Lesson:

Window Object

PW SKILLS

Topics:

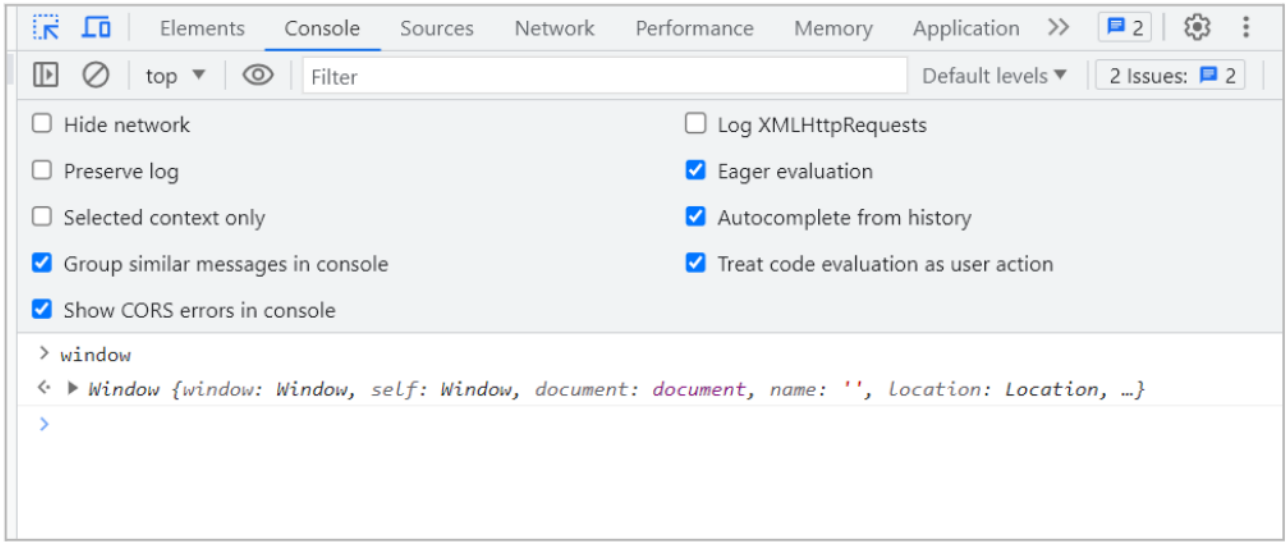
- Introduction to the Window object
- Window properties (screen & location)
- Cookies storage
- SetTimeout
- ClearTimeout
- SetInterval
- ClearInterval
- Alert
- Prompt
- Confirm

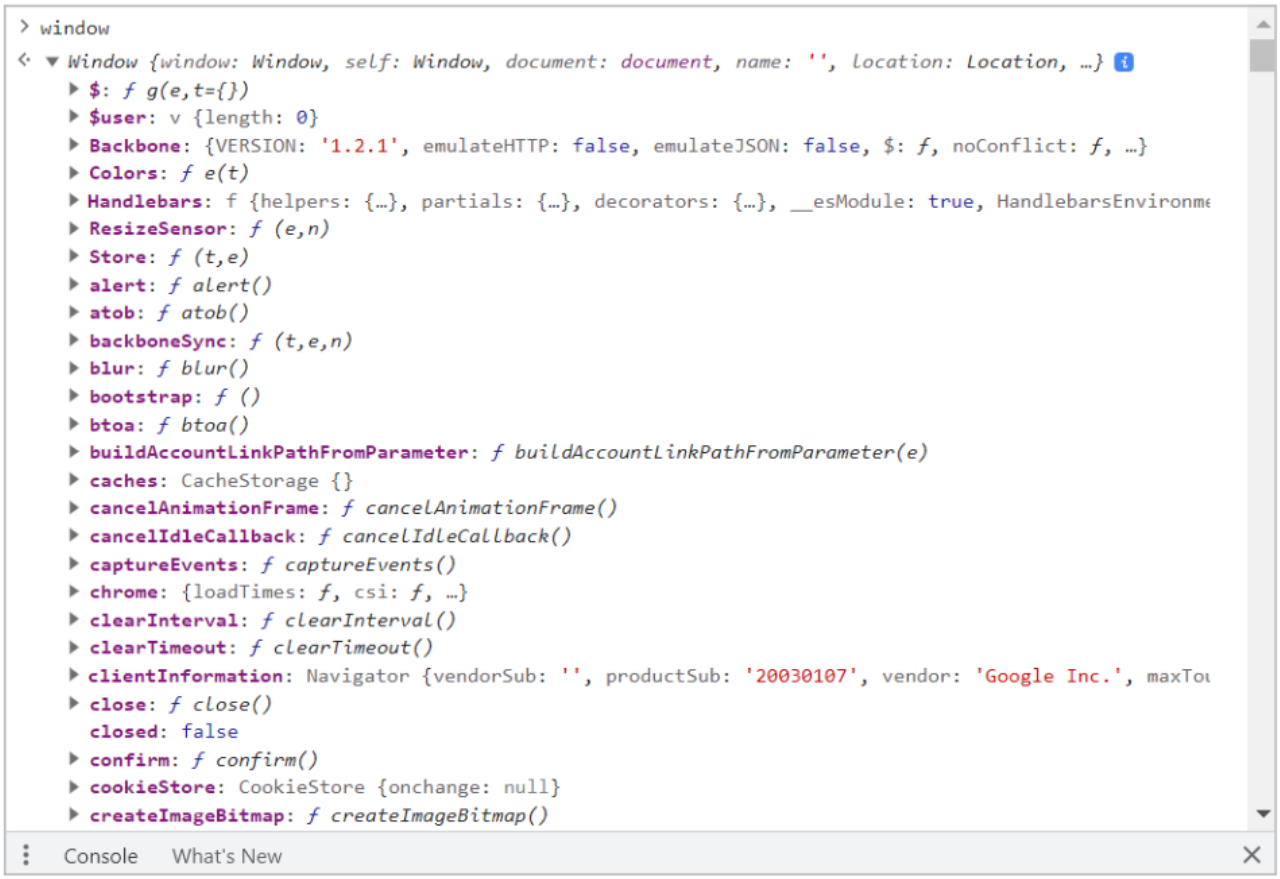
Introduction to the Window object

In JavaScript, the window keyword represents the global object in a web browser environment. It is the top-level object that contains various properties and methods related to the browser window or tab.

The window keyword can be used to view all the global objects present in a web browser by going to the developer mode of the browser and writing a window in the console.

i.e





Here are a few key points to understand about the window object:

- Global Scope: When you declare a variable or function in the global scope (outside of any function), it becomes a property or method of the window object. For example, if you define a variable x directly in the global scope, it can be accessed as window.x.
- Browser Environment Access: The window object provides access to various browser-related features and APIs. For example, you can use window.document to access the Document Object Model (DOM), which represents the web page's structure and content.
- Window Methods and Properties: The window object has several built-in methods and properties that allow interaction with the browser window. Some commonly used properties include window.location (provides information about the current URL), window.alert() (displays an alert dialog), window.prompt() (displays a prompt dialog), and window.console (provides access to the browser console).
- Global Object for Scripts: In the context of running JavaScript scripts within a web page, the window object acts as the global object. This means that variables and functions declared in script files are attached to the window object and can be accessed globally.

Window Properties (screen & location)

window.screen Property

The window.screen property provides information about the user's screen or monitor. It's an object that contains properties like:

- **screen.width:**

  Returns the width of the user's screen in pixels.
  This property is often used to determine the screen's horizontal resolution.

- **screen.height:**

  Returns the height of the user's screen in pixels.
  It's commonly used to find the vertical resolution of the screen.

Example for screen width & height

```
// Accessing screen width and height
const screenWidth = window.screen.width;
const screenHeight = window.screen.height;

console.log("Screen width: " + screenWidth);
console.log("Screen height: " + screenHeight);

// output
// Screen width: 1024
// Screen height: 885
```

Note: the values may vary depending on your screen size

- **screen.availWidth:**

  Returns the available width of the screen.
  It may exclude space used by taskbars, system bars, or dock on the screen.
  Useful for designing web applications to fit within the visible area of the screen.

- **screen.availHeight:**

  Returns the available height of the screen.
  Like screen.availWidth, it excludes any space occupied by taskbars or system bars.
  Useful for determining the vertical space available for content.

Example for screen availWidth & availHeight

```
// Accessing available screen width and height
const availScreenWidth = window.screen.availWidth;
const availScreenHeight = window.screen.availHeight;

console.log("Available screen width: " + availScreenWidth);
console.log("Available screen height: " + availScreenHeight);

//output
// Available screen width: 1024
// Available screen height: 885
```

Note: the values may vary depending on your screen size

- **screen.colorDepth:**

  Returns the color depth of the screen in bits per pixel.
  It represents the number of distinct colors a screen can display.
  Common values are 8 (256 colors), 16 (65,536 colors), 24 (16.7 million colors), or 32 bits (true color).

- **screen.pixelDepth:**

  Similar to screen.colorDepth, this property also represents the pixel depth, often the same as color depth.
  It's an alternative way to access the color depth of the screen.

Example for screen colorDepth & pixelDepth

```
// Accessing colorDepth and pixelDepth
const colorDepth = window.screen.colorDepth;
const pixelDepth = window.screen.pixelDepth;

console.log("Color depth: " + colorDepth + " bits per
pixel");
console.log("Pixel depth: " + pixelDepth + " bits per
pixel");

//Output
// Color depth: 24
// Pixel depth: 24
```

- **window.location Property**

The window.location property in JavaScript is a crucial part of the window object, and it provides information about the current URL of the web page. It can also be used to navigate to other pages and manipulate the browser's history.

- **Accessing window.location:**

You can access the window.location property to obtain information about the URL of the current page

```
const currentURL = window.location.href; // Retrieves the
complete URL
const hostname = window.location.hostname; // Retrieves the
hostname (e.g., www.example.com)
const path = window.location.pathname; // Retrieves the path
(e.g., /path/to/page)
const searchParams = window.location.search; // Retrieves the
query parameters (e.g., ?param1=value1&param2=value2)
```

- **Navigating with window.location methods:**

The window.location property provides methods to navigate to other pages. Let's look at them

- **window.location.assign(url):**

  Loads the specified URL in the current window. It's commonly used for navigation and behaves like clicking a link.

```
window.location.assign("https://www.example.com/newpage");
```

- **window.location.replace(url):**

  Similar to assign(), but it replaces the current URL in the browser's history, making it difficult for the user to navigate back to the previous page.

```
window.location.replace("https://www.example.com/newpage");
```

- **window.location.reload(forceReload):**

  Reloads the current page. If forceReload is set to true, it forces a reload from the server rather than using the cache.

```
window.location.reload(); // Standard page reload
window.location.reload(true); // Forced server refresh
```

- **Modifying window.location:**

  You can modify the window.location property to change the current URL. This is useful for creating dynamic navigation or updating the query parameters.

```
window.location.href = "https://www.example.com/newpage"; //
Change the entire URL
window.location.search = "?param1=newvalue&param2=newvalue";
// Update query parameters
```

- **Accessing Query Parameters:**

  To access and manipulate query parameters in the URL, you can use window.location.search. You can parse and update query parameters using JavaScript. We use the URLSearchParams constructor to parse the query string.

```
const urlParams = new URLSearchParams(window.location.search);
const paramValue = urlParams.get("param1");
```

- **Page Navigation and Browser History:**

  The window.location property is also involved in managing the browser's session history. It keeps track of pages visited by the user, allowing you to go back and forward in the history stack using methods like window.history.back() and window.history.forward().

- **Security Considerations:**

  When manipulating window.location, be aware of potential security issues, such as open redirect vulnerabilities. Always validate and sanitize user input to prevent unauthorized redirection to malicious websites.

- **cookies storage**

  Cookies are a fundamental part of web development and provide a way to store small pieces of information on a user's device. They are commonly used for various purposes, such as user authentication, session management, and tracking user preferences. In JavaScript, you can work with cookies to store and retrieve data on the client side.

- **What are Cookies?**

  Cookies are small text files that websites can store on a user's device.
  They are primarily used to persist data between web page visits, allowing websites to remember user information.

- **Creating a Cookie:**

  To create a cookie in JavaScript, you can use the document.cookie property. You assign a string value to it in the form of "key=value," followed by optional attributes.

- **For example:**

```
document.cookie = "username=John; expires=Thu, 31 Dec 2030
23:59:59 UTC; path=/";
```

The above code sets a cookie named "username" with the value "John," an expiration date, and a path.

- **Reading a Cookie:**

To read a cookie, you can access the document.cookie property and parse it.
JavaScript does not provide a direct way to read cookies by name, so you often need to implement a custom function or use a library for this.
For example, to read the "username" cookie:

```
function getCookie(name) {
  const cookies = document.cookie.split('; ');
  for (const cookie of cookies) {
    const [key, value] = cookie.split('=');
    if (key === name) return value;
  }
}
const username = getCookie("username");
```

- **Modifying a Cookie:**

To modify an existing cookie, you can reassign it with the same name, which effectively updates its value and attributes.
To delete a cookie, set its expiration date in the past.

- **Example:**

```
document.cookie = "username=Jane; expires=Thu, 31 Dec 2030
23:59:59 UTC; path=/";
```

- **Cookie Attributes:**

Cookies can have various attributes, such as "expires," "path," "domain," "secure," and "samesite," which control when and how the cookie is sent to the server.
The "expires" attribute sets the expiration date and time for the cookie.
The "path" attribute specifies the URL path for which the cookie is valid.
The "domain" attribute restricts the cookie to a specific domain.
The "secure" attribute ensures the cookie is only sent over secure (HTTPS) connections.
The "samesite" attribute controls when the cookie is sent based on the same-site or cross-site context.

- **Limitations:**

Cookies have limitations, including a small storage capacity (usually around 4KB per domain) and being sent with every HTTP request, which can impact performance.
Alternatives to Cookies:

For handling more extensive data storage, you can consider alternatives like Web Storage (localStorage and sessionStorage), IndexedDB, or Server-Side Sessions.

- **setTimeout**

setTimeout function is a crucial tool for scheduling the execution of code after a specified delay. This delay is measured in milliseconds, meaning you can set a timer to execute a function or a piece of code after a specific amount of time has passed and setTimeout function returns a timer identifier, which is a unique numerical value. It's a fundamental part of creating interactive and asynchronous web applications.

- **Syntax:**

```
setTimeout(function, delay);
```

- **function:**

This is the function or code snippet that you want to execute after the specified delay.

- **delay:**

The delay (in milliseconds) before the function execution.

- **Example:**

```
function sayHello() {
  console.log("Hello, world!");
}

// Schedule the 'sayHello' function to execute after 2000
milliseconds (2 seconds)
setTimeout(sayHello, 2000);

//output
Hello, world!
```

- **Example 1: Display a set of messages using setTimeout()**

```
setTimeout(() => {console.log("First message")}, 1000);
setTimeout(() => {console.log("Second message")}, 2000);
setTimeout(() => {console.log("Third message")}, 3000);
```

- **Output:**

```
First message
Second message
Third message
```

In the above code, the first time out function after 1 second delay will be executed first, the second timeout function will be executed after 2 seconds delay and the third function will execute after 3 second delay.

- **Example 2 :**

- **Output:**

```
First message without timeout
Second message without timeout
Third message without timeout
First message with timeout
Second message with timeout
Third message with time out
```

Here in the above example,

The line console.log("First message without timeout"); is executed first, and it immediately logs the message "First message without timeout" to the console.

The line setTimeout(() => {console.log("First message with timeout")}, 1000); schedules the execution of a callback function after a delay of 1000 milliseconds (1 second). This means that the callback function, which logs the message "First message with timeout", will be executed after a 1-second delay.

The line console.log("Second message without timeout"); is executed next and logs the message "Second message without timeout" to the console immediately.

The line setTimeout(() => {console.log("Second message with timeout")}, 2000); schedules the execution of a callback function after a delay of 2000 milliseconds (2 seconds). This callback function logs the message "Second message with timeout".

The line console.log("Third message without timeout"); is executed and logs the message "Third message without timeout" to the console immediately.

The line setTimeout(() => {console.log("Third message with time out")}, 3000); schedules the execution of a callback function after a delay of 3000 milliseconds (3 seconds). The callback function logs the message "Third message with timeout".

- **Example 3 : setTimeout() with 0 delay**

```
console.log("First message");

setTimeout(() => {
  console.log("Second message");
}, 0);

console.log("Third message");
```

In this code, we have three console.log statements and a setTimeout call with a delay of 0 milliseconds.

- **Output:**

```
First message
Third message
Second message
```

Here in the above example,

The first console.log("First message"); statement is executed, and it immediately logs the message "First message" to the console.
The setTimeout function is then called with a callback function and a delay of 0 milliseconds. Despite the 0-millisecond delay, the callback function is not executed immediately. Instead, it is queued to be executed by the event loop as soon as possible after the current task completes.
The execution continues to the next line, which logs the message "Third message" to the console.
After the current task is completed, the event loop picks up the queued callback function from the setTimeout call with a delay of 0 milliseconds. It then executes the callback, logging the message "Second message" to the console.

It's important to note that even though the delay is set to 0 milliseconds, the callback function is not executed immediately. Instead, it is placed in the event loop and scheduled to be executed in the next available task. This demonstrates the non-blocking nature of asynchronous operations in JavaScript and how they are managed by the event loop.

- **Example 4 : Tricky loop**

```
for (var i = 1; i ⩽ 5; i++) {
  setTimeout(() ⇒ {
    console.log(i);
  }, 3000);
}
```

- **Output:**

```
6
6
6
6
6
```

The intention of the code appears to be printing the numbers 1 to 5 with a delay of 3 seconds (3000 milliseconds) between each number. However, due to the behavior of closures and the "var" keyword having functional scope, the output will not be as expected.

The issue lies in the fact that the arrow function passed to setTimeout retains a reference to the variable i, and by the time the function is executed after the delay time which is 3 second here, the loop has already completed, leaving i with a value of 6. As a result, when the arrow function is invoked, it will print 6 for all iterations.

So how we can fix this issue?

1. We can use “let” instead of “var”

```
for (let i = 1; i ⩽ 5; i++) {
  setTimeout(() ⇒ {
    console.log(i);
  }, 3000);
}
```

- **OUTPUT:**

```
1
2
3
4
5
```

By using let instead of var, each iteration of the loop will have its own block-scoped variable i, ensuring that the correct value is captured and logged after the delay. With this modification, the expected output will be the numbers 1 to 5, each printed after a delay of 3 seconds.

2. We can pass the parameter to the function inside setTimeout instead of relying on the "var" keyword. This can be achieved by creating a separate scope using an Immediately Invoked Function Expression (IIFE)

```
for (var i = 1; i ⩽ 5; i++) {
  (function(num) {
    setTimeout(() ⇒ {
      console.log(num);
    }, 3000);
  })(i);
}
```

- **Output:**

```
1
2
3
4
5
```

In this updated code, an IIFE is used to create a new scope for each iteration of the loop. The current value of i is passed as an argument num to the IIFE, ensuring that each iteration has its own copy of num. Within the arrow function passed to setTimeout, the correct value of num is logged after the specified delay.

- **clearTimeout**

The clearTimeout() method cancels a timeout previously established by calling setTimeout().

- **Syntax**

```
clearTimeout(timeoutID)
```

The clearTimeout() method requires the id returned by setTimeout() to know which setTimeout() method to cancel.

- **Example: To demonstrate the functioning of clearTimeout()**

```
const timeoutId = setTimeout(function(){
    console.log("Hi");
}, 2000);

clearTimeout(timeoutId);
console.log(`Timeout ID ${timeoutId} has been cleared`);
```

- **Output:**

```
Timeout ID 2 has been cleared
```

- **setInterval**

This method repeats a given function at every given time interval and returns an interval ID that uniquely identifies the interval.

- **Syntax:**

```
setInterval(function, delay milliseconds);
```

function: the first parameter is the function to be executed
delay milliseconds: indicates the length of the time interval between each execution.

A function to be executed every delay milliseconds. The first execution happens after a delay of the specified milliseconds.

There is also an option of passing arguments to the callback function in setInterval, for that the syntax would be:

```
setInterval(function,delay milliseconds, arg1, arg2……argN)
```

As you can see, you may pass as many arguments as required.

Let us look at a few examples for a better understanding

- **Example 1: Displaying messages with the help of setInterval()**

```
const exSetInterval = setInterval(myRepeatedMessage, 300);
function myRepeatedMessage(){
 console.log('Hi');
 console.log('Hi Again !');
}

// OUTPUT
/*
Hi
Hi Again !
```

```
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !

... this continues
*/
```

Here, the function myRepeatedMessage() will keep on printing the messages 'Hi' and 'Hi Again' after every 300 milliseconds.

- **Example 2: Modify the above example by passing arguments to the function.**

```
const exSetInterval = setInterval(myRepeatedMessage, 300, 'Hi', 'Hi Again');
function myRepeatedMessage(a,b)
{
 console.log(a);
 console.log(b);
}

// OUTPUT
/*
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !

... this continues
*/
```

- **Example 3: Displaying time with the help of setInterval and another helping function.**

```
setInterval(timer, 1000);
function timer() {
  const date = new Date();
  const newTime = date.toLocaleTimeString();
  console.log(newTime);
}
// OUTPUT
/*
11:33:15 AM
11:33:16 AM
11:33:17 AM
11:33:18 AM
11:33:19 AM
11:33:20 AM
11:33:21 AM
11:33:22 AM
11:33:23 AM
11:33:24 AM
11:33:25 AM
11:33:26 AM
11:33:27 AM
11:33:28 AM
11:33:29 AM
11:33:30 AM
11:33:31 AM
11:33:32 AM
11:33:33 AM
11:33:34 AM
11:33:35 AM
... this continues
*/
```

- **clearInterval**

The clearInterval() method cancels a timed, repeating action which was previously established by a call to setInterval().
If the parameter does not identify a previously established action, this method does nothing.

- **Syntax :**

```
clearInterval(intervalID)
```

intervalID: The identifier of the repeated action you want to cancel. This ID was returned by the corresponding call to setInterval().

- **Example: How clearInterval works on setInterval()**

```
var interval = setInterval(warning, 3000);
function warning() {
  console.log('3 second warning');
  clearInterval(interval);
}

// OUTPUT:
3 second warning
```

- **alert**

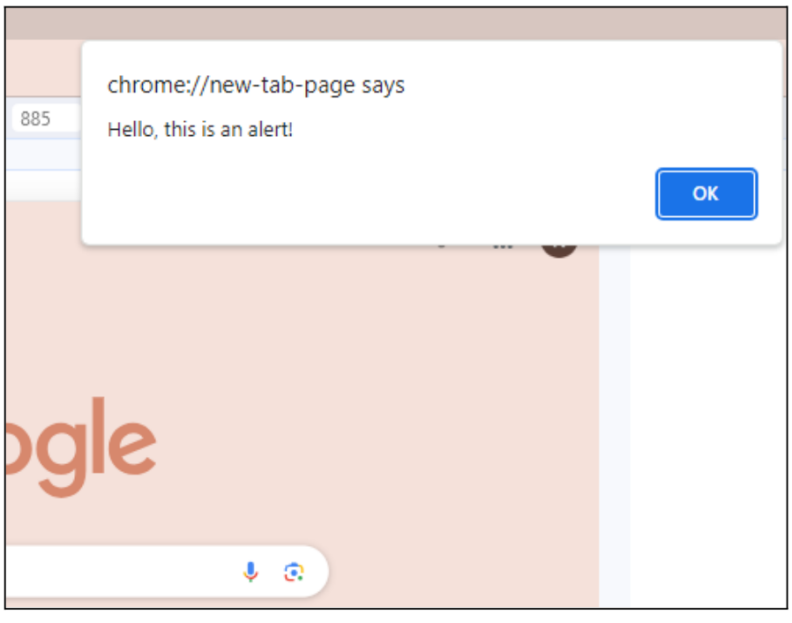
The alert function is a built-in feature provided by web browsers for displaying a simple dialog box with a message to the user. It's a basic way to communicate with the user and obtain their attention in a web application.

- **Syntax:**

```
alert(message);
```

- **Example:**

```
alert("Hello, this is an alert!");
```



alert dialogs are synchronous and block the execution of JavaScript code until the user dismisses the dialog. Its disruptive nature and limited customization make it a less suitable choice for more complex user interactions, where alternative methods like modals or notifications are preferred.

- **prompt**

The prompt function is a built-in feature provided by web browsers to display a dialog box for collecting user input. It's commonly used for basic data entry in web applications.

- **Syntax :**

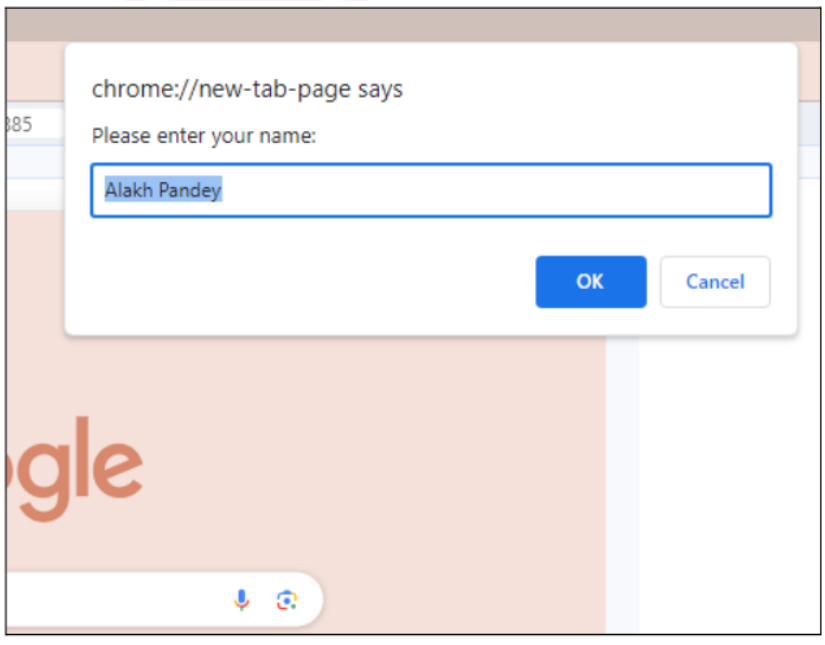
```
prompt(message, defaultInput);
```

message: This is the message or question you want to display to the user in the dialog. It should be a string.

defaultInput (optional): This is the default input text that is displayed in the input field. It should also be a string. This parameter is optional.

- **Example:**

```
const userName = prompt("Please enter your name:", "Alakh Pandey");
```



In this example, a dialog box with the message "Please enter your name:" is displayed to the user. It also includes an input field with the default value "Alakh Pandey" The user can type their name in the input field.

While prompt is useful for basic data collection, it's important to be mindful of user experience. Dialog boxes can be intrusive, and there are more modern ways to collect and validate user input, such as using HTML forms and input elements.

- **confirm**

The confirm function is a built-in JavaScript function provided by web browsers to display a confirmation dialog to the user. It's used to obtain a binary (yes/no) choice from the user and can be helpful for user interactions requiring a simple "OK" or "Cancel" response.

- **Syntax :**

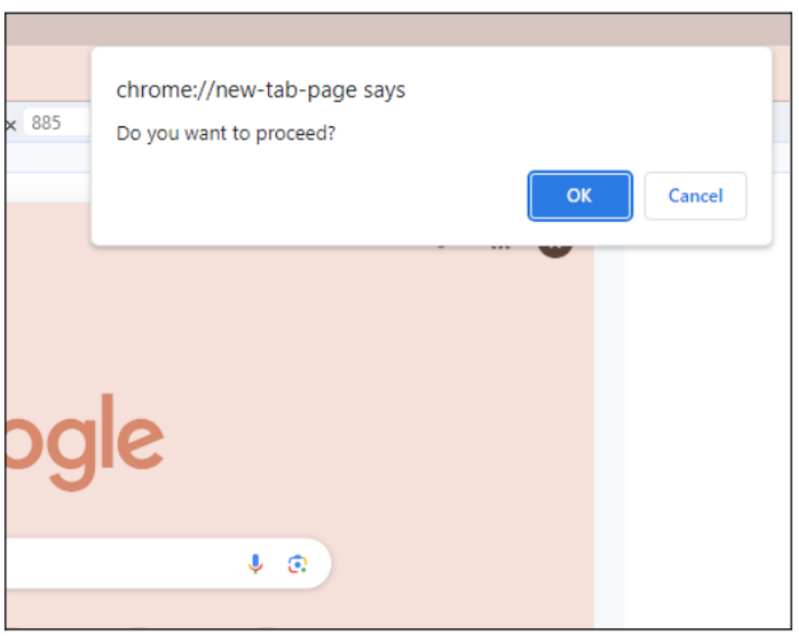
```
confirm(message);
```

message: This is the message or question you want to display to the user in the confirmation dialog. It should be a string.

- **Example:**

```
const userResponse = confirm("Do you want to proceed?");

console.log(userResponse)
```



In this example,
If user click on ok, the console will log true and if user clik on cancel, the console will log false.