BEM (Block, Element, Modifier) is a popular naming convention for organizing CSS classes in a modular and maintainable way. It helps to make the relationship between HTML structure and styles clearer by breaking down components into distinct parts.

1. \*\*Block\*\*

The standalone entity or component, which is meaningful on its own. Examples include `header`, `menu`, `button`, or `form`.

\*\*Example:\*\*

```html

<div class="menu"></div>

2. \*\*Element\*\*

A part of a block that has no standalone meaning and is dependent on the block. It’s always tied to a specific block and is written after the block name, separated by two underscores (`\_\_`).

\*\*Example:\*\*

```html

<div class="menu\_\_item"></div>

<div class="menu\_\_link"></div>

3. \*\*Modifier\*\*

A flag that changes the appearance, behavior, or state of a block or element. It's written after the block or element name, separated by two hyphens (`--`)

\*\*Example:\*\*

```html

<div class="menu menu--compact"></div> <!-- Modifier applied to the block -->

<div class="menu\_\_item menu\_\_item--active"></div> <!-- Modifier applied to the element -->

### Key Benefits:

- \*\*Consistency:\*\* Clear rules for naming CSS classes.

- \*\*Readability:\*\* Makes it easier to understand the relationship between HTML elements and styles.

- \*\*Scalability:\*\* Works well in large projects as it's easy to manage and update components.

By following BEM, you make your styles reusable and avoid naming conflicts.

Difference between const, var and let:  
- const : declare once and assign once ( ref is same but is mutable for ref based data types)  
- let: declare once and reassign multiple times  
- var: declare multiple times and can be reassigned multiple times. So not safe

InnerHTML: this property is used to get or set the HTML content inside an element on a webpage. It allows you to manipulate the DOM ( Document Object Model ) by directly changing the content, including any HTML tags, inside a specific element.

How innerHTML Works:

Getting innerHTML: You can use innerHTML to retrieve the HTML content of an element as a string.  
let content = element.innerHTML;

Setting innerHTML: You can use innerHTML to replace the content of an element with new HTML, which can include tags, text, and other elements.  
element.innerHTML = 'new content here';

Ex:   
<!DOCTYPE html>

<html lang="en">

<head>

<title>Get innerHTML Example</title>

</head>

<body>

<div id="myDiv">This is <strong>bold</strong> text.</div>

<script>

let myDiv = document.getElementById('myDiv');

console.log(myDiv.innerHTML); // Outputs: **This is <strong>bold</strong> text.**

</script>

</body>

</html>

**Use Cases of innerHTML:**

* **Displaying dynamic content**: You can dynamically update the content of your page without reloading it.
* **Injecting HTML content**: It can be used to insert HTML elements, like paragraphs, images, or buttons.
* **Creating or modifying interactive elements**: Useful for adding buttons, forms, etc., based on user interaction or responses from an API.

How to define functions: 2 ways to define the functions  
1. Arrow Function : const incrementCount = ( ) => { };  
An arrow function is a more concise way to write a function. It is defined using any variable (var, const, let)  
**Key Characteristics of Arrow Functions:**

* **No this binding**: Arrow functions do not have their own **this** context. They inherit **this** from the surrounding (lexical) context. This is one of the major differences from regular functions.
* **No arguments object**: Arrow functions do not have their own **arguments** object. You can still access arguments via rest parameters (...args), but not through **arguments**.
* **Can't be used as constructors**: You cannot use arrow functions with the **new** keyword to create instances (i.e., arrow functions cannot be constructors).
* **Must be defined before use**: Arrow functions are not hoisted. You need to define them before you can use them.

2. Regular function: function incrementCount( ) { };  
**Key Characteristics of Regular Functions:**

* **Own this binding**: Regular functions have their own this context, which is determined based on how the function is called. If used as a method inside an object, this refers to the object. If called in the global scope, this refers to the global object (in non-strict mode).
* **Own arguments object**: Regular functions have access to the arguments object, which contains all the arguments passed to the function.
* **Can be used as constructors**: Regular functions can be used as constructors, which means you can use them with the new keyword to create object instances.
* **Function hoisting**: Regular functions are hoisted to the top of their scope. This means you can call the function before its declaration in the code.

**Summary of Key Points:**

* **Arrow functions** are useful when you want to inherit this from the surrounding context (often in event handlers or callbacks).
* **Regular functions** provide more flexibility with this binding, can be hoisted, and have access to the arguments object.

**Map() function:** The **map()** function in JavaScript is an array method that creates a **new array** by calling a provided function on every element in the array it is invoked on.  
Syntax: const newArray = array.map((element, index, array) => {

// Return a value for the new array

});  
**Parameters:**

1. **element** – The current element being processed in the array.
2. **index** (optional) – The index of the current element being processed.
3. **array** (optional) – The array map() was called upon.

**Key Features of map():**

* It **does not modify** the original array (i.e., it's non-mutative).
* It returns a **new array** based on the transformation you apply to each element.
* The callback function can perform any operation and return any type of value for each element.

**Difference Between map() and forEach()**

* **map()** returns a **new array** containing the results of applying the function to each element.
* **forEach()** is used purely for iteration and **does not return** anything (i.e., it returns undefined).

## Events:

Some events are inbuilt in JS like highlighting or opening a link in a new window. So when working with events, if something doesn’t work then look for defaults.

Event.preventDefault is the function to use then.

Events works differently in JS and React.

React does Virtual DOM, so whenever there’s an event . In Js the whole DOM tree is refreshed while React only refreshes the subtree where the event took place. So the rendering of the DOM is very fast in React with the help of mirroring DOM and manipulating it called virtual DOM. So the performance is better for bigger no. of operations.

Another Advantage of React is , we can have reusable components.

Event Bubbling: event is propagated to the parent and grand parent and so on until root for any event occurring at any sub tree. It’s propagated through nodes.

\*\*Event bubbling\*\* is a concept in the DOM (Document Object Model) event model, where an event starts at the most specific element (the target element) and then propagates upwards through the ancestor elements (from child to parent) until it reaches the root element (usually the `document` object).

### How it Works:

When an event (like a click) occurs on an element, the event first triggers on that specific element (the target). Then, it "bubbles" up, triggering the same event on all its parent elements in the DOM hierarchy, one by one.

### Example:

Consider the following HTML structure:

```html

<div class="outer">

<div class="middle">

<button class="inner">Click me</button>

</div>

</div>

```

### Event Bubbling in Action:

If you attach event listeners to each of these elements, they will be triggered in this order:

```javascript

document.querySelector('.inner').addEventListener('click', () => {

console.log('Inner clicked');

});

document.querySelector('.middle').addEventListener('click', () => {

console.log('Middle clicked');

});

document.querySelector('.outer').addEventListener('click', () => {

console.log('Outer clicked');

});

```

If you click the button (`.inner`), the console will log:

```

Inner clicked

Middle clicked

Outer clicked

```

### Stopping Event Bubbling:

You can prevent event bubbling using the `stopPropagation()` method. This stops the event from propagating to parent elements.

```javascript

document.querySelector('.inner').addEventListener('click', (event) => {

console.log('Inner clicked');

event.stopPropagation(); // Stops event from bubbling up

});

```

In this case, if you click on the inner button, only "Inner clicked" will be logged.

### Summary:

- \*\*Event Bubbling\*\* is when an event triggers on the target element and propagates up through its ancestors.

- It helps in managing events at multiple levels and allows for event delegation, where you attach a listener to a parent element and handle events for its child elements.

- You can stop the bubbling using `event.stopPropagation()`.

Package.json: For js based projects, this is the entry file. It will have the basic scripts

Npx – Node package eXecute.

In React, we don’t use HTML directly, but we use jsx ( java Script eXecutor ).   
JSX – Embed JS in HTML

JS: is a functional language

React doesn’t do deep rendering, it does shallow rendering for reference based objects. So the react will render( reload) only when the reference is changed. So if the reference is not changed and the data structure is edited, then react will not show the edited data structure.