let num = 256;  
alert(num.toString(2)); // 100000000 (binary representation)  
  
uint8array[0] = 256;  
uint8array[1] = 257;  
  
alert(uint8array[0]); // 0  
alert(uint8array[1]); // 1

Uint8ClampedArray is special in this aspect, its behavior is different. It saves 255 for any number that is greater than 255, and 0 for any negative number. That behavior is useful for image processing.

## [TypedArray methods](https://javascript.info/arraybuffer-binary-arrays#typedarray-methods)

TypedArray has regular Array methods, with notable exceptions.

We can iterate, map, slice, find, reduce etc.

There are few things we 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 us to copy typed arrays, mix them, create new arrays from existing ones, and so on.

## [DataView](https://javascript.info/arraybuffer-binary-arrays#dataview)

[DataView](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView) is a special super-flexible “untyped” view over ArrayBuffer. It allows to access 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).

For instance, here we extract numbers in different formats from the same buffer:

// binary array of 4 bytes, all have the maximal value 255  
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;  
  
let dataView = new DataView(buffer);  
  
// get 8-bit number at offset 0  
alert( dataView.getUint8(0) ); // 255  
  
// now get 16-bit number at offset 0, it consists of 2 bytes, together interpreted as 65535  
alert( dataView.getUint16(0) ); // 65535 (biggest 16-bit unsigned int)  
  
// get 32-bit number at offset 0  
alert( dataView.getUint32(0) ); // 4294967295 (biggest 32-bit unsigned int)  
  
dataView.setUint32(0, 0); // set 4-byte number to zero, thus setting all bytes to 0

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