What is BOM in JavaScript?

BOM - Browser Object Model

Browser gives us a Window Object by default.

This window object has some properties which allows us to interact with windows.

Let us see them one by one:

Use these in developer console in Browser

1. location - This has some methods like host, hostname, path, href, and reload()

```
1.1. location.reload() - While doing reload(), our site gets reloaded
```

`location.reload()`

1.2 location.href() - While doing this, we can alter our website's address

eg. `location.href = 'index.html'`

This will shift our website to index.html file (note to do this, make sure that the file that we are shifting our website to, should already occur in our codebase, here since every website has a default file of index.html, we are able to do it successfully).

Using this attribute, we can also go from one website to another.

eg. `location.href = 'website url'`

eg. `location.href = 'https://chatgpt.com/c/66e1d2f7-4ff8-8005-82d8-ab1ba8974a0c'`

---

2. history - This has some methods like:

a. backward() and forward() - This allows us to go back and forth when we open multiple web pages at once (like -> and <- arrow in our browser).

In console, do `history.back()` or `history.forward()`.

When we cannot go back or forward anymore, we get `undefined` as the return value.

b. go() - This allows us to go back and forth multiple pages.

eg. `history.go(7) -> Go forward 7 pages`

`history.go(-4) -> Go back 4 pages`

These are found in prototypes in history.

---

3. innerHeight and innerWidth - This gives us height and width of the display browser screen.

It may change as screen size changes.

4. outerHeight & outerWidth - This will give us the widths of display browser + console + all the widths of task bar, menu bar, tab ball all.

To find all the extra height and width, do `outerHeight - innerHeight` and `outerWidth - innerWidth`

`Outer Height is fixed while inner height is variable`

---

5. open() - Opens a new tab

Returns the window object of a new tab.

`Different tabs have different window objects`

`open()`

This can be used to navigate to different file pages or even go to a new website in a new tab.

`open('https://www.youtube.com/')`

Syntax -> `open(url, name)`

we can check this name in the opened site's name variable.

By default, `name = ''`

---

6. close() - Closes that window which was opened via open()

eg

```
open("https://www.youtube.com/");

// Window {window: Window, self: Window, document: document, name: '', location:
Location, …}

// Youtube is open now in a new tab

close("https://www.youtube.com/");
// Youtube is close now and the tab is also closed now.
```

7. resizeBy() and resizeTo()

Pre-requisite:

We need to have a resizable tab opened by doing the following:

```
open("https://www.youtube.com/", "ksd", "resizable");

// Window {window: Window, self: Window, document: document, name: 'ksd',
location: Location, …}
```

Then we can do either `resizeBy(400, 800)` or `resizeTo(400, 800)`

Syntax => `resizeTo(width, height)`

Key Differences:

- resizeBy() changes the size relative to the current window size.
- resizeTo() sets the window size to specific, absolute dimensions.

8. moveBy() and moveTo()

Syntax => moveBy(x length, y length);

`moveTo(0,0) => Top left corner of screen`

Key Differences

- moveBy() shifts the window by a relative number of pixels from its current position.
- moveTo() moves the window to specific, absolute screen coordinates.

9. scrollBy() and scrollTo()

In JavaScript, scrollBy(), scrollTo(), and scroll() are used to programmatically scroll elements (like the window or any scrollable element) in a web page. They differ slightly in their usage and scope.

1. scrollBy() Purpose: Scrolls an element (or the window) by a specific number of pixels relative to its current scroll position.

Syntax:

```
element.scrollBy(xDelta, yDelta);
```

- xDelta: The number of pixels to scroll horizontally (positive values scroll to the right, negative values scroll to the left).
- yDelta: The number of pixels to scroll vertically (positive values scroll down, negative values scroll up).

Example:

```
window.scrollBy(100, 200); // Scrolls the window 100px to the right and 200px
down.
```

This method moves the scroll position relative to the current position of the element or the window.

2. scrollTo() Purpose: Scrolls an element (or the window) to a specific set of coordinates. It sets the scroll position to the exact values provided.

Syntax:

```
element.scrollTo(x, y);
```

- x: The horizontal position to scroll to (in pixels).
- y: The vertical position to scroll to (in pixels).

Example:

```
window.scrollTo(0, 500); // Scrolls to the vertical position 500px down and keeps the horizontal position unchanged.
```

In this case, the window or element will be scrolled to the absolute coordinates provided.

3. scroll() Purpose: This method works similarly to scrollTo(), and in modern browsers, it's essentially the same. It scrolls an element (or window) to the specified coordinates. scroll() also allows for additional options like smooth scrolling.

Syntax:

```
element.scroll(x, y); // Basic syntax like scrollTo()

// Or with additional options
element.scroll({
  top: y,
  left: x,
  behavior: "smooth", // Optional for smooth scrolling
});
```

Example:

```
window.scroll(0, 300); // Equivalent to window.scrollTo(0, 300)

window.scroll({
  top: 300,
  left: 0,
  behavior: "smooth", // Scrolls smoothly to 300px down
});
```

scroll() offers a flexible option with smooth scrolling, which scrollTo() doesn't natively provide unless invoked with additional arguments in modern browsers.

Key Differences:

- scrollBy(): Scrolls relative to the current position (adds or subtracts from the current scroll position).
- scrollTo(): Scrolls to a specific, absolute position.
- scroll(): Essentially an alias for scrollTo(), but supports smooth scrolling and additional options.

Smooth Scrolling:

The scroll() method can include a behavior option (e.g., 'smooth') to animate the scrolling, while scrollTo() is limited unless combined with options in modern implementations.

10. Print()

Used to print a web page via a printer.

eg. `print()` and the printer window pops up.

11. Document (IMP) - This is DOM (Document Object Model). DOM is a small part of BOM but it is very important.

DOM is a part of `Web APIs`. These APIs help us in running code from the code from the JS engine to the Web Browser. Also these help us in interacting with the Web Browser via JavaScript.

What is BOM?

Different Browser vendors like Chrome, FireFox, Safari has given us an Object Model called `BOM (Browser Object Model)` to interact with the Browser, we can move it, resize it, scroll it, see its history, locate it etc.. So BOM helps us do all of this and all these functionalities is present in a main object called `WINDOW OBJECT`.

Number System and Number Object

Number Systems in JS

- Binary
- Octal
- Decimal
- Hexadecimal

Decimal System: 123, 1.58, 1 etc..

Octal System: 0124, 0o124, 01, 0o1 etc..

Use the `0o123` instead of `0123` in VS Code console.

Octal

Uses numbers from 0 to 7 (8 characters)

```
const numOct = 0o15;
console.log(numOct + 15);
// numOct = 13, hence 13 + 15(decimal) = 28
```

Here in `0o15` -> `0o` represents the Number System (octal here) and `15` represents the number `15` in octal. Upon conversion, it will be changed to decimal (13)

- `We cannot write 8 and 9 in octal (only 0 to 7)`

Binary

`We can only put 0s and 1s here` (2 characters)

```
const numB = 0b1100;
console.log(numB);

// 12 in decimal
```

`0b` Number System `1100` = binary number -> 12 in decimal

HexaDecimal

Uses 16 characters (0 to 9) and (A to F) (10 digits and 6 alphabets => 16 characters).

eg

```
const numH = 0x00aa7c9d;
console.log(numH);

// 11173021
```

Here also `0x` => Number System & `00AA7C9D` -> Number (Hexa Decimal)

Upon conversion, we get 11173021 in decimal.

Also here we cannot use any other character apart from `0 to F` else we get error.

Scientific (Exponential) Notation

```
const numSPos = 1.4578e7;
console.log(numSPos);
// 1.4578 * (10 ^ 7) = 14578000

const numSNeg = 1.4578e-3;
console.log(numSNeg);
// 1.4578 * (10 ^ -3) = 0.0014578
```

Some Conventional Practices

- The number before e should be from 0.0 to 9.9 i.e. always less than 10, if it is greater than 10, raise the value of the number after e by 1.

```
1234e4 // this is valid but not the right practice
```

Let us now look at some properties in Number

1. Number.MIN_SAFE_INTEGER & Number.MAX_SAFE_INTEGER

These give us the maximum and minimum possible value of numbers that are safe to work with in JS.

eg.

```
console.log(Number.MAX_SAFE_INTEGER);
// 9007199254740991 (2^53 - 1)

console.log(Number.MIN_SAFE_INTEGER);
// -9007199254740991 (2 ^ -53)
```

Any number greater than this will make our lives difficult while performing operations,

eg. `9007199254740991 + 4 = 9007199254740996 and not 9007199254740995`

So performing simple operations here becomes difficult

Solution -> Use BigInt for numbers greater than this

`9007199254740991n + 4n = 9007199254740995n`

Big int + big int ☑
Big int + int ❌

Hence we did `+ 4n` and not `+ 4`

```
const num = Number.MAX_SAFE_INTEGER;
const ans = BigInt(num) + 4n;
```

```
console.log(ans);

// 9007199254740995n
```

2. Number.MAX_VALUE & Number.MIN_VALUE

These give us the Maximum and Minimum value of number we can store in JS under the Number datatype (big int is separate from this).

```
console.log(Number.MAX_VALUE);
// 1.7976931348623157e+308

console.log(Number.MIN_VALUE);
// 5e-324
```

If we try to add or subtract values from these, there will be no change seen in output as this number is too large to show value changes in the console.

3. Number.EPSILON

```
console.log(Number.EPSILON);
// 2.220446049250313e-16
```

This is a number which is very small and is very close to 0.

It is the difference between 1 and the nearest decimal value greater than 1. Any value smaller than this nearest decimal value will be treated as equal to 1.

Where do we use this:

- IN JS when we tend to add small decimal numbers, they are not always giving us the accurate result, so we can use epsilon to add a conditional check and the print the number.

see this eg.

```
const result = Math.abs(0.2 - 0.3 + 0.1);
// result can be a negative number, so we try to make it positive via Math.abs()

console.log(result);
// Expected output: 2.7755575615628914e-17

// We now check if this value is equal to 0 or not by comparing if it is smaller
than Epsilon.

// If it is true, then the result is 0
// If it is false, then the result is not 0
```

```
console.log(result < Number.EPSILON);
// Expected output: true

// Output:
// 2.7755575615628914e-17 // result
// true // yes this is equal to 0
```

some more egs

```
1 + Number.EPSILON;
// 1.0000000000000002

// 1 + 1.0000000000000002
2;

1.0000000000000002;
// 1.0000000000000002

1.0000000000000001;
// 1

// This last line means that any number smaller than 1 + Number.Epsilon will be
treated as one and not a unique number, this application is shown in the above
example where we compare if the result is equal to 0 or not
```

4. Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY

These give us values `Infinity` and `-Infinity`

5. Number.NaN

This gives us the value `NaN`

Always remember `NaN === NaN` is always false

Some Number methods

1. Number.isNaN()

Gives a true or false value and checks if a Number is NaN or not

```
console.log(Number.isNaN(145));
console.log(Number.isNaN(NaN));
console.log(Number.isNaN("l"));
console.log(Number.isNaN(""));

// false
// true
```

```
// false
// false
```

There is a `isNaN()` method also which first converts the value into a number and then checks if it is equal to NaN or not, it will give true if it is not able to convert the value into a number and false if it successfully converts the value into a number.

But `Number.isNaN()` checks directly if the value is === NaN or not without converting it into a number.

2. Number.isFinite()

Tells us true or false if a number is finite or not

```
console.log(Number.isFinite("5jklmf5"));
console.log(Number.isFinite("50"));
console.log(Number.isFinite(50));
console.log(Number.isFinite(5000000000000e17));
console.log(Number.isFinite(Infinity));

// false
// false
// true
// true
// false
```

It will give true till `Number.MAX_VALUE` after that it will give false.

Same case for -ve numbers.

3. Number.isInteger()

Tells us true or false if a number is an integer or not

```
console.log(Number.isInteger(-5.4));
console.log(Number.isInteger("-5.40"));
console.log(Number.isInteger(Infinity));
console.log(Number.isInteger(-80));
console.log(Number.isInteger(0));
console.log(Number.isInteger(05));
console.log(Number.isInteger(5.0));

// false
// false
// false
// true
// true
```

```
// true
// true
```

4. Number.isSafeInteger()

Checks if a number is a `safe integer or not`.

It ranges from MIN_SAFE_INTEGER to MAX_SAFE_INTEGER (i.e. returns `true` for all these values). For other values, including non integers and values greater or lesser than MIN_SAFE_INTEGER & MAX_SAFE_INTEGER, it gives `false`

```
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER));
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1));
console.log(Number.isSafeInteger(Number.MIN_SAFE_INTEGER));
console.log(Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1));
console.log(Number.isSafeInteger(3.59856));
// checks for integer, not float

// true
// false
// true
// false
// false
```

5. Number.parseInt()

The Number.parseInt() static method parses a string argument and returns an integer of the specified radix or base.

eg

```
function roughScale(x, base) {
  const parsed = Number.parseInt(x, base);
  if (Number.isNaN(parsed)) {
    return 0;
  }
  return parsed * 100;
}

console.log(roughScale(" 0xF", 16));
// Expected output: 1500 as F = 15 and 15 * 100 = 1500

console.log(roughScale("321", 2));
// Expected output: 0 as 321 is not properly converted to binary, it returns NaN
hence we get 0

// Outputs:
```

```
// > 1500
// > 0
```

Explanation for case 2

`console.log(roughScale('321', 2));`

- The input '321' is interpreted in base 2 (binary).
- Since 321 is not a valid binary number (binary digits can only be 0 or 1), the parsing fails, resulting in NaN.
- The Number.isNaN(parsed) condition is true, so the function returns 0.

Radix or Base

An integer between 2 and 36 that represents the radix (the base in mathematical numeral systems) of the string.

If radix is undefined or 0, it is assumed to be 10 except when the number begins with the code unit pairs 0x or 0X, in which case a radix of 16 is assumed.

6. Number.parseFloat()

The Number.parseFloat() static method parses an argument and returns a floating point number. If a number cannot be parsed from the argument, it returns NaN.

```
function circumference(r) {
  if (Number.isNaN(Number.parseFloat(r))) {
    return 0;
  }
  return parseFloat(r) * 2.0 * Math.PI;
}

console.log(circumference("4.567abcdefgh"));
// Expected output: 28.695307297889173

console.log(circumference("abcdefgh"));
// Expected output: 0

// Outputs:
// > 28.695307297889173
// > 0
```

Introduction to DOM | Document Object Model

Now this `document` inside Window Object is an object, so to see all its properties clearly, do `console.dir(document)`

This document has many properties: One of them is children. This children has HTML inside it.

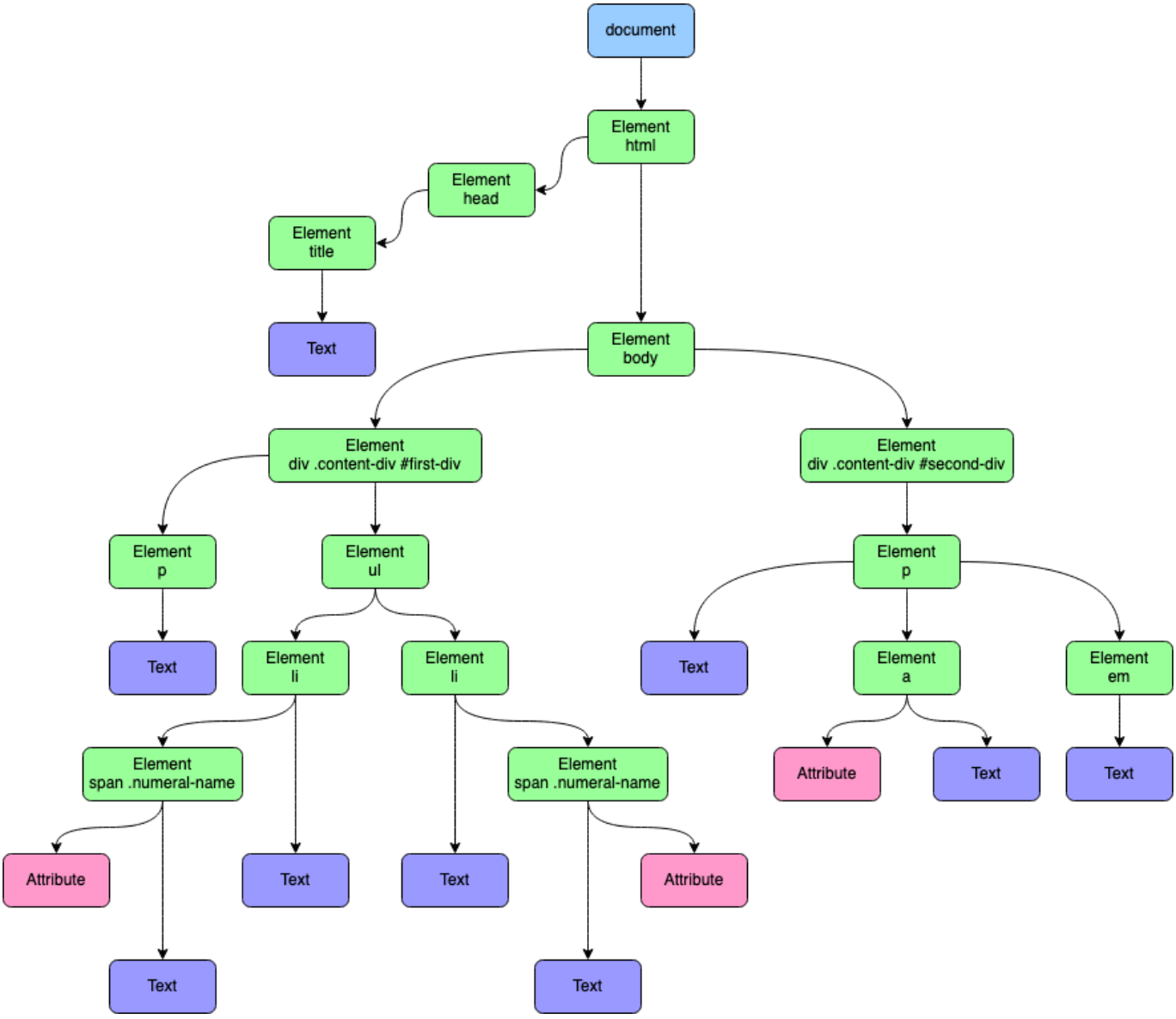
Inside this html children, we get another children property, this children has head and body as properties.

Inside head children, we have, meta, title, etc...

So this parent and its children keeps going on like a tree, this is called `DOM TREE`.

`Every HTML element (tags + content) is an object behind the scenes.`

So there is a thing in HTML called `HTML Parser`, this HTML parser reads all the HTML lines one by one and then converts each element into an object and then establishes a parent-child relation between different elements and thus makes a DOM Tree.



Also every attribute that we give to an element (which is an object), becomes that object's property where attribute name becomes the key and the attribute value becomes the value of that key.

```
document.children;

// HTMLCollection [html]
```

This HTML Collection is an object, not an array as we can not use array methods with it, but to access its values i.e. children inside it, we will need to use (.) or ([]) notation.

```
document.children[0];

/*
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Front-End-Roadmap</title>
  </head>
  <body style="font-family: sans-serif">
    <h1>Frontend Development for KSD</h1>
    <hr />
    <p>
      <strong> Frontend development </strong> is the development of the
      <a href="https://en.wikipedia.org/wiki/Graphical_user_interface">
        graphical user interface
      </a>
      of a website, through the use of
      <a href="https://en.wikipedia.org/wiki/HTML" target="_blank"> HTML,</a>

      <a href="https://en.wikipedia.org/wiki/CSS" target="_blank">CSS,</a>
      and
      <a href="https://en.wikipedia.org/wiki/JavaScript">JavaScript,</a>
      so that users can view and interact with that website.
    </p>
    <img
      src="./assets/assets/images/html-css-javascript.png"
      alt="html-css-javascript image"
      style="width: 100%; max-width: 600px"
    />

    <ul>
      <li>
        <b><big>HTML</big>: </b>
        HTML: The <big>HyperText Markup Language</big> or <em>HTML</em> is the
        <small>standard</small>
        markup language for documents designed to be displayed in a web browser.
      </li>
      <br />
      <br />
      <li>
        <b>CSS: </b>
        CSS: Cascading
        <pre>            Style Sheets       </pre>
        <i>(CSS)</i> is a style sheet language used for describing the
        presentation of a document written in a markup language such as HTML or
        XML
      </li>
```

```
<br />
<img
  src="./assets/assets/images/css.png"
  alt="css-image"
  style="width: 100%; max-width: 600px"
  title="CSS Image"
/>

<!-- <ol type="1" start="6"> -->
<ol>
  <li>Pure CSS</li>
  <br />

  <li>Bootstrap <i>(Framework)</i></li>
  <p></p>
  <!-- 🔘 Also used for spaces -->
  <li>Tailwind <i>(Framework)</i></li>
  <br />

  <li>SASS <i>(Preprocessor)</i></li>
  <br />
</ol>

<br />
<br />

<li>
  <b>JavaScript: </b>
  JavaScript often abbreviated JS, is a programming language that is one
  of the core technologies of the World Wide Web, alongside HTML and CSS.
  It is use to add functionality in the website.
</li>
<br />
<img
  src="./assets/assets/images/javascript.png"
  alt="javascript-image"
  style="width: 100%; max-width: 600px"
/>
<br />
<br />
<ol type="a">
  <li>Vanilla JS</li>
  <br />

  <li>jQuery <i>(Library)</i></li>
  <br />

  <li>React <i>(Framework)</i></li>
  <br />

  <li>Angular <em>(Framework)</em></li>
  <br />
```

```
        <li>Vue <i>(Framework)</i></li>
        <br />

        <li>Typescript <i>(Preprocessor)</i></li>
        <br />
      </ol>
    </ul>
    <p style="text-align: center">All copyrights &copy; reserved</p>
  </body>
</html>

<!-- Html Entities: &copy; (copyright) -->

<!-- &times; => X -->

*/
```

But this is a long method, so we can directly access the body via `document.body`

```
document.body;
// HTMLCollection(7) [h1, hr, p, img, ul, p, script]

document.body.children[4];
// <ul>...</ul> (full code inside <ul></ul>)
```

Now if we do:

```
console.dir(document.body.children[4]);
```

We will get ul in an object form, with many properties, some of them are:

innerHTML and innerText

let us see some examples:

```
document.body.children;
// HTMLCollection(7) [h1, hr, p, img, ul, p, script]

document.body.children[0].innerHTML;
// 'Frontend Development for KSD'

document.body.children[0].innerHTML = "I am a new heading";
// 'I am a new heading'
```

So we can temporarily change this innerHTML in DOM and we will be able to see these changes too in the screen, but when we reload the page, the changes go away as this is only temporary and the old values come

back.

innerHTML vs innerText vs textContent

The properties innerHTML, innerText, and textContent are used in JavaScript to interact with the content of HTML elements, but they have important differences. Here's an explanation with examples for each:

1. innerHTML

Purpose: Used to get or set the HTML content inside an element.

Behavior: It returns the element's HTML, including tags. If you set it, any HTML code inside the string will be parsed and rendered as part of the document.

Use Case: When you need to include HTML tags in your manipulation.

Example:

```
<div id="myDiv">
  <p>This is a <strong>paragraph</strong>.</p>
</div>

<script>
  const div = document.getElementById("myDiv");
  console.log(div.innerHTML); // Output: <p>This is a <strong>paragraph</strong>.
</p>

  // Changing innerHTML
  div.innerHTML = "<h2>New Heading</h2>";
  // The content becomes: <div id="myDiv"><h2>New Heading</h2></div>
</script>
```

Advantages: Can be used to dynamically insert HTML.

Disadvantages: It's not safe when dealing with user input, as it can introduce XSS vulnerabilities if the input is not properly sanitized.

2. innerText

Purpose: Used to get or set the visible text content of an element, as seen by the user.

Behavior: It strips out any HTML tags and returns only the text visible on the page. It respects styles such as display: none and visibility: hidden, so hidden elements are ignored.

Use Case: When you want the user-visible text and ignore the underlying HTML structure.

Example:

```
<div id="myDiv">
  <p>This is a <strong>paragraph</strong>.</p>
```

```
</div>

<script>
  const div = document.getElementById("myDiv");
  console.log(div.innerText); // Output: This is a paragraph.

  // Changing innerText
  div.innerText = "New text content";
  // The content becomes: <div id="myDiv">New text content</div>
</script>
```

Advantages: Only deals with visible text, ignoring HTML tags and hidden content.

Disadvantages: Modifies only the text and can't be used to insert HTML.

3. textContent

Purpose: Used to get or set the text content of an element, including all child elements, but without parsing or recognizing HTML tags.

Behavior: It returns all the text within the element, including text inside hidden elements, but it treats HTML tags as plain text.

Use Case: When you want all text (including from hidden elements) but without any concern for HTML.

Example:

```
<div id="myDiv">
  <p>This is a <strong>paragraph</strong>.</p>
</div>

<script>
  const div = document.getElementById("myDiv");
  console.log(div.textContent); // Output: This is a paragraph.

  // Changing textContent
  div.textContent = "Plain text content";
  // The content becomes: <div id="myDiv">Plain text content</div>
</script>
```

Advantages: Useful when you want to handle plain text and don’t care about formatting or hidden elements.

Disadvantages: Doesn’t recognize HTML, treats everything as text.

Summary of Differences:

Property	Returns HTML	Includes Hidden Elements	Recognizes HTML Tags

Property	Returns HTML	Includes Hidden Elements	Recognizes HTML Tags
innerHTML	Yes	Yes	Yes
innerText	No	No	No
textContent	No	Yes	No

Each property serves different purposes depending on whether you need to manipulate HTML structure, get user-visible text, or retrieve the full plain-text content.

So `textContent` returns hidden text also but `innerText` does not return hidden text. Text can be hidden via display: none or visibility: hidden etc.

Some more tasks

1. Changing the content of h2 from Heading 2 to Hello World

```
document.body.children;
// HTMLCollection(8) [h1, h2, hr, p, img, ul, p, script]

document.body.children[1].innerText = "Hello World";
// 'Hello World'
```

2. Changing the image src

```
document.body.children[4].src =
  "https://cdn.pixabay.com/photo/2015/03/17/02/01/cubes-677092_1280.png";
```

Similarly we can select the paragraph element

```
document.body.children[6];
```

but doing this again and again is too long, so we store it in a variable.

How is this possible? This is because all elements are objects, and all objects can be stored in a variable.

```
const myPara = document.body.children[6];
```

now we can make modifications by calling myPara

eg.

```
myPara.innerText = "Hello";
```

Selecting Elements in JavaScript

There are the following ways to select elements in DOM:

1. Type it from starting, eg.

`document.body.children[3].children[1]` like this but it is too long method

2. Store the code in step 1 in a variable and then move forward.

eg

```
const myVar = document.body.children[3].children[1];

myVar.innerText = "gbnvdmjcxki";
```

3.`document.getElementsByTagName('tagName')`

This will return an HTML Collection of all tags that we want to get, then we can use the array indexing method to search for that specific tag that we want to search.

eg

`document.getElementsByTagName('tagName')[1]`

This will give us access to the p tag at index 1 of the HTML Collection.

Special Case -> For images, we can directly do `document.images[2]` This will give us the access to image at index 2 of the HTML Collection.

4.`document.getElementsByClassName('class name')`

We need to have a className for this i.e. say we are selecting the 2nd image, then we have to give that img tag a class Name.

```
<img
  src="./assets/assets/images/html-css-javascript.png"
  alt="html-css-javascript image"
  style="width: 100%; max-width: 600px"
  class="myImg"
/>
```

```
document.getElementsByClassName("myImg");
// HTMLCollection [img.myImg]
```

With this, we can select multiple images having the same class name.

When there are multiple elements with same class name, just use array indexing to navigate to your desired element.

```
const images = document.getElementsByClassName("myImg");

images[2];
// gets the image at index 2 of the HTML Collection
```

5.document.getElementById('id name')

We know IDs are unique so by using this method, we can select a unique element having a unique id even if its class is not unique and is shared with other elements

eg

```
<img
  src="./assets/assets/images/javascript.png"
  alt="javascript-image"
  style="width: 100%; max-width: 600px"
  class="myImg"
  id="img3"
/>
```

```
document.getElementById("img3");
// <img src="./assets/assets/images/javascript.png" alt="javascript-image" style=
"width: 100%; max-width: 600px" class="myImg" id="img3">
```

See: id is unique but class is shared

6.document.querySelector()

This is the most widely used as we can give ids, classes and tags inside (). Just use it like a CSS selector

egs

```
document.querySelector(".myImg");
// As a class selector

document.querySelector("#img3");
```

```
// As an id selector

document.querySelector("img");
// As a tag selector

document.querySelector('[alt="javascript-image"]');
// As an attribute selector
```

If there are multiple elements which are same, querySelector will select the first element out of many.

7.`document.querySelectorAll()`

For selecting multiple elements.

eg.

```
document.querySelectorAll(".myImg");
// NodeList(3) [img.myImg, img.myImg, img#img3.myImg]
```

This gives us a `NODE LIST` of all elements which matches the criteria of the condition given inside the ()

`Node List is better than HTML Collection as we can use array methods like forEach inside of it.`

```
const images = document.querySelectorAll(".myImg");
// undefined

images.forEach((img) => {
  console.log(img);
});

// We get all the img tag code in each iteration of this method
```

Even if we use this method and give same id to multiple elements, it will not throw any error. But it is not a good practice.

Some Practical Applications

```
const allImages = document.querySelectorAll(".myImg");

const myImages = [
  "https://pbs.twimg.com/profile_images/1249432648684109824/J0k1DN1T_400x400.jpg",
  "https://thumbs.dreamstime.com/b/man-feeling-suspicious-face-expression-emotion-
hesitating-facial-studio-shot-white-isolated-background-copy-space-90927117.jpg",
  "https://encrypted-tbn0.gstatic.com/images?
q=tbn:ANd9GcTWCVnMftmnJZ8Nis5HnBeuahDMftylU9guag&s",
];
```

```
// for each
allImages.forEach((img, index2) => {
  img.src = myImages[index2];
  // using 2 indices
});
```

`document.querySelector()` -> Search in whole document

```
const ulVar = document.querySelector("ul");
ulVar.querySelector();
```

Search in just this ulVar i.e. inside the ul

Shortcut to copy total path of an element in js:

In console -> go to html code > go to the element > right click > copy > copy js path

---

Inner Text vs Text Content in JS

When we write code and give content in our HTML file, we do some things like give spaces, add new lines etc...

So when we display it in a browser, it ignores some things like extra spaces so, when we do innerText, we see whatever we have seen in the browser but with textContent, we get whatever we have entered in the Html file, like new lines, spaces etc..

Also some things like invisible texts are also not visible in innerText, but are visible in text content.

eg

```
const para = document.querySelector("p");
// undefined

para.innerHTML;
// '\n      <strong> Frontend development </strong> is the development of the\n
<a href="https://en.wikipedia.org/wiki/Graphical_user_interface">\n
graphical user interface\n      </a>\n      of a website, through the use of\n
<a href="https://en.wikipedia.org/wiki/HTML" target="_blank"> HTML,</a>\n\n
<a href="https://en.wikipedia.org/wiki/CSS" target="_blank">CSS,</a>\n      and\n
<a href="https://en.wikipedia.org/wiki/JavaScript">JavaScript,</a>\n      so that
users can view and interact with that website.\n    '

para.innerText;
// 'Frontend development is the development of the graphical user interface of a
```

```
website, through the use of HTML, CSS, and JavaScript, so that users can view and
interact with that website.'

para.textContent;
// '\n       Frontend development  is the development of the\n      \n
graphical user interface\n      \n      of a website, through the use of\n
HTML,\n\n      CSS,\n      and\n      JavaScript,\n      so that users can view
and interact with that website.\n
```

```
visibility : hidden
display: none
```

When used these with text, we cannot see anything while using innerText, but we can see things via textContent.

---

getAttribute and setAttribute in JavaScript

We can either give an element a valid attribute with a value or a custom attribute with a value.

```
<h2 kaustubhya="Shukla">Heading 2</h2>
```

```
document.querySelector('[kaustubhya="Shukla"]').attributes[0];
// kaustubhya="Shukla"

document.querySelector('[kaustubhya="Shukla"]').attributes[0].value;
// 'Shukla'
```

How to get that attribute in JS DOM

```
document.querySelector("[kaustubhya]").getAttribute("kaustubhya");

// 'Shukla'
```

If the attribute mentioned inside () is not there, then we get `null`.

We can also get class attribute values in a string or id attribute values in a string, if we follow this method.

```
<a
  class="src src2"
  id="mySrc"
```

```
  href="https://en.wikipedia.org/wiki/CSS"
  target="_blank"
  >CSS,</a
>
```

```
document.querySelector(".src").getAttribute("class");
// 'src src2'

document.querySelector(".src").getAttribute("id");
// 'mySrc'
```

Set Attribute

Before:

```
<a
  class="src src2"
  id="mySrc"
  href="https://en.wikipedia.org/wiki/CSS"
  target="_blank"
  >CSS,</a
>
```

Write Code and give title

```
document.querySelector("a").setAttribute("title", "I am CSS");
```

After

```
<a
  class="src src2"
  id="mySrc"
  href="https://en.wikipedia.org/wiki/CSS"
  target="_blank"
  title="I am CSS"
  >CSS,</a
>
```

Hover over CSS hyperlink text to get a tooltip that says 'I am CSS'.

There are some attributes like `href`, `title`, `class` and `id` that we can update directly without using setAttribute:

eg.

`document.querySelector('a').id = 'myHeading' document.querySelector('a').className = 'myHeading2'`

Use className here and not class

```
<a
  href="https://en.wikipedia.org/wiki/Graphical_user_interface"
  id="myHeading"
  class="myHeading2"
>
  graphical user interface
</a>
```

How to apply styles in JavaScript

Changing color of text of all anchor elements to teal

```
const anchor = document.querySelectorAll("a");
// undefined

// node list to array
const anchorArr = [...anchor];
// undefined

anchorArr.map((col) => {
  col.style.color = "teal";
  col.style.textDecoration = "none";
  col.style.fontWeight = "700";
});
// (4) [undefined, undefined, undefined, undefined]
```

Shortcut way inside loop

```
anchorArr.map((col) => {
  col.style.cssText = `
  color: red;
  font-size: 18px;
  font-family: cursive;
  `;
});
```

Do like this to avoid repetition. `Also use backticks to write code in multiple lines`.

Selecting Classes using classList

eg

```
document.querySelector("#mySrc");
// <a class="src src2" id="mySrc" href="https://en.wikipedia.org/wiki/CSS" target=
"_blank">CSS,</a>

document.querySelector("#mySrc").classList;
// DOMTokenList(2) ['src', 'src2', value: 'src src2']

document.querySelector("#mySrc").classList.add("src3");
// undefined

document.querySelector("#mySrc").classList;
// DOMTokenList(3) ['src', 'src2', 'src3', value: 'src src2 src3']

document.querySelector("#mySrc").classList;
// DOMTokenList(3) ['src', 'src2', 'src3', value: 'src src2 src3']

document.querySelector("#mySrc").classList.remove("src3");
// undefined

document.querySelector("#mySrc").classList;
// DOMTokenList(2) ['src', 'src2', value: 'src src2']

document.querySelector("#mySrc").classList.toggle("src5");
// true

document.querySelector("#mySrc").classList;
// DOMTokenList(3) ['src', 'src2', 'src5', value: 'src src2 src5']
```

add() -> adds a class

remove() -> removes a class

toggle() -> If there is a class of the same name that matches the class inside (), then remove it, else add that class.

With this, we can write the css in styles.css and use the above code to add or remove the class which will apply the css or remove the css from our element in the document.

DOM Traversal | Access Parent Sibling & Children Elements

Accessing Parents

To access parent element of any element use `.parentElement`

Keep using .parentElement in chain to access parent of parent, parent of that parent and so on.

```
const myP = document.querySelector("#mySrc");

myP;
// anchor tag with id='mySrc'

myP.parentElement;
// p tag

myP.parentElement.parentElement;
// body tag

myP.parentElement.parentElement.parentElement;
// html tag
```

Accessing Children

To access the children of any element, use `children`.

```
const myP = document.querySelector("body");
// undefined

myP;
// <body style="font-family: sans-serif" class="vsc-initialized">…</body>

myP.children;
// HTMLCollection(8) [h1, h2, hr, p, img.myImg, ul, p, script]

// chaining
myP.children[3].children;
// HTMLCollection(5) [strong, a.hi.hi2.hi3, a.h1.h2, a#mySrc.src.src2, a, mySrc:
a#mySrc.src.src2]
```

Accessing Siblings

Siblings types:

- next sibling
- previous sibling

Also chaining is allowed here too!!

```
const myP = document.querySelector("#mySrc");

myP;
// Css anchor tag
```

```
myP.nextElementSibling;
// JS anchor tag

myP.nextElementSibling.nextElementSibling;
// null

myP.previousElementSibling;
// JS anchor tag

myP.previousElementSibling.previousElementSibling;
// CSS anchor tag
```

Siblings can come inside a parent only

here all `a tags` and a `strong tag` are inside the `p tag` (parent) hence these `a and strong tag are siblings`

```
<p>
  <strong> Frontend development </strong> is the development of the
  <a
    class="hi hi2 hi3"
    href="https://en.wikipedia.org/wiki/Graphical_user_interface"
  >
    graphical user interface
  </a>
  of a website, through the use of
  <a class="h1 h2" href="https://en.wikipedia.org/wiki/HTML" target="_blank">
    HTML,</a
  >

  <a
    class="src src2"
    id="mySrc"
    href="https://en.wikipedia.org/wiki/CSS"
    target="_blank"
    >CSS,</a
  >
  and
  <a href="https://en.wikipedia.org/wiki/JavaScript">JavaScript,</a>
  so that users can view and interact with that website.
</p>
```

`nextSibling, parentNode, childNodes, previousSibling -> All these give nodes not elements.`

What is the Difference Between Element and Node?

Nodes contain, tags, tag's text content, spaces, comments, new lines and many more things. So they are different from elements.

Use these methods to access parentNode, childNode, and sibling nodes:

```
`nextSibling, parentNode, childNodes, previousSibling` -> All these give nodes not elements.
```

Nodes are objects

Node name and Node type list

Name	Type Value
ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION_NODE	4
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11

All elements are nodes but all nodes are not elements

Eg -> Changing the text content of this text node

```
<h1>Frontend Development for KSD</h1>
"Hello There" // a text node
```

Code

```
document.body.childNodes;
// NodeList(19) [text, h1, text, h2, text, hr, text, p, text, img.myImg, text, ul, text, p, text, comment, text, script, text]0: textassignedSlot: nullbaseURI: "http://127.0.0.1:5502/html/projects/front-end-roadmap/index.html"childNodes: NodeList []data: "\n    "firstChild: nullisConnected: truelastChild: nulllength: 5nextElementSibling: h1nextSibling: h1nodeName: "#text"nodeType: 3nodeValue: "\n "ownerDocument: documentparentElement: body.vsc-initializedparentNode: body.vsc-initializedpreviousElementSibling: nullpreviousSibling: nulltextContent: "\n "wholeText: "\n    "[[Prototype]]: Text1: h12: text3: h24: text5: hr6: text7:
```

paccessKey: ""align: ""ariaAtomic: nullariaAutoComplete: nullariaBrailleLabel:
nullariaBrailleRoleDescription: nullariaBusy: nullariaChecked: nullariaColCount:
nullariaColIndex: nullariaColIndexText: nullariaColSpan: nullariaCurrent:
nullariaDescription: nullariaDisabled: nullariaExpanded: nullariaHasPopup:
nullariaHidden: nullariaInvalid: nullariaKeyShortcuts: nullariaLabel:
nullariaLevel: nullariaLive: nullariaModal: nullariaMultiLine:
nullariaMultiSelectable: nullariaOrientation: nullariaPlaceholder:
nullariaPosInSet: nullariaPressed: nullariaReadOnly: nullariaRelevant:
nullariaRequired: nullariaRoleDescription: nullariaRowCount: nullariaRowIndex:
nullariaRowIndexText: nullariaRowSpan: nullariaSelected: nullariaSetSize:
nullariaSort: nullariaValueMax: nullariaValueMin: nullariaValueNow:
nullariaValueText: nullassignedSlot: nullattributeStyleMap:
StylePropertyMap {size: 0}attributes: NamedNodeMap {length: 0}autocapitalize:
""autofocus: falsebaseURI: "http://127.0.0.1:5502/html/projects/front-end-
roadmap/index.html"childElementCount: 5childNodes: NodeList(11) [text, strong,
text, a.hi.hi2.hi3, text, a.h1.h2, text, a#mySrc.src.src2, text, a, text]children:
HTMLCollection(5) [strong, a.hi.hi2.hi3, a.h1.h2, a#mySrc.src.src2, a, mySrc:
a#mySrc.src.src2]classList: DOMTokenList [value: '']className: ""clientHeight:
37clientLeft: 0clientTop: 0clientWidth: 913contentEditable:
"inherit"currentCSSZoom: 1dataset: DOMStringMap {}dir: ""draggable:
falseeditContext: nullelementTiming: ""enterKeyHint: ""firstChild:
textfirstElementChild: stronghidden: falseid: ""inert: falseinnerHTML: "\n
 Frontend development is the development of the\n   <a
class=\"hi hi2 hi3\"
href=\"https://en.wikipedia.org/wiki/Graphical_user_interface\">\n
graphical user interface\n   \n   of a website, through the use of\n

HTML,\n\n   <a class=\"src src2\" id=\"mySrc\"
href=\"https://en.wikipedia.org/wiki/CSS\" target=\"_blank\">CSS,\n   and\n
JavaScript,\n   so
that users can view and interact with that website.\n   "innerText: "Frontend
development is the development of the graphical user interface of a website,
through the use of HTML, CSS, and JavaScript, so that users can view and interact
with that website."inputMode: ""isConnected: trueisContentEditable: falselang:
""lastChild: textlastElementChild: alocalName: "p"namespaceURI:
"http://www.w3.org/1999/xhtml"nextElementSibling: img.myImgnextSibling:
textnodeName: "P"nodeType: 1nodeValue: nullnonce: ""offsetHeight: 37offsetLeft:
8offsetParent: body.vsc-initializedoffsetTop: 182offsetWidth: 913onabort:
nullonanimationend: nullonanimationiteration: nullonanimationstart:
nullonauxclick: nullonbeforecopy: nullonbeforecut: nullonbeforeinput:
nullonbeforematch: nullonbeforepaste: nullonbeforetoggle: nullonbeforexrselect:
nullonblur: nulloncancel: nulloncanplay: nulloncanplaythrough: nullonchange:
nullonclick: nullonclose: nulloncontentvisibilityautostatechange:
nulloncontextlost: nulloncontextmenu: nulloncontextrestored: nulloncopy:
nulloncuechange: nulloncut: nullondblclick: nullondrag: nullondragend:
nullondragenter: nullondragleave: nullondragover: nullondragstart: nullondrop:
nullondurationchange: nullonemptied: nullonended: nullonerror: nullonfocus:
nullonformdata: nullonfullscreenchange: nullonfullscreenerror:
nullongotpointercapture: nulloninput: nulloninvalid: nullonkeydown:
nullonkeypress: nullonkeyup: nullonload: nullonloadeddata: nullonloadedmetadata:
nullonloadstart: nullonlostpointercapture: nullonmousedown: nullonmouseenter:
nullonmouseleave: nullonmousemove: nullonmouseout: nullonmouseover: nullonmouseup:
nullonmousewheel: nullonpaste: nullonpause: nullonplay: nullonplaying:
nullonpointercancel: nullonpointerdown: nullonpointerenter: nullonpointerleave:

```
nullonpointermove: nullonpointerout: nullonpointerover: nullonpointerrawupdate:
nullonpointerup: nullonprogress: nullonratechange: nullonreset: nullonresize:
nullonscroll: nullonscrollend: nullonsearch: nullonsecuritypolicyviolation:
nullonseeked: nullonseeking: nullonselect: nullonselectionchange:
nullonselectstart: nullonslotchange: nullonstalled: nullonsubmit: nullonsuspend:
nullontimeupdate: nullontoggle: nullontransitioncancel: nullontransitionend:
nullontransitionrun: nullontransitionstart: nullonvolumechange: nullonwaiting:
nullonwebkitanimationend: nullonwebkitanimationiteration:
nullonwebkitanimationstart: nullonwebkitfullscreenchange:
nullonwebkitfullscreenerror: nullonwebkittransitionend: nullonwheel:
nullouterHTML: "<p>\n      <strong> Frontend development </strong> is the
development of the\n      <a class=\"hi hi2 hi3\"
href=\"https://en.wikipedia.org/wiki/Graphical_user_interface\">\n
graphical user interface\n      </a>\n      of a website, through the use of\n
<a class=\"h1 h2\" href=\"https://en.wikipedia.org/wiki/HTML\" target=\"_blank\">
HTML,</a>\n\n      <a class=\"src src2\" id=\"mySrc\"
href=\"https://en.wikipedia.org/wiki/CSS\" target=\"_blank\">CSS,</a>\n      and\n
<a href=\"https://en.wikipedia.org/wiki/JavaScript\">JavaScript,</a>\n      so
that users can view and interact with that website.\n    </p>"outerText: "Frontend
development is the development of the graphical user interface of a website,
through the use of HTML, CSS, and JavaScript, so that users can view and interact
with that website."ownerDocument: documentparentElement: body.vsc-
initializedparentNode: body.vsc-initializedpart: DOMTokenList [value: '']popover:
nullprefix: nullpreviousElementSibling: hrpreviousSibling: textrole:
nullscrollHeight: 37scrollLeft: 0scrollTop: 0scrollWidth: 913shadowRoot: nullslot:
""spellcheck: truestyle: CSSStyleDeclaration {accentColor: '', additiveSymbols:
'', alignContent: '', alignItems: '', alignSelf: '', …}tabIndex: -1tagName:
"P"textContent: "\n       Frontend development  is the development of the\n
\n        graphical user interface\n      \n      of a website, through the use
of\n       HTML,\n\n      CSS,\n      and\n      JavaScript,\n      so that users
can view and interact with that website.\n    "title: ""translate:
truevirtualKeyboardPolicy: ""writingSuggestions: "true"[[Prototype]]:
HTMLParagraphElement(...)8: text9: img.myImg10: text11: ul12: text13: p14: text15:
comment16: text17: script18: textlength: 19[[Prototype]]: NodeList

document.body.childNodes[2];
// " "Hello There" "

console.dir(document.body.childNodes[2]);
// VM6520:1 #textassignedSlot: nullbaseURI:
"http://127.0.0.1:5502/html/projects/front-end-roadmap/index.html"childNodes:
NodeList []data: "\n    \"Hello There\"\n    "firstChild: nullisConnected:
truelastChild: nulllength: 23nextElementSibling: h2nextSibling: h2nodeName:
"#text"nodeType: 3nodeValue: "\n    \"Hello There\"\n    "ownerDocument:
documentparentElement: body.vsc-initializedparentNode: body.vsc-
initializedpreviousElementSibling: h1previousSibling: h1textContent: "\n
\"Hello There\"\n    "wholeText: "\n    \"Hello There\"\n    "[[Prototype]]: Text
// undefined

document.body.childNodes[2].nodeValue = "Changed Value";
// 'Changed Value'
```

append and appendChild?

appendChild()

appendChild -> Sabse last mein add karna

eg

```
const container = document.querySelector(".container");
const card = document.querySelector(".card");
const h1 = document.querySelector("h1");

container.appendChild(h1);
```

This is `cut and paste` i.e. it cuts the h1 tag from its place and pastes (appends) it after the card element, i.e. at the end of container element.

for `copy and paste`, we will use `cloneNode()`

This has the option to make a shallow copy or a deep copy

- Shallow Copy (only the tag)
- Deep Copy (tags + text inside tags)

eg

```
h1
// <h1>Append Child and Append</h1>

// Shallow Copy
h1.cloneNode()
// <h1></h1>

// Deep Copy -> (deep = true)
h1.cloneNode(true)
<h1>Append Child and Append</h1>
```

Here the copied element h1 will not show any changes,

So to make it show changes

So to do cut (copy and paste)

eg

```
const container = document.querySelector(".container");
const card = document.querySelector(".card");
```

```
const h1 = document.querySelector("h1");

container.appendChild(h1.cloneNode(true));
```

task -> Copy card inside container 100 times

```
const container = document.querySelector(".container");
const card = document.querySelector(".card");
const h1 = document.querySelector("h1");

// Already 1 is created so we do from 2 to 100
for (let i = 2; i < 101; i++) {
  const newCard = card.cloneNode();
  // document mein new card create nahi hua, memory mein copy create hua hai bas
  newCard.innerText = i;
  container.appendChild(newCard);
  // donot do document.appendChild as only one element is allowed on document
}
```

If we need to do same via html

```
.card*n{$} n -> no. of times we need a tag to be duplicated
```

`We can not append parents to children using appendChild()`

We can do it with other things like children, siblings etc.

append()

This is similar to appendChild() but with a few differences

1. append() does not return any value like appendChild() (appendChild returns us the text node that it has appended)
2. We can append things like strings, using append(), but we cannot do so via appendChild()

`container.append('string')`

we can append textNodes, elements and children and siblings using appendChild()

We can append all these using append() too but we can also append strings too!!

```
container.append("Hello World");
```

To append Strings via appendChild(), we need to first convert it into a text node.

i.e. create a new text node with the string

```
const newTextNode = document.createTextNode("Hello World");
container.appendChild(newTextNode);
```

This will work

3. We can also append multiple things at the same time using append()

```
container.append(h1, "Hello World", "three");
```

Append vs AppendChild

The append method appends elements as the last child of a parent element, while the appendChild method appends elements as the next sibling of the last child of a parent element.

Creating Elements using JavaScript

Task -> You have an image of a pokemon, clone that image, change the source and make 100s of those images.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <h1>Dom.create Element</h1>
    <div class="container">
      <img

src="https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/1.pn
g"
        alt="img1"
      />
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}
```

```
const h1 = document.querySelector("h1");
const container = document.querySelector(".container");

const img = document.querySelector(".container img");

for (let i = 2; i < 1010; i++) {
  const copy = img.cloneNode(true);
  copy.src =
`https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/${i}.png
`;
  container.appendChild(copy);
}

// make 100 clones, all with different src and append them all
```

Now if we did not have an image already, then what?? How would you clone it??

We can use `document.createElement()` for this, using this, we can create any element (this is an object so use console.dir to see its properties)

We can create any element (even custom elements too) using this property.

Custom elements are inline.

Task -> Create a 'p' element and append it:

```
const para = document.createElement("p");

document.body.append(para);
```

Write something in it

```
const para = document.createElement("p");
para.innerText = "Jai HO";

document.body.append(para);
```

Here we appended para in body so it goes in to the last place (after script, before body)

Alternatively, we can also append first then insert the innerText.

Adding some more things here like className, id, another class via classList

```
const h1 = document.querySelector("h1");
const container = document.querySelector(".container");

const img = document.querySelector(".container img");

const para = document.createElement("p");
para.innerText = "Jai HO";
para.id = "my-ID";
para.className = "ck";
para.classList.add("my-Class");

document.body.append(para);
```

Doing the pokemon task with create element

- use a loop
- create an img element
- set its source
- append it in container

```
const h1 = document.querySelector("h1");
const container = document.querySelector(".container");

for (let i = 1; i < 101; i++) {
  const image = document.createElement("img");
  image.src =
`https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/${i}.png
`;

  container.append(image);
}
```

Recognized Elements are created with all its attributes (properties).

Unknown elements are created in a different way.

Task -> From the previous task, we will create the images, and number them, each number below the image

- For this, we create another container, this will have image and text (do it inside loop)

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <h1>Dom.create Element</h1>
    <div class="container"></div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

.container-inner {
  display: flex;
  flex-direction: column;
}
```

```
const h1 = document.querySelector("h1");
const container = document.querySelector(".container");
// const container2 = document.querySelector('.container-inner')

for (let i = 1; i < 101; i++) {
  const container2 = document.createElement("div");
  container2.className = "container-inner";
  const image = document.createElement("img");
  const number = document.createElement("p");
  image.src =
`https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/${i}.png
`;
  number.innerText = `Image - ${i}`;
  number.style.color = "#000";
  number.style.textAlign = "center";
  container.append(container2);
  container2.append(image, number);
}
```

So when the container2 is created and its class is named, its CSS also becomes active.

We can also use,

```
const html = `
<img src="${i}" />
<p>${i}</p>
`;
container2.innerHTML = html;
```

To shorten our work but it is not recommended.

Another way to add 100 pokemons

```
const h1 = document.querySelector("h1");
const container = document.querySelector(".container");
// const container2 = document.querySelector('.container-inner')

let html = ``;

for (let i = 0; i <= 100; i++) {
  html += `
<div class='container-inner'>
  <img src="${i}" />
  <p>${i}</p>
</div>
`;
  container.innerHTML = html;
}
```

This creates a verry long string of html.

Removing Elements using JavaScript

remove()

Task -> Remove the 100th container from the pokemon code

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <h1>Remove</h1>
    <div class="container"></div>
```

```
  </body>
</html>
```

```
const h1 = document.querySelector("h1");
const container = document.querySelector(".container");
// const container2 = document.querySelector('.container-inner')

for (let i = 1; i < 101; i++) {
  const container2 = document.createElement("div");
  container2.className = "container-inner";
  const image = document.createElement("img");
  const number = document.createElement("p");
  image.src =
`https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/${i}.png
`;
  number.innerText = `Image - ${i}`;
  number.style.color = "#000";
  number.style.textAlign = "center";
  container.append(container2);
  container2.append(image, number);
}

// Remove the 100th pokemon container (image + text)
const hundPokemon = document.querySelector(".container-inner:nth-child(100)");
hundPokemon.remove();
```

removeChild()

Take the previous example

Syntax -> Go to parent of hundPokemon i.e. main container then remove its child i.e. hundPokemon container

`hundPokemon.parentElement.removeChild(hundPokemon);`

```
// Remove the 100th pokemon container (image + text)
const hundPokemon = document.querySelector(".container-inner:nth-child(100)");
hundPokemon.parentElement.removeChild(hundPokemon);
```

use remove() as this removeChild() is tricky

Here the code gets removed from DOM but not JS memory

To remove it from the JS memory, do:

- directly remove it, donot store it in a variable
- if storing in a variable
    - use `let` keyword then remove it

- set the variable value to `null` after removing, this clears the variable and it is only possible when we use let keyword while naming.

Event Listeners

1. Mouse Click

a. onClick()

Event is activated upon mouse click

egs.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card" onclick="console.log('HelloJi')">1</div>
    </div>
  </body>
</html>
```

eg2

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card" onclick="hello()">1</div>
    </div>
  </body>
</html>
```

```
function hello() {
  console.log("Function on Click heelo!!");
}
```

Here in eg2, we called the function in HTML inside onClick()

b. ondblclick (On Double Click)

eg

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card" ondblclick="hello()">1</div>
    </div>
  </body>
</html>
```

```
function hello() {
  console.log("Function on Double Click heelo!!");
}
```

Method 3 - ALL JS

Using methods like onClick() ondblclick()....

Now Earlier we wrote the function code in JS file and called the function in HTML. But by doing this, we were creating a new function when we called the function in HTML and passed the code in it from our JS file as both had same names. So now, we will try to write everything in JS.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
```

```
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card">1</div>
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
}
```

(CSS is same for all egs)

```
const card = document.querySelector(".card");

function hello() {
  console.log("Function on Double Click heelo!! Method 3");
}

card.ondblclick = hello;
```

`card.ondblclick = hello();` If we do this (add ()) then the text inside function will be executed and function is not called when trying to do so in an event.

addEventListener('event name', 'function name (for calling)')

same html & css code

```
const card = document.querySelector(".card");
```

```
function hello() {
  console.log("Function on Event Listener -  Click heelo!! Method Latest");
}

card.addEventListener("click", hello);
```

Using this, we can add multiple event listeners to the same element at once. Also using .addEventListener() we can execute all event code at once.

In the .onClick() or .ondblclick() method if we try to add multiple event listeners, then only the last one will be executed.

eg. With Anonymous Functions

same html & css code

```
const card = document.querySelector(".card");

card.addEventListener("click", function () {
  console.log(
    "Function on Event Listener -  Click heelo!! Method with Anonymous Function"
  );
});
```

eg -> 2 event listeners on 1 item

```
const card = document.querySelector(".card");

card.addEventListener("click", function () {
  console.log(
    "Function on Event Listener -  Click heelo!! Method with Anonymous Function"
  );
});

card.addEventListener("click", function () {
  console.log(
    "Function on Event Listener -  Click heelo!! Method with Anonymous Function
2nd one"
  );
});
```

When 2 event listeners different are used for same element

```
const card = document.querySelector(".card");

card.addEventListener("click", function () {
```

```
  console.log(
    "Function on Event Listener -  Click heelo!! Method with Anonymous Function"
  );
});

card.addEventListener("dblclick", function () {
  console.log(
    "Function on Event Listener -  Click heelo!! Method with Anonymous Function
2nd one with double click"
  );
});
```

Task -> Add empty cards whenever we click on the card

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card">+1</div>
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
  cursor: pointer;
}
```

```
const card = document.querySelector(".card");
const container = document.querySelector(".container");

card.addEventListener("click", function () {
  const newCard = card.cloneNode();
  container.append(newCard);
});
```

Another way

```
const card = document.querySelector(".card");
const container = document.querySelector(".container");

card.addEventListener("click", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  // giving the new div the same css as card by making their classes same
  container.append(newCard);
});
```

Numbering card

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card" title="Add new card">+</div>
    </div>
  </body>
</html>
```

task -> Card with numbering incremented

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
```

```
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
  cursor: pointer;
}

.card:nth-child(1) {
  width: 50px;
  height: 50px;
  background-color: blue;
  position: absolute;
  bottom: 15px;
  right: 15px;
  border-radius: 50%;
  text-align: center;
  color: red;
  line-height: 104px;
}
```

```
const card = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

card.addEventListener("click", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  count++;
  newCard.innerText = count;
  container.append(newCard);
});
```

Form Event and Event Object

Now in form inputs, they are handled a bit differently.

`input` event

Here we have an `input` event listener.

But to access the form input value, we need an `event object (e)`

So we pass this `e` as the function parameter inside the addEvent Listener function

eg.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Form Events</title>
    <script src="script.js" defer></script>
  </head>
  <body>
    <h1>Form Events</h1>
    <form>
      <input type="text" id="myInput" />
    </form>
  </body>
</html>
```

```
const textInp = document.querySelector("#myInput");

textInp.addEventListener("input", function (e) {
  console.log(e);
  console.log("Hello World");
});
```

We see that everytime, we type something from our keyboard, this `e` object is fired i.e. console.logged. (Also the Hello World is console.logged).

`We can give it any name other than e but usually people give it, e, evt, or event`

Inside this event object we have a `target` attribute, it means -> Kis cheez ke upar event hua hai (here text input field is the target, and event is being applied to it).

Inside the target, we have a value attribute, this gives us the value entered inside the text input field.

So to access the input text field value, -> Do `e.target.value`

```
const textInp = document.querySelector("#myInput");

textInp.addEventListener("input", function (e) {
  // console.log(e);
  // console.log(e.target);
  console.log(e.target.value);
```

```
  // console.log('Hello World');
});
```

Now we can manipulate this input value. Store it in a variable and do as you please.

`change` event

In this event type when ever we make some changes in the input field and then go out of it, then only we will see the change event firing, unlike the input event which gets fired on every input value entered.

Same HTML Code

```
const textInp = document.querySelector("#myInput");

textInp.addEventListener("change", function (e) {
  // console.log(e);
  // console.log(e.target);
  console.log(e.target.value);
  // console.log('Hello World');
});
```

When we go inside the input field, then the input is in `focus`

When we go out of the input field, then the input is in `blur`

`focus` event

This event gets fired whenever the input field is `in focus`

```
const textInp = document.querySelector("#myInput");

textInp.addEventListener("focus", function (e) {
  // console.log(e);
  // console.log(e.target);
  console.log(e.target.value);
  // console.log('Hello World');
});
```

eg.

- Empty field (nothing)
- Click on the input field (we get an empty output printed)
- Type `apple`
- Go out
- Focus it again by clicking on the input field (apple is printed).

`blur` event

Works exactly opposite to focus event

```
const textInp = document.querySelector("#myInput");

textInp.addEventListener("blur", function (e) {
  // console.log(e);
  // console.log(e.target);
  console.log(e.target.value);
  // console.log('Hello World');
});
```

Till now, we were only using the input fields alone. But what if we enclose it in a `<form></form>` tag

How the form tag works,

so say if we have a form with 2 inputs (name and email) and a submit button. Every time, you click a submit button, the form will get submitted and the site will reload. Now where do you want to send the form after submitting, depends on the `action` attribute. So it will go to the action attribute value url after submitting. Now there will be search params included, to include those search params, we will use the name attribute and the value of the name attribute will serve as the title and the value of the input field will serve as the content of the search params.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Form Events</title>
    <script src="script.js" defer></script>
  </head>
  <body>
    <h1>Form Events</h1>
    <form action="www.googly.com">
      <input type="text" id="myInput" name="username" />
      <input type="email" id="myEmail" name="usermail" />
      <button type="submit">Click to Submit</button>
    </form>
  </body>
</html>
```

```
http://127.0.0.1:5502/javascript/www.googly.com?
username=ll&usermail=op%40gmail.com
```

By default button inside of a form is of type `submit` which submits the form when we click it.

%40 = @

`submit` event

Happens when we submit a form

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Form Events</title>
    <script src="script.js" defer></script>
  </head>
  <body>
    <h1>Form Events</h1>
    <form action="www.google.com">
      <input type="text" id="myInput" name="username" />
      <input type="email" id="myEmail" name="usermail" />
      <button type="submit">Click to Submit</button>
    </form>
  </body>
</html>
```

```
const textInp = document.querySelector("#myInput");
const form = document.querySelector("form");

form.addEventListener("submit", function (e) {
  e.preventDefault();
  console.log(e);
  console.log(e.target);
  console.log(e.target.value);
  console.log("Hello World");
});
```

Now to prevent the form from reloading and redirecting us to another site (the form's default behavior), we can do `e.preventDefault()`. This prevents the form's default behavior.

Event.target value depends on the element on which we fired an event, so if in a form we click the input text, then e.target = input text and not the form.

Event.currentTarget value does not change in this case and it gives us the value on which the event is started (here we started firing the event on form) but inside form we also fire events, so e.target keeps changing but e.currentTarget is same.

What is the difference between an event listener and an event handler in JavaScript?

An event listener is a function that is called when an event occurs, while an event handler is a function that is called when an event is triggered.

Keyboard events in JavaScript

Keyboard Events:

- Key Up
- Key Down
- Key Press

Key Press

We can put a keyboard event on any element, but it has to be in focus.

Some eleemnts are automatically in focus like `<input>` but for non-focusable elements, to get them in focus, we can do in html is add a `tabindex = '0'` attribute to the non focusable element.

Then click that element in window to get them in focus and then start pressing the keys.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Keyboard Events</title>
    <script src="./script.js" defer></script>
  </head>
  <body>
    <h1 tabindex="0">Keyboard Events</h1>
  </body>
</html>
```

```
const h1 = document.querySelector("h1");

h1.addEventListener("keypress", (e) => {
  console.log(e.key);
  console.log(e.code);
});
```

e.key: Whenever we need the value of a key, when we do keypress, we use `e.key`

e.value: Whenever we need the code of a key, when we do keypress, we use `e.code`

keyup

Works when we lift a pressed key.

same html

```
const h1 = document.querySelector("h1");

h1.addEventListener("keyup", (e) => {
  console.log(e.key);
  console.log(e.code);
});
```

In keyup, keys such as shift, alt, ctrl, and arrow keys work but in keypress, they donot work.

keydown

Works when we press a key down.

```
const h1 = document.querySelector("h1");

h1.addEventListener("keydown", (e) => {
  console.log(e.key);
  console.log(e.code);
});
```

keydown vs keypress

- In keydown, keys such as shift, alt, ctrl, and arrow keys work but in keypress, they donot work.
- In keypress, it takes both keydown + keyup but in keydown it takes only keydown.

If the keyboard language changes, then the key value also changes.

Mouse Events in JavaScript

We have already seen `click` and `dblclick` event for mouse events

Let us see some more

1. mousedown

Mouse has a button inside it so, when that button is pressed down, we see this event get fired.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
    <title>Add cards</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <div class="container">
      <div class="card" title="Add new card">+</div>
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
  cursor: pointer;
}

.card:nth-child(1) {
  width: 50px;
  height: 50px;
  background-color: blue;
  position: absolute;
  bottom: 15px;
  right: 15px;
  border-radius: 50%;
  text-align: center;
  color: red;
  line-height: 104px;
}
```

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("mousedown", function () {
```

```
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

2. mouseup

This event is triggered when the pressed mouse button is gone up.

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("mouseup", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

3. mouseenter

Gets triggered when we hover in over the target element

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("mouseenter", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

4. mouseleave

Gets triggered when we hover out of over the target element

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
```

```
let count = 1;

addCardBtn.addEventListener("mouseleave", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

5. mousemove

Whenever we move our mouse inside the target element, this event will get triggered.

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("mousemove", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

6. mouseout

More sensitive than mouseleave

Difference between mouseout and mouseleave

In JavaScript, both `mouseout` and `mouseleave` events are used to detect when the mouse pointer leaves an element, but there are key differences between them:

Bubbling:

- mouseout: This event bubbles up through the DOM. When the pointer moves from an element to a child element or a different element within the same parent, it triggers a mouseout event on the element and then bubbles up to its ancestors.
- mouseleave: This event does not bubble. It only fires when the mouse leaves the boundary of the element itself, not when moving between child elements within it.

Triggering:

- mouseout: Fires when the mouse leaves the element or any of its children. This means if the cursor moves from a parent element to a child element, a mouseout event is triggered on the parent.
- mouseleave: Fires only when the mouse leaves the element entirely, regardless of child elements. Moving to a child element does not trigger this event on the parent.

Example Consider this structure:

```
<div class="parent">
  <div class="child">Child Element</div>
</div>
```

If you attach a mouseout event listener to .parent, moving from .parent to .child will trigger the mouseout event on .parent.

If you attach a mouseleave event listener to .parent, it will only fire if you move the mouse entirely out of .parent, not when moving to .child.

When to Use Each

Use mouseout if you need to detect when the mouse moves to any other element, including child elements.

Use mouseleave if you only want to detect when the mouse has left the element entirely, ignoring any movements to child elements.

This distinction can help manage events more precisely, especially in nested or complex UI components.

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("mouseout", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

7. mouseover

Same as mouseenter but with some key differences

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;
```

```
addCardBtn.addEventListener("mouseover", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

Differences

The `mouseover` and `mouseenter` events in JavaScript are similar to mouseout and mouseleave, with key differences mainly related to event bubbling and triggering:

Bubbling:

- mouseover: This event bubbles up through the DOM. When the mouse enters an element or any of its children, the mouseover event triggers on that element and bubbles up to its ancestors.
- mouseenter: This event does not bubble. It triggers only when the mouse enters the element itself, not when entering any child elements.

Triggering:

- mouseover: Fires when the mouse enters an element or any of its child elements. Moving the mouse from the parent to a child element will trigger a mouseover event on the parent.
- mouseenter: Only fires when the mouse enters the boundary of the element itself, ignoring any child elements. Moving the mouse into a child element will not re-trigger mouseenter on the parent.

Example

Consider this HTML structure:

```
<div class="parent">
  <div class="child">Child Element</div>
</div>
```

- If you attach a mouseover event listener to .parent, moving the mouse from .parent to .child will trigger the mouseover event on .parent.
- If you attach a mouseenter event listener to .parent, it will only fire the first time the mouse enters .parent itself, not when moving into .child.

When to Use Each

Use mouseover if you want to detect when the mouse enters an element or any of its child elements.

Use mouseenter if you want to detect when the mouse enters only the specific element, ignoring any children.

In summary, mouseover and mouseenter are useful for different cases, depending on whether child elements should trigger the event or not.

8. wheel

Works when we scroll up or down inside the target element.

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("wheel", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

9. Scroll Event

Happens in the window object, see the difference between wheel and scroll more clearly below.

In JavaScript, both scroll and wheel events are related to detecting scroll behavior, but they have distinct purposes and use cases:

1. scroll

- Description: The scroll event fires when an element or the document itself is scrolled, regardless of the method used to scroll (e.g., mouse wheel, trackpad, keyboard arrows, dragging scrollbars, or programmatically).
- Trigger: Occurs whenever scrolling happens within an element (like a div with overflow: auto;) or the entire page.
- Applicable To: Elements that are scrollable, including window (for the entire page).
- Typical Usage: Used for actions that depend on scroll position, such as lazy loading, infinite scrolling, or updating a scroll-based animation.

Example:

```
window.addEventListener("scroll", () => {
  console.log("The page was scrolled.");
});
```

2. wheel

- Description: The wheel event specifically detects the use of a mouse wheel or a similar input device (e.g., a touchpad with a two-finger scroll gesture) to scroll within an element or the page. It provides additional details like the amount of scroll (delta values) and the direction.
- Trigger: Fires when the mouse wheel or similar scroll gesture is used. This event is granular and fires for each step or unit of scroll input.
- Applicable To: Any scrollable element or the window object, but it only fires if scroll input comes from a wheel or trackpad gesture, not keyboard or other scroll methods.
- Typical Usage: Used for custom scroll behaviors, fine control over scroll speed, or when you need detailed scroll information. The wheel event provides deltaX, deltaY, and deltaZ values, which give the exact amount and direction of scroll input, enabling more precise adjustments.

Example:

```
window.addEventListener("wheel", (event) => {
  console.log(`Scroll amount - X: ${event.deltaX}, Y: ${event.deltaY}`);
});
```

- Key Differences

Feature	scroll	wheel
Trigger Source	Any scroll action	Mouse wheel or similar gestures
Frequency	Lower, per scroll update	Higher, per small scroll step
Event Data	No detailed scroll data	Includes deltaX, deltaY, etc.
Use Cases	Scroll-based animations, lazy loading	Custom scroll handling, precise adjustments
When to Use Each	Use scroll for general scroll events that work across all scroll actions (keyboard, dragging, etc.).	Use wheel for handling scroll input directly from a mouse wheel or touchpad gesture when you need more detailed control over scroll behavior or need the exact scroll increments.

10. drag (works only in desktop and not in mobile)

This event can be used to make to-do lists etc. i.e. event is fired when we drag an element.

For that add the following attribute to the target element `draggable="true"` in html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Add cards</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <h1 draggable="true">Drag Me</h1>
    <div class="container">
      <div class="card" title="Add new card">+</div>
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

h1 {
  background-color: blue;
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
  cursor: pointer;
}

.card:nth-child(1) {
  width: 50px;
  height: 50px;
  background-color: blue;
  position: absolute;
  bottom: 15px;
  right: 15px;
  border-radius: 50%;
  text-align: center;
  color: red;
  line-height: 104px;
}
```

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

h1.addEventListener("drag", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

Some more mouse events but for mobile devices (to check in desktop, use dynamic resizing of devices in inspect, responsive design)

1. touchstart

Activates when we touch or tab any element. Works like keydown.

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("touchstart", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

2. touchend

Works in the same way but triggers when we lift the touch up, like keyup.

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("touchend", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
```

```
  container.append(newCard);
});
```

3. touchmove

Touch the button and then move i.e. press and move

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");

let count = 1;

addCardBtn.addEventListener("touchmove", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

Events for both desktop and mobile

1. pointermove (mousemove + touchmove => Works for both desktop and mobile)

For testing in mobile, press and then drag around the button

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

addCardBtn.addEventListener("pointermove", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

2. pointerdown (mousedown + touchdown)

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");
```

```
let count = 1;

addCardBtn.addEventListener("pointerdown", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

3. pointerleave (touchleave + mouseleave)

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

addCardBtn.addEventListener("pointerleave", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

Event Bubbling and Event Capturing in JavaScript

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Event Bubbling and Event Capturing</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <div class="green">
    <div class="pink">
      <div class="yellow"></div>
    </div>
  </div>
</html>
```

```css
html,
body {
  margin: 0;
  padding: 0;
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
}

.green {
  background-color: green;
  width: 500px;
  height: 250px;
  display: flex;
  justify-content: center;
  align-items: center;
}

.pink {
  background-color: pink;
  width: 300px;
  height: 150px;
  display: flex;
  justify-content: center;
  align-items: center;
}

.yellow {
  background-color: yellow;
  width: 120px;
  height: 60px;
}
```

```js
const green = document.querySelector(".green");
const pink = document.querySelector(".pink");
const yellow = document.querySelector(".yellow");
const body = document.querySelector("body");

green.addEventListener("click", function (e) {
  console.log("Green Box Clicked");
});

pink.addEventListener("click", function (e) {
  console.log("Pink Box Clicked");
});

yellow.addEventListener("click", function (e) {
  console.log("Yellow Box Clicked");
});
```

```
body.addEventListener("click", function (e) {
  console.log("Body Clicked");
});

document.addEventListener("click", function (e) {
  console.log("Html Document Clicked");
});

window.addEventListener("click", function (e) {
  console.log("Window Clicked");
});

/*
const green = document.querySelector('.green');
const pink = document.querySelector('.pink');
const yellow = document.querySelector('.yellow');
const body = document.querySelector('body');


yellow.addEventListener('click', function(e) {
    console.log('1. Yellow Box Clicked');
    // e.stopPropagation();

});


pink.addEventListener('click', function(e) {
    console.log('2. Pink Box Clicked');
    // e.stopPropagation();

});

green.addEventListener('click', function(e) {
    console.log('3. Green Box Clicked');
    // e.stopPropagation();

});


body.addEventListener('click', function(e) {
    console.log('4. Body Clicked');
    e.stopPropagation();

});

document.addEventListener('click', function(e) {
    console.log('5. Html Document Clicked');
    // e.stopPropagation();

});

window.addEventListener('click', function(e) {
    console.log('6. Window Clicked');
```

```
    // e.stopPropagation();
});


/*
Output:

1. Yellow Box Clicked

2. Pink Box Clicked

3. Green Box Clicked

4. Body Clicked

5. Html Document Clicked

6. Window Clicked

*/
```

Here normally on any event listener when we interact with a child here, i.e. when we click on yellow here we will see that we have outputs of yellow box clicked, pink box clicked, and green box clicked.

So we are observing event bubbling here it starts from a child and goes on till the outermost parent i.e. till window object.

Flow of event bubbling here:

yellow > pink > green > body > document (html) > window (last parent)

To stop this event from bubbling, we do `e.stopPropagation()` on a child to prevent it to bubble up to its parent.

Now when we click on yellow, we will see only (yellow button clicked) on output.

Same with other cases

Event Capturing

This is reverse of event bubbling, i.e. it starts from outermost parent (window) and comes to innermost child (yellow here)

For using this event capturing, we will do the following: (mark the last parameter as true).

```
const green = document.querySelector(".green");
const pink = document.querySelector(".pink");
const yellow = document.querySelector(".yellow");
const body = document.querySelector("body");

yellow.addEventListener(
```

```
  "click",
  function (e) {
    console.log("1. Yellow Box Clicked");
    // e.stopPropagation();
  },
  true
);

pink.addEventListener(
  "click",
  function (e) {
    console.log("2. Pink Box Clicked");
    // e.stopPropagation();
  },
  true
);

green.addEventListener(
  "click",
  function (e) {
    console.log("3. Green Box Clicked");
    // e.stopPropagation();
  },
  true
);

body.addEventListener(
  "click",
  function (e) {
    console.log("4. Body Clicked");
    // e.stopPropagation();
  },
  true
);

document.addEventListener(
  "click",
  function (e) {
    console.log("5. Html Document Clicked");
    // e.stopPropagation();
  },
  true
);

window.addEventListener(
  "click",
  function (e) {
    console.log("6. Window Clicked");
    // e.stopPropagation();
  },
  { capture: true }
);

/*
```

```
Output:

6.Window Clicked

5. Html Document Clicked

4. Body Clicked

3. Green Box Clicked

2. Pink Box Clicked

1. Yellow Box Clicked

*/
```

(Event bubbling is false (by default))

This can also be stopped by using `e.stopPropagation()`

Do it in innermost child to get only the outermost output and vice versa.

Another Way of writing event capture

```
window.addEventListener(
  "click",
  function (e) {
    console.log("6. Window Clicked");
    // e.stopPropagation();
  },
  { capture: true }
);
```

```
window.addEventListener(
  "click",
  function (e) {
    console.log("6. Window Clicked");
    // e.stopPropagation();
  },
  { capture: true, once: true }
);
```

This makes the event listener active for only one click.

Applications - Useful in making pop ups.

What is the difference between addEventListener() and attachEvent() in JavaScript?

`addEventListener()` is used for event bubbling, while `attachEvent()` is used for event capturing

Event Simulation in JavaScript

In javascript we normally trigger events via mouse or keyboard, but we can also trigger them using js code, this is called `event simulation`, however, this is not an official term.

Adding 1000 cards using js code

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Add cards</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <!-- <h1 draggable="true">Drag Me</h1> -->
    <div class="container">
      <div class="card" title="Add new card">+</div>
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

h1 {
  background-color: blue;
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
  cursor: pointer;
}
```

```
.card:nth-child(1) {
  width: 50px;
  height: 50px;
  background-color: blue;
  position: absolute;
  bottom: 15px;
  right: 15px;
  border-radius: 50%;
  text-align: center;
  color: red;
  line-height: 104px;
}
```

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

addCardBtn.addEventListener("click", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});

for (let i = 0; i < 1000; i++) {
  addCardBtn.click();
}
```

Slowly adding 1000 cards (using setInterval() => 1 card per m sec)

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

addCardBtn.addEventListener("click", function () {
  const newCard = document.createElement("div");
  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});

const intervalId = setInterval(function () {
```

```
  if (count > 999) {
    clearInterval(intervalId);
    // Stop when count reaches 1000
  }
  addCardBtn.click();
}, 1);
```

Similarly we have `input.focus()`, this will bring an input to focus without clicking on it.

We also have `input.blur()`, `form.submit()`, `form.reset()` etc.

Use setTimeout to execute them after some delay.

For `form.submit()` => button should be of type submit and for `form.reset()` => button should be of type reset.

Event Delegation in JavaScript

Let us do a simple task, when the user clicks on a particular card, the card should get removed.

One way, add the event listener of remove before you append the card

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Add cards</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <!-- <h1 draggable="true">Drag Me</h1> -->
    <div class="container">
      <div class="card" title="Add new card">+</div>
    </div>
  </body>
</html>
```

```
.container {
  color: green;
  display: flex;
  flex-wrap: wrap;
  gap: 8px;
}

h1 {
```

```
  background-color: blue;
}

.card {
  width: 100px;
  height: 120px;
  background-color: pink;
  display: flex;
  align-items: center;
  justify-content: center;
  border-radius: 4px;
  font-size: 36px;
  cursor: pointer;
}

.card:nth-child(1) {
  width: 50px;
  height: 50px;
  background-color: blue;
  position: absolute;
  bottom: 15px;
  right: 15px;
  border-radius: 50%;
  text-align: center;
  color: red;
  line-height: 104px;
}
```

```
const addCardBtn = document.querySelector(".card");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

addCardBtn.addEventListener("click", function () {
  const newCard = document.createElement("div");
  newCard.addEventListener("click", () => {
    newCard.remove();
  });

  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});
```

But by using this method, this will create multiple event listeners in our code which is not optimized, so let us try to use another method:

IMP INTERVIEW QUESTION => Putting the event listener in the parent container instead of putting it in each individual child (Event Delegation)

This method reduces the event listeners and saves a lot of memory

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Add cards</title>
    <link rel="stylesheet" href="styles.css" />
    <script src="script.js" defer></script>
  </head>
  <body style="font-family: cursive">
    <!-- <h1 draggable="true">Drag Me</h1> -->
    <div class="container">
    </div>
    <div class="card" title="Add new card" id="plus">+</div>

  </body>
</html>
```

```
.container {
    color: green;
    display: flex;
    flex-wrap: wrap;
    gap: 8px;
}

h1 {
    background-color: blue;
}

.card {
    width: 100px;
    height: 120px;
    background-color: pink;
    display: flex;
    align-items: center;
    justify-content: center;
    border-radius: 4px;
    font-size: 36px;
    cursor: pointer;
}

#plus {
    width: 50px;
    height: 50px;
```

```
    background-color: blue;
    position: absolute;
    bottom: 15px;
    right: 15px;
    border-radius: 50%;
    text-align: center;
    color: red;
    line-height: 104px;
}
```

```
const addCardBtn = document.querySelector("#plus");
const container = document.querySelector(".container");
const h1 = document.querySelector("h1");

let count = 1;

addCardBtn.addEventListener("click", function () {
  const newCard = document.createElement("div");

  newCard.classList.add("card");
  newCard.innerText = count;
  count++;
  container.append(newCard);
});


// Adding the event listener to the container (parent)
container.addEventListener('click', (e) => {
  if(e.target !== container) {
    e.target.remove();
  }
})
```

`e.target` ensures here on which item we have clicked, so in e.target we have the container itself and the cards that are created inside the container. So we want to remove only when we click on the cards and not when we click on the container hence specified. (We took out the + button outside the container hence, see html)

What is the primary benefit of event delegation in JavaScript?

A. Optimisizing memory usage and page load

Local Storage

There is a local storage object present inside the window object.

Each website has a separate local storage.

Go to devtools -> Application => Local Storage

Benifit of local storage is that the value will be saved even after the page is reloaded.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Local Storage</title>
  <script src="./script.js" defer></script>
</head>
<body>
  <input type="text" id="name" placeholder="Enter your name" />
  <p>Your name is : <span class="my-name"></span></p>
</body>
</html>
```

Old way

```
const input = document.querySelector("#name");
const nameSpan = document.querySelector(".my-name");

nameSpan.innerText = localStorage.myName;

input.addEventListener("input", (e) => {
  localStorage.myName = e.target.value;
  nameSpan.innerText = localStorage.myName;
});
```

New and Optimized way

```
const input = document.querySelector("#name");
const nameSpan = document.querySelector(".my-name");

// Update the local storage with the value of input and put that value in span


// fetch from local storage
nameSpan.innerText = localStorage.getItem("myName");

input.addEventListener("input", (e) => {
  // update the local storage
  localStorage.setItem("myName", e.target.value);

// fetch from local storage again
  nameSpan.innerText = localStorage.getItem('myName');
```

```
});
```

One small difference when we try to access a key which is not there in localstorage

```
localStorage.getItem('age') => null

localStorage.age => undefined
```

age is not a valid key here

LocalStorage data only fails to get retained in Incognito tab when we try to close one incognito tab and open another incognito tab.

Using multiple different inputs with local storage

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Local Storage</title>
  <script src="./script.js" defer></script>
</head>
<body>
  <input type="text" id="name" placeholder="Enter your name" name="name" />
  <input type="number" id="age" placeholder="Enter age" name="age" />
  <p>Your name is : <span class="my-name"></span></p>
  <p>I am <span class="my-age"></span> years old</p>
</body>
</html>
```

```
const input = document.querySelector("#name");
const nameSpan = document.querySelector(".my-name");
const ageInput = document.querySelector("#age");
const ageSpan = document.querySelector(".my-age");

nameSpan.innerText = localStorage.getItem('myName');

input.addEventListener("input", (e) => {
  localStorage.setItem('myName', e.target.value);
  nameSpan.innerText = localStorage.getItem('myName');
});

ageSpan.innerText = localStorage.getItem('myAge');

ageInput.addEventListener("input", (e) => {
  localStorage.setItem('myAge', e.target.value);
```

```
  ageSpan.innerText = localStorage.getItem('myAge');
});
```

We can only store strings in local storage.

Convert object to string (JSON string) => `JSON.stringify(object)`

Convert JSON string back to object => `JSON.parse(JSON String)`

Now here we are using a single object key and storing various keys inside it:

We are not using the getItem and setItem methods completely here, see the code

```
const nameInput = document.querySelector("#name");
const nameSpan = document.querySelector(".my-name");
const ageInput = document.querySelector("#age");
const ageSpan = document.querySelector(".my-age");

// We will try to fetch(get) the object data from local storage if it already
exists, else we will create an empty object

const myData = JSON.parse(localStorage.getItem("myData")) || {};

// now we try to import the name and age from this object if it already exists
if (myData.myName) {
  nameSpan.innerText = myData.myName;
}

if (myData.myAge) {
  ageSpan.innerText = myData.myAge;
}


nameInput.addEventListener("input", (e) => {
  myData.myName = e.target.value;
  localStorage.setItem('myData', JSON.stringify(myData));
  nameSpan.innerText = e.target.value;
});


ageInput.addEventListener("input", (e) => {
  myData.myAge = e.target.value;
  localStorage.setItem('myData', JSON.stringify(myData));
  ageSpan.innerText = e.target.value;
});
```

Now suppose we have multiple keys inside of localStorage and we want to remove them all then we do `localStorage.clear()`

If we want to delete a particular key object from the localStorage, then we do:
`localStorage.removeItem('keyName')`

Keep in mind, that the URL of the localStorage data is unique for each site.

```
Q. What is the difference between local storage and session storage in JavaScript?

A. Local storage stores data persistently, while session storage stores data
temporarily


Q. What is the primary purpose of local storage in JavaScript?

A. To store data persistently (continously)
```
