**Popups and window method.**

Popup has 3 commonly used popup boxes for interacting with users, the alert box, confirm box and prompt box.

**Alert Box**:

* The **alert ()** method is used to display an alert box with a message to the user.
* It typically has an "OK" button, and it's used to provide information or display a message to the user.

**Confirm Box:**

* The confirm () method displays a dialog box with a message and two buttons: "OK" and "Cancel."
* It's used when you need the user to confirm or cancel an action.
* It returns true if the user clicks "OK" and false if they click "Cancel."

**Prompt Box**:

* The prompt ()
* method displays a dialog box with a message, an input field for the user to enter data, and "OK" and "Cancel" buttons.
* It's used when you need to get input from the user.
* It returns the text entered by the user when they click "OK" or null if they click "Cancel" or close the dialog.

**Window Methods**:

**window.close()**:

* The window.close() method is used to close the currently focused browser window or tab.

**window.location**:

* The window.location object is used to get or set the URL of the current browser window.

**window.reload()**:

* The window.reload() method is used to reload the current page.

These are some of the common methods and techniques for working with popups and manipulating browser windows in JavaScript. They are often used for various user interactions and navigation within web applications.

**Popup blocking**

Popup blocking is a feature in web browsers that prevents unwanted or potentially malicious pop-up windows from opening automatically when you visit a website.

**Automatic Blocking**: Most modern web browsers have built-in popup blockers. These blockers automatically detect and prevent pop-up windows from appearing when you visit a website. This helps prevent annoying and potentially harmful popups.

**User Interaction**: Browsers usually allow popups that are triggered by user interactions like clicking a link or a button. These popups are considered "user-initiated," and the browser allows them to open.

**Exceptions**: Users can often configure their browser settings to allow popups from specific websites or domains, especially when they trust the site. This is useful for websites that legitimately use popups for functionality, such as online banking or e-commerce sites.

**Managing Popup Blocking**:

**Browser Settings**: You can control popup blocking in your browser's settings.

**Exceptions**: In browser settings, you can usually add exceptions for specific websites that you trust. This allows popups on those sites while still blocking them elsewhere.

**Extensions**: Some browser extensions or add-ons provide additional control over popup blocking. You can install and configure these extensions to customize your popup blocking experience.

**Browser Alerts**: When a popup is blocked, browsers often display an icon or alert in the address bar to inform you. You can click on this alert to manage popup settings for the current website.

**Resetting Preferences**: If you encounter issues with popups, you can reset your browser's settings to the default state, which usually reverts popup blocking settings to their original state.

**On window. open.**

The window. open () method in JavaScript is used to open a new browser window or tab with the specified URL and additional configuration options.

The **window. open ()** method in JavaScript is used to open a new browser window or tab with the specified URL and additional configuration options. Here's a breakdown of the syntax and the available configuration settings:

**Syntax**:

javascriptCopy code

window. open (url, name, params);

**Parameters**:

1. **url**: The URL of the web page or resource you want to load into the new window.
2. **name**: A name for the new window. If a window with this name already exists, the URL will open in that window; otherwise, a new window is created.
3. **params**: A configuration string containing settings for the new window. The settings are delimited by commas and have no spaces. Here are some commonly used settings.

* **left**/**top** (numeric): The coordinates for the top-left corner of the new window on the screen
* **width**/**height** (numeric): The width and height of the new window.

**Window Features**

* menubar: Shows or hides the browser menu in the new window.
* toolbar: Shows or hides the browser navigation bar (back, forward, reload, etc.) in the new window.
* location: Shows or hides the URL field in the new window. Note that some browsers don't allow hiding it.
* status: Shows or hides the status bar in the new window. Most browsers force it to show.
* resizable: Allows or disallows resizing the new window. Not recommended to disable.
* scrollbars: Allows or disallows scrollbars in the new window. Not recommended to disable

**a minimalistic window**

Creating a minimalistic window in JavaScript involves using the **window.open()** method with specific parameters to control its appearance and behaviour.

**Focus or blur on a window.**

JavaScript provides window.focus() and window.blur() methods to programmatically set the focus on a window or remove focus from it, respectively.

**Focus and Blur Events:**

* The **focus** and **blur** events allow you to detect when a window gains or loses focus.
* You can use these events to track user interactions and adjust the behavior of your web application accordingly.

Abuse Prevention

In the past, malicious websites attempted to abuse focus and blur events to trap users within their pages. For example, using window.onblur to force focus back on the window.

To protect users, modern browsers have implemented limitations and restrictions on these methods and events. Some browsers may ignore or restrict the behavior of **focus()** and **blur()** calls.

**Cross-Browser and Cross-Device Variations:**

* The behavior of these methods and events can vary across different browsers and devices.
* For instance, mobile browsers may handle focus and blur events differently, and focusing doesn't work the same way for popups in separate tabs compared to new windows.

You can use **window.onfocus** and **window.onblur** events to track user activity and suspend/resume in-page activities or animations.

**Same origin**

Same Origin means that two web pages or resources come from exactly the same website, including having the same protocol (like "http" or "https"), domain (like "site.com"), and port (like ":80" for HTTP).

Examples of URLs with the same origin:

http://site.com

http://site.com/

http://site.com/my/page.html

Same Origin Policy is a security rule in web browsers that says:

If you have a reference to another window (e.g., a popup or an iframe) that comes from the same origin, you have full access to that window. You can interact with its content, access its variables, and manipulate its documents.

If the other window comes from a different origin, you can't access its content, variables, or document due to security restrictions. However, you can change the location to redirect the user to another page.

Examples:

You can freely interact with a popup or iframe that comes from the same website.

You can't directly access or manipulate the content of a popup or iframe if it's from a different website to prevent security issues, except for changing its location.

**Iframes**

Accessing iframe Properties from Different Origins:

When you have an iframe from a different origin (i.e., a different website), you are limited in what you can do with it due to the Same Origin Policy.

You can access the reference to the inner window of the iframe using iframe. contentWindow. This is allowed because it doesn't expose the content of the iframe to your parent window directly.

You can write to the location property of the iframe to change its URL (i.e., perform a redirect). This is a special exception, and it's permitted even for iframes from different origins.

iframe. onload vs. iframe. contentWindow.onload:

iframe. onload is an event that is triggered on the <iframe> element itself. It fires when the iframe's initial HTML structure and external resources (like images and scripts) have finished loading. It doesn't wait for the content inside the iframe to load.

iframe. contentWindow.onload is an event that you can attach to the window object within the iframe (accessed through iframe. contentWindow). It fires when the content of the iframe, including all its resources, has fully loaded.

If the iframe and the parent window have the same origin, you can use iframe. onload or iframe. contentWindow.onload to track when the iframe is fully loaded. You can also access and manipulate the content inside the iframe freely.

If the iframe is from a different origin, you can only use iframe. onload to detect when it has loaded. You won't be able to access or manipulate the content inside the iframe due to the Same Origin Policy, except for changing its location.

**Subdomain and domains**

In the world of web browsers, there's a security feature called the "same-origin policy." It's like a set of rules that helps protect your data and privacy when you visit different websites. According to this policy, web pages from different domains (the main part of a web address, like google.com and yahoo.com) are usually not allowed to talk to each other directly in a web browser. This restriction is in place to prevent one website from messing with or stealing data from another website that you might have open in another tab.

However, there's a way to bypass this restriction if you have a specific setup. Imagine you have three web pages: john.site.com, peter.site.com, and site.com. In this case, "site.com" is the common second-level domain they share. Second-level domain means the part just before ".com" or ".org" in a web address.

**Document pitfall**

when an iframe is created, it immediately has a document associated with it, but that document may not be the one that will load its content. This is an important consideration when working with iframes in JavaScript. To ensure that you are working with the correct document, you should wait for the iframe's onload event to fire. This event indicates that the iframe and all its resources have been fully loaded, and you can safely access its document at that point.

we wait for the **onload** event of the iframe to fire before accessing its document. This ensures that you are working with the correct document that has loaded inside the iframe, and any event handlers or manipulations will take effect as expected.

**Window.frames.**

The window. frames property is an array-like object in JavaScript that provides access to all the iframes (inline frames) within the current window or document. It allows you to interact with and manipulate the content of iframes on a web page.

You can use window. frames to access individual iframes by their index, similar to how you access elements in an array.

Once you have a reference to an iframe using **window.frames**, you can interact with its properties and methods just like you would with any other window or document object.

**The sand box.**

The sandbox attribute in HTML <iframe> elements is used to apply restrictions on what an iframe can do in order to enhance security. By default, when you use the sandbox attribute without any values like this: <iframe sandbox src="...">, it imposes the strictest limitations possible on the iframe, effectively treating it as if it's from a different origin, even if it's from the same site. This means that scripts, form submissions, and popups are all disabled within the iframe.

You can selectively relax these restrictions by specifying certain values in the sandbox attribute, separated by spaces. For example, <iframe sandbox="allow-forms allow-popups"> would enable form submissions and the ability to open pop-up windows within the iframe

**Cross window messaging**

The postMessage interface in JavaScript allows different windows, even from different origins (websites), to communicate with each other safely. This is a way to get around the "Same Origin" policy, which typically restricts communication between different websites.

This is when you sending the message. If one window (let's call it Window A) wants to send a message to another window (Window B), it can use the postMessage method. For example, Window A can call win.postMessage(data, targetOrigin).

* data: This is the information you want to send. It can be any JavaScript object, and it gets sent securely.
* targetOrigin: This specifies the origin (website) of the target window (Window B). It ensures that only the intended window receives the message. This is a security measure to prevent unauthorized access.

For example, you can use targetOrigin to make sure that the message is only received by a window from a specific origin, like <http://example.com>.

This when you receiving the message. The target window B should have an event for the message event. This event handler triggers when the post message is called, and the target origin check is successfully.

postMessage allows different windows to communicate securely, ensuring that only the intended recipient receives and processes the messages. It's a powerful tool for creating interactive web applications that need to share data or coordinate actions between different parts of a web page or even across different websites.

**Clickjacking**

Clickjacking is a deceptive technique where an attacker tricks a user into clicking on something different from what they perceive.

1. Luring the User: The attacker gets a user to visit their malicious web page through various means.
2. Transparent Overlay: On the malicious page, the attacker places a transparent <iframe> element, typically from a trusted website like Facebook or Twitter, on top of a harmless-looking link or button.
3. Deceptive Appearance: The overlay is positioned in such a way that it appears the user is clicking on the harmless link/button, but they're actually clicking on the invisible <iframe>.
4. Mouse Actions: When the user clicks, it triggers an action within the overlay, such as liking a post on Facebook or following someone on Twitter.
5. Limitations: Clickjacking primarily affects mouse actions, not keyboard input. While it's theoretically possible to use a similar technique to redirect keyboard input, it's much more challenging because the user can't see what they're typing if it's redirected to an invisible overlay.

**Old school defenses.**

Old-school defenses against clickjacking were often simple and somewhat weak compared to modern security measures. Here are some of the common old-school defenses, explained in simple terms:

1. Frame Busting: This is a technique where a website uses JavaScript to check if it's being displayed within an <iframe>. If it detects that it is, it tries to break out of the frame by changing the top-level location. This approach was not foolproof because attackers could find ways to bypass it.
2. X-Frame-Options Header: Some websites used a security header called X-Frame-Options to specify whether their pages should be allowed to be displayed within an <iframe>. It had options like "DENY" (never allow framing) and "SAMEORIGIN" (only allow framing from the same origin). However, it required support from the browser, and not all browsers implemented it consistently.
3. Frame killer Scripts: Website developers sometimes included scripts on their pages that attempted to break out of frames created by attackers. These scripts often relied on JavaScript alerts or other tricks to disrupt the attacker's framing attempt. However, these were not always effective, and attackers could find ways to bypass them.
4. Content Security Policy (CSP): CSP is a security feature that allows websites to specify which sources of content are trusted. While CSP was not specifically designed to prevent clickjacking, it could be used to mitigate the risk by controlling where content could be loaded from.

These old-school defenses had limitations and could be bypassed by determined attackers. As a result, they were not as effective as modern security measures like the Content-Security-Policy header, which provides more robust protection against clickjacking and other web security threats.

**SAMe site cookie attribute**

The X-Frame-Options server-side header is a security feature used to control whether a web page can be displayed within an <iframe> on another website. Here's an explanation of how it works and its possible values:

1. DENY: If a website sends the X-Frame-Options header with the value "DENY," it means the page should never be displayed inside an <iframe> on any other website. This is the strictest option and prevents framing entirely.
2. SAMEORIGIN**:** When the header is set to "SAMEORIGIN," it allows the page to be displayed inside an <iframe> only if the parent document (the one containing the <iframe>) comes from the same origin (i.e., the same website). This provides a level of protection against clickjacking while allowing legitimate embedding within the same website.
3. ALLOW-FROM domain: With this option, you specify a specific domain (e.g., "[https://example.com](https://example.com/)") in the header. The page can be displayed within an <iframe> on a website from the specified domain. It allows more flexibility compared to "SAMEORIGIN" but restricts framing to the specified domain.

**An Array buffer.**

An ArrayBuffer in JavaScript is a built-in object used to represent a fixed-length, binary data buffer. It is essentially a container for raw binary data, and it provides a way to work with binary data directly in memory.

**Typed Array**

Typed Arrays in JavaScript are a group of built-in objects that provide a way to work with binary data in a structured manner. They are used in conjunction with ArrayBuffer objects and offer more efficient and precise manipulation of binary data compared to traditional JavaScript arrays.

**Out of bound behavior.**

accessing array elements or typed array elements outside of their bounds (i.e., trying to access an index that doesn't exist) does not throw an error or crash your program as it might in some other programming languages. Instead, JavaScript exhibits specific behavior when you attempt to access elements out of bounds.