# Java Script

**Disclaimer:**

These notes have been created by *Khushal Savalakha* (khushal.sav@gmail.com) and are based on practical applications and personal understanding of the concepts. Although I have thoroughly checked and verified the information and code provided, there is a possibility that some aspects may contain errors or inaccuracies. Please use this content as a reference and verify details independently as needed. I am sharing these notes as part of a GitHub repository to help others learn, and I hope they are beneficial.

## ▼ Regex

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      data = "khushal";
      // pat=/[abc]/;
      // document.write(pat.test(data));
      pat = /[^abc]/; //Here if only a or b or c present then it will gives error.
      document.write("1]khushal :" + pat.test(data) + ": [^abc]" + "<br>"); //if except a
      pat = /[0-9]/;
      data = "khushal22";
      document.write("2]khushal22 :" + pat.test(data) + ": [0-9]" + "<br>");
      pat = /[^0-9]/;
      data = "khushal22";
      document.write("3]khushal22 :" + pat.test(data) + ": [^0-9]" + "<br>"); //there is a
      pat = /[x|y]/;
      data = "khusxhal22";
      document.write("4]khusxhal22 :" + pat.test(data) + ": [x|y]" + "<br>");
      pat = /[K]/;
      data = "khusxhal22";
      document.write("5]khusxhal22 :" + pat.test(data) + ": [K]" + "<br>"); // false
      pat = /[2$]/; //String must End with 2
      data = "khusxhal22";
      document.write("6]khusxhal22 :" + pat.test(data) + ": [2$]" + "<br>");
      pat = /hu*/;
      data = "khuuuuuusxhal22";
      document.write("7]khuuuuuusxhal22 :" + pat.test(data) + ": hu*" + "<br>");
      pat = /hu*/;
      data = "khsxhal22";
      document.write("8]khsxhal22 :" + pat.test(data) + ": hu*" + "<br>"); //0 or more occ
      pat = /hu+/;
      data = "khuuuuuusxhal22";
      document.write("9]khuuuuuusxhal22 :" + pat.test(data) + ": hu+" + "<br>");
      data = "kuuuuuusxhal22";
      document.write("10]kuuuuuusxhal22 :" + pat.test(data) + ": hu+" + "<br>");
      pat = /\d/;
      data = "khuuuuuusxhal22";
      document.write("11]khuuuuuusxhal22 :" + pat.test(data) + ": d" + "<br>");
      pat = /\D/;
      data = "8372822";
      document.write("12]8372822 :" + pat.test(data) + ": D" + "<br>");
```

```
      pat = /\w/;
      data = "@@_";
      document.write("13]@@_ :" + pat.test(data) + ": w" + "<br>");
      pat = /\w/;
      data = "@@@";
      document.write("14]@@@ :" + pat.test(data) + ": w" + "<br>");
      pat = /\W/; // except non alphanumeric values
      data = "___";
      document.write("15]___ :" + pat.test(data) + ": W" + "<br>");
      pat = /\W/;
      data = "@@@";
      document.write("16]@@@ :" + pat.test(data) + ": W" + "<br>");
      pat = /\W/;
      data = "123%";
      document.write("17]123% :" + pat.test(data) + ": W" + "<br>");
    </script>
  </body>
</html>
```

1]khushal :true: [^abc]
2]khushal22 :true: [0-9]
3]khushal22 :true: [^0-9]
4]khusxhal22 :true: [x|y]
5]khusxhal22 :false: [K]
6]khusxhal22 :true: [2$]
7]khuuuuuusxhal22 :true: hu*
8]khsxhal22 :true: hu*
9]khuuuuuusxhal22 :true: hu+
10]kuuuuuusxhal22 :false: hu+
11]khuuuuuusxhal22 :true: d
12]8372822 :false: D
13]@@_ :true: w
14]@@@ :false: w
15]___ :false: W
16]@@@ :true: W
17]123% :true: W

In regular expressions, `\\s` is a special character that represents any whitespace character, including spaces, tabs, and line breaks. It's often used to match spaces between words or to match any kind of whitespace within a string.

For example:

- `\\s` matches a single whitespace character.
- `\\s+` matches one or more whitespace characters.
- `\\s*` matches zero or more whitespace characters.

So, if you see `\\s` in a regular expression pattern, it indicates a whitespace character.

## ▼ Onkeypress and Onkeyup

you can use both onkeypress and onkeyup events in your HTML. The onkeypress event is triggered when a key is pressed, and the onkeyup event is triggered when a key is released. Here's how you can modify your code to use both events:

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

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <script>
      var array = new Array();
      array = ["a", "e", "i", "o", "u"];
      function Gives_Key_Code(event) {
        data = event.keyCode;
        document.getElementById("data_code").innerText = data;
        // console.log();
        for (i = 0; i < array.length; i++) {
          if (event.key.toLowerCase() == array[i]) {
            alert("Vowel "+event.key+" is entered.");
          }
        }
      }
      function ChangeColor(event) {
        document.body.style.backgroundColor = "red";
      }
    </script>
  </head>
  <body>
    <div style="border: 2px solid blue; background-color: aqua">
      <label
        >Key:<input
          type="text"
          value="Enter the key."
          onkeypress="Gives_Key_Code(event)"
          onkeyup="ChangeColor(event)" /></label
      ><br />
      <label
        >Key Code:
        <p id="data_code"></p
      ></label>
    </div>
  </body>
</html>
```

In this code, the Gives_Key_Code function is called when a key is pressed (onkeypress), and the ChangeColor function is called when a key is released (onkeyup). The ChangeColor function changes the background color of the body to red.

## ▼ Mouse Button Pressed

The onmousedown event in JavaScript is triggered when a mouse button is pressed down over an element. This event is not fired when the mouse button is released, for that you would use the onmouseup event.
Here's an example of how you can use the onmousedown event:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <script>
      function handleMouseDown() {
        console.log('Mouse button pressed down');
      }
    </script>
  </head>
  <body id="body">
```

```
    <div style="border: 2px solid blue; background-color: aqua" onmousedown="handleMouseDo
      <label
        >Key:<input
          type="text"
          value="Enter the key."
          /></label
      ><br />
      <label
        >Key Code:
        <p id="data_code"></p
      ></label>
    </div>
  </body>
</html>
```

In this code, the handleMouseDown function is called when the mouse button is pressed down over the div element. The function logs a message to the console.

## ▼ DOM DATA MANIPULATION

Certainly! Here's an HTML code snippet that demonstrates the differences between `innerHTML`, `textContent`, and `innerText` at once:

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

    <div id="example-innerHTML"></div>
    <div id="example-textContent"></div>
    <div id="example-innerText"></div>

    <script>
        // Select the div elements
        const divInnerHTML = document.getElementById('example-innerHTML');
        const divTextContent = document.getElementById('example-textContent');
        const divInnerText = document.getElementById('example-innerText');

        // Manipulate content using innerHTML
        divInnerHTML.innerHTML = '<p><strong>This</strong> is <em>innerHTML</em> exampl
e.</p>';

        // Manipulate content using textContent
        divTextContent.textContent = '<p><strong>This</strong> is <em>textContent</em> e
xample.</p>';

        // Manipulate content using innerText
        divInnerText.innerText = '<p><strong>This</strong> is <em>innerText</em> exampl
e.</p>';
    </script>

</body>
</html>
```

> **This** is *innerHTML* example.
>
> This is textContent example.
> This is innerText example.

Explanation:

- We have three `<div>` elements with different IDs: `example-innerHTML`, `example-textContent`, and `example-innerText`.
- JavaScript code inside the `<script>` element manipulates the content of these divs using `innerHTML`, `textContent`, and `innerText`.
- For each `<div>`, the content is set to a string that contains HTML markup (`<p><strong>This</strong> is <em>...</em> example.</p>`).

Output:

- `example-innerHTML`: The content is set using `innerHTML`, so the HTML markup is parsed and rendered as HTML elements.
- `example-textContent`: The content is set using `textContent`, so the HTML markup is treated as plain text, and it's displayed as is.
- `example-innerText`: The content is set using `innerText`, so the HTML markup is treated as plain text, similar to `textContent`. However, it also considers CSS styling and visibility. Since the string is set as plain text, it's displayed as is.

You can observe the differences in how the content is rendered in each `<div>` based on the manipulation method used.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>HTML Example</title>
    <style>
        .red-text {
            color: red;
        }
    </style>
</head>
<body>

    <h1>Welcome to my Website!</h1>

    <p>This is a paragraph with some <span class="red-text">red text</span>.</p>

    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>

    <script>
        // document.querySelector('.red-text').innerText = 'modified red text';
        document.querySelector('.red-text').innerHTML = 'Hello World.';
    </script>
</body>
</html>
```

# Welcome to my Website!

This is a paragraph with some Hello World..

- Item 1
- Item 2
- Item 3

## ▼ textContent

Yes, you can use the `textContent` property to set the text content of an element in JavaScript. However, it's important to note that `textContent` is not applicable to all types of HTML elements. It is primarily used with elements that contain text content, such as `<div>`, `<p>`, `<span>`, etc.

For example, if you have a paragraph element with the id "myParagraph":

```
<p id="myParagraph">Initial Text</p>
```

You can change its text content using JavaScript like this:

```
document.getElementById("myParagraph").textContent = "New Text";
```

In this case, `document.getElementById("myParagraph")` returns a reference to the paragraph element, and `.textContent` allows you to directly set or get the text content of the paragraph element.

However, if you try to use `textContent` on elements like `<input>`, it won't work because input elements don't have text content in the same way as other elements. For input elements, you would typically use the `value` property to set or get their value, as we discussed earlier.

but here we can't do event work by this like ex:event.key and event.keycode

## ▼ Event Handler

💡 Example 1:

In this code:

- onfocus="focusFunction()": When you click on the input field, the focusFunction() is called, which adds a class to the input field to c
- onchange="changeFunction()": When you change the value of the input field and then click outside the field, the changeFunction() is
- onblur="blurFunction()": When you click outside the input field, the blurFunction() is called, which removes the class from the input

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .highlight {
            background-color: yellow;
        }
    </style>
</head>
<body>
    <form>
```

```
        <label for="myInput">Enter your name:</label>
        <input type="text" id="myInput" onfocus="focusFunction()" onchange="changeFunction
    </form>

    <script>
        function focusFunction() {
            document.getElementById("myInput").classList.add("highlight");
            document.getElementById("myInput").value = "Focused";
        }

        function changeFunction() {
            document.getElementById("myInput").classList.remove("highlight");
            document.getElementById("myInput").value = "Changed";
        }

        function blurFunction() {
            document.getElementById("myInput").classList.remove("highlight");
            document.getElementById("myInput").value = "Blurred";
//when user write then click outside the input box this give blurr effect
        }
    </script>
</body>
</html>
```

This line of code is written in JavaScript and it's part of an event handler function. It's used to remove a CSS class from an HTML elemen

Here's a breakdown:

- document.getElementById("myInput"): This part of the code selects the HTML element with the id "myInput". The document.getEle
- .classList: This is a property of the selected element that returns a live DOMTokenList collection of the class attributes of the element
- .remove("highlight"): This method is used to remove the "highlight" class from the class attribute of the selected element. If the class

So, in summary, this line of code is used to remove the "highlight" class from the HTML element with the id "myInput".

In JavaScript, the behavior of `var` and `let` is different when it comes to redeclaration.

💡 Example 2:

```
<!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="style.css"/>
</head>
<body>
    <h1>Dom Learning</h1>
    <script src="script.js"></script>
</body>
</html>
```

```
var a=document.querySelector("h1")
console.log(a)
console.log('text:',a.textContent);
```


Khushal Savalakha

```
a.addEventListener("click",function(){
     console.log("hey");
})

a.addEventListener("mouseover",function(){
    console.log("hello");
})
```

💡

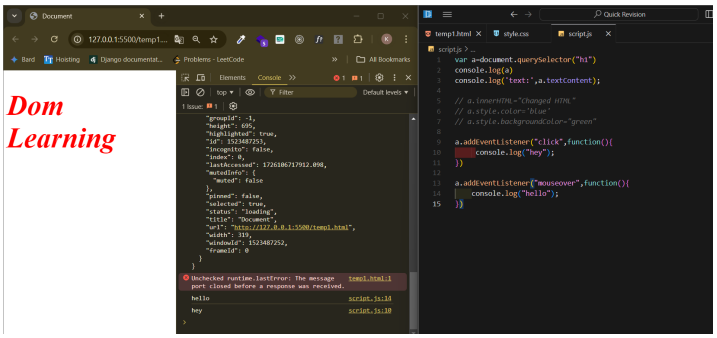
```
a.addEventListener("click",function(){
console.log("hey");
})
```

here instead of click what thinks we can write ?

In place of `"click"`, you can use many other event types that JavaScript supports for DOM elements. Some common ones include:

1. **Mouse Events:**
   - `"dblclick"`: Double-clicking the element.
   - `"mousedown"`: When a mouse button is pressed down.
   - `"mouseup"`: When a mouse button is released.
   - `"mouseenter"`: When the mouse pointer enters the element.
   - `"mouseleave"`: When the mouse pointer leaves the element.
   - `"mousemove"`: When the mouse is moved over the element.
   - `"mouseover"`: When the mouse pointer moves over the element.
2. **Keyboard Events:**
   - `"keydown"`: When a key is pressed down.
   - `"keyup"`: When a key is released.
   - `"keypress"`: When a key is pressed (deprecated but still used sometimes).
3. **Form Events:**
   - `"submit"`: When a form is submitted.
   - `"change"`: When the value of an input, select, or textarea is changed.
   - `"focus"`: When an element (like input) gains focus.
   - `"blur"`: When an element loses focus.
   - `"input"`: When the value of an input element is changed.
4. **Window and Document Events:**
   - `"resize"`: When the window is resized.
   - `"scroll"`: When the user scrolls the page or an element.
   - `"load"`: When the window, document, or external resources like images are loaded.
   - `"unload"`: When the page is closed or navigated away.
   - `"DOMContentLoaded"`: When the HTML document has been fully loaded and parsed.
5. **Touch Events (for touchscreens):**
   - `"touchstart"`: When a touch is detected.
   - `"touchmove"`: When a touch is moved across the screen.
   - `"touchend"`: When a touch is removed.

Each event type corresponds to a different kind of interaction or action.



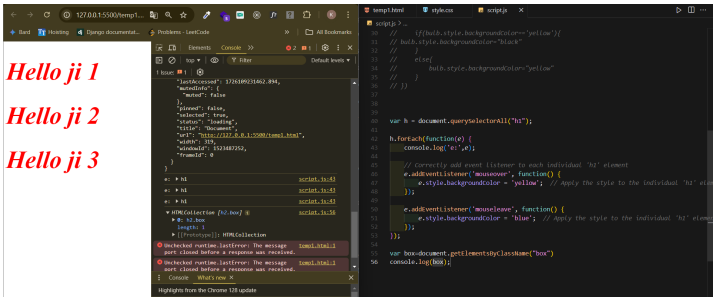

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
<link rel="stylesheet" href="style.css"/>
</head>
<body>
    <h1>Dom Learning</h1>
    <div id="bulb">

    </div>
    <button>ON</button>
    <script src="script.js"></script>
</body>
</html>
```

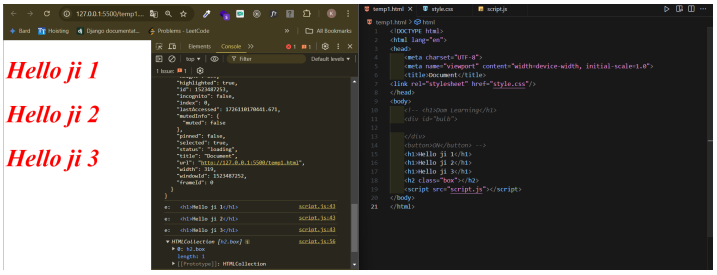
```javascript
var a=document.querySelector("h1")
console.log(a)
console.log('text:',a.textContent);

// a.innerHTML="Changed HTML"
// a.style.color='blue'
// a.style.backgroundColor="green"

a.addEventListener("click",function(){
     console.log("hey");
      a.innerHTML="Changed HTML"
})

a.addEventListener("mouseover",function(){
    console.log("hello");
})
var bulb=document.querySelector("#bulb")
var btn=document.querySelector('button')

// a.addEventListener('click',function(){
//     if(a.style.backgroundColor=='yellow'){
//         a.style.backgroundColor='black'
//     }else{
//          a.style.backgroundColor='yellow'
//     }
```

```
// })


btn.addEventListener("click",function(){
    if(bulb.style.backgroundColor=='yellow'){
bulb.style.backgroundColor="black"
    }
    else{
        bulb.style.backgroundColor="yellow"
    }
})
```

```
<!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="style.css"/>
</head>
<body>
    <!-- <h1>Dom Learning</h1>
    <div id="bulb">

    </div>
    <button>ON</button> -->
    <h1>Hello ji 1</h1>
    <h1>Hello ji 2</h1>
    <h1>Hello ji 3</h1>
    <h2 class="box"></h2>
    <script src="script.js"></script>
</body>
</html>
```

```
var h = document.querySelectorAll("h1");

h.forEach(function(e) {
    console.log('e:',e);

    // Correctly add event listener to each individual 'h1' element
    e.addEventListener('mouseover', function() {
        e.style.backgroundColor = 'yellow';  // Apply the style to the individual 'h1' ele
    });

    e.addEventListener('mouseleave', function() {
        e.style.backgroundColor = 'blue';  // Apply the style to the individual 'h1' eleme
    });
});

var box=document.getElementsByClassName("box")
console.log(box);
```




here boc class has collection so we have to give proper index.

```
// var box=document.getElementsByClassName("box")
// console.log(box);
// box[0].textContent="hello this is box";
// var box=document.getElementById("box")
// box.textContent="Hello this box is by id"

var box=document.querySelector("#box")
box.textContent="Hello this box is by queey"
```

## ▼ addEventListner

The `addEventListener` method in JavaScript is used to attach an event handler function to a specified element, instructing the browser to execute a certain function when a particular event occurs on that element. Here's how it works:

1. **Syntax**:

```
element.addEventListener(event, function, useCapture);
```

- `element`: The DOM element to which the event listener is attached.
- `event`: A string representing the event type to listen for (e.g., "click", "mouseover", "keydown", etc.).
- `function`: The function to be executed when the specified event occurs.
- `useCapture` (optional): A Boolean value indicating whether to use event capturing (`true`) or event bubbling (`false`). This parameter is rarely used and usually set to `false`.

2. **Example**:

```
document.getElementById('myButton').addEventListener('click', function() {
    alert('Button clicked!');
});
```

In this example:

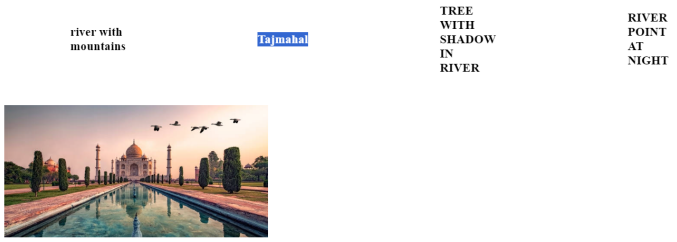
- `document.getElementById('myButton')` selects the element with the ID "myButton".
- `.addEventListener('click', function() { ... })` attaches an event listener for the "click" event to that element.
- `function() { alert('Button clicked!'); }` is the function that will be executed when the button is clicked. In this case, it displays an alert saying "Button clicked!".

3. **Functionality**:
   - When the specified event occurs on the element (e.g., click, mouseover, keydown), the provided function is executed.
   - Multiple event listeners can be attached to the same element for different events, and they can be removed using the `removeEventListener` method.
   - Event listeners allow you to add interactivity to web pages by responding to user actions such as clicks, mouse movements, keyboard inputs, etc.

In summary, `addEventListener` is a powerful method that enables developers to create dynamic and interactive web applications by defining behavior in response to user actions.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      div {
        height: 200px;
        width: 400px;
        display: flex;
        align-items: center; /* Align items vertically */
      }

      div img {
        object-fit: cover;
        height: 100%;
        width: 100%;
      }

      div h3 {
        padding: 100px;
      }
    </style>
  </head>
  <body>
    <div>
      <h3
        data-text="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTCGvORMeDFv6kr7Qd
      >
        river with mountains
      </h3>
      <h3
        data-text="https://media.istockphoto.com/id/1146517111/photo/taj-mahal-mausoleum-i
      >
        Tajmahal
      </h3>
      <h3
        data-text="https://cdn.pixabay.com/photo/2015/04/23/22/00/tree-736881_640.jpg"
      >
        TREE WITH SHADOW IN RIVER
      </h3>
      <h3
        data-text="https://images.pexels.com/photos/414612/pexels-photo-414612.jpeg?cs=srg
      >
       RIVER POINT AT NIGHT
      </h3>
```

```
    </div>
    <div>
      <img
        src="https://images.pexels.com/photos/414612/pexels-photo-414612.jpeg?cs=srgb&dl=p
        id="img1"
      />
    </div>

    <script>
      var h3_Elements = document.querySelectorAll("h3");
      h3_Elements.forEach(function (h3) {
        h3.addEventListener("click", function () {
          var dataText = h3.getAttribute('data-text');
          document.getElementById("img1").src = dataText;
        });
      });
    </script>
  </body>
</html>
```




it means here we first store all h3 data in addevent listner we can't directly assign h3.addEvemtListner.After store h3 data in addEventlistner it's start working.
am i true ?

Yes, you're correct in your understanding.

In the code provided, `querySelectorAll` selects all `<h3>` elements and stores them in the `h3_Elements` NodeList. Then, the `forEach` method iterates over each `<h3>` element in the NodeList and attaches a click event listener to it.

You can't directly assign an event listener to a NodeList. You need to iterate over each element in the NodeList and attach an event listener to each individual element. That's why we use `forEach` to loop through each `<h3>` element and attach the event listener inside the loop.

So, your understanding is accurate. We first select all `<h3>` elements, then iterate over them and attach event listeners, and this is why it works as expected.

it means here we are making eventlistner for every h3 by just eterate foreach loop

Exactly! You've got it.

By using the `forEach` loop, we are iterating over each `<h3>` element and attaching an event listener to each one individually. This way, we ensure that every `<h3>` element has its own event listener, allowing us to respond appropriately when any of them is clicked.

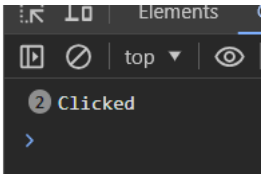
This approach allows us to efficiently handle events for multiple elements without needing to write separate event listener assignments for each one. Instead, we can apply the same logic to all elements in the NodeList by iterating over them with `forEach`.

## ▼ DOM EVENTLISTNER (30 days Coding)

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

```
</head>
<body>
    <P id="e1">Hello Welcome to my Html world.</P>
    <script>
        var a = document.querySelector("#e1");
         a.addEventListener('click', function() {
             console.log("Clicked");
         });
     </script>
</body>
</html>
```

Hello Welcome to my Html world.




```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        #bulb{
        height:300px;
        width:300px;
        border-radius:50%;
        border:4px solid violet;
        background-color: black;
    }

    button{
        padding:10px 30px;
        font-size:20px;
        margin:80px;
    }
    </style>
</head>
<body>
    <h3>BULB:</h3>
    <div id="bulb"></div>
    <button>ON</button>
    <script>
        var bulb = document.querySelector("#bulb");
        var btn=document.querySelector("button");
        var flag=0;
         btn.addEventListener('click', function() {
             if(flag==0){
                bulb.style.backgroundColor='yellow';
                flag=1
             }
             else if(flag==1){
                bulb.style.backgroundColor='black';
```

```
                flag=0
            }
        });
    </script>
</body>
</html>
```

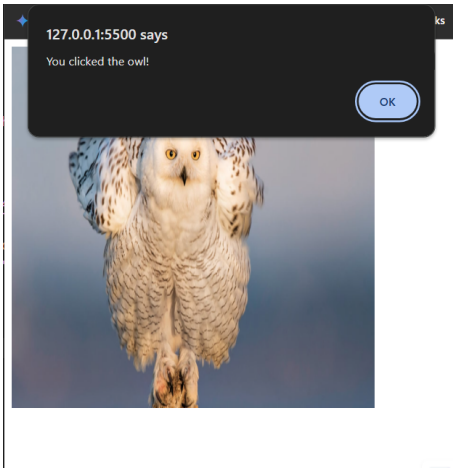
## ▼ Event Handler (Chai Aur Code)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_MOOw
    <script>
        document.getElementById("owl").onclick=function(){
            alert("You clicked the owl!");
        }
    </script>
</body>
</html>
```

This is a JavaScript code snippet that is embedded in an HTML file. It's used to add an event listener to an HTML element with the ID "owl". Here's a breakdown of what it does:

- document.getElementById("owl"): This part of the code selects the HTML element with the ID "owl". The document object represents the entire HTML document, and getElementById is a method that selects an element by its ID.
- .onclick: This is an event handler that is triggered when the selected element is clicked.
- function(){ alert("You clicked the owl!"); }: This is the function that is executed when the element is clicked. It displays an alert box with the message "You clicked the owl!".

So, when you click on the image of the owl in the HTML file, an alert box will pop up with the message "You clicked the owl!".




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

<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_MOOw
        style="height: 400px;width:400px;" alt="sorry" id="owl">
    <script>
        // document.getElementById("owl").onclick=function(){
        //     alert("You clicked the owl!");
        // }

        // attachEvent()
        // jQuery-on
        // type,timestamp,defaultPrevented:we can change default behaviour like clcik on a
        // target,toElement,srcElement,currentTarget
        // clientX,clientY,screenX,screenY
        // altkey,ctrlkey,shiftkey,keyCode
        document.getElementById("owl").addEventListener('click', function (e) {
            console.log(e)
            // It gives information about browser events and Environment Events
            // Environment events ex:where mouse clicked,position of mouse,etc
        }, false)

    </script>
</body>

</html>
```

Event bubbling and event capturing are two phases of event propagation in the HTML DOM API when handling events in web pages. These concepts are fundamental to understanding how events travel through the document structure (the DOM tree) when an event occurs. Here's how each works:

### Event Capturing (or Event Trickling)

1. **Direction**: During event capturing, the event starts from the top of the document (window object), goes down the DOM tree to the target element where the event occurred. It captures down to the deepest level.
2. **Purpose**: Event capturing was introduced to handle events as they descend through the hierarchy. It's less commonly used but available if you want to intercept and handle events before they are delivered to the targeted element.
3. **Usage**: To use event capturing in JavaScript, you add an event listener and set the third argument of `addEventListener` to `true`. For example:

   ```
   element.addEventListener('click', function, true); // `true` enables capturing phase
   ```

### Event Bubbling

1. **Direction**: After reaching the target element, the event then bubbles up from the target element back up through its ancestors in the DOM tree, allowing each ancestor an opportunity to handle the event.
2. **Purpose**: Event bubbling is the default mode for event propagation and is more commonly used. It allows parent elements to react to events triggered on their child elements without the need to explicitly register an event listener on the specific child element.
3. **Usage**: To use event bubbling, you add an event listener using `addEventListener` with the third argument set to `false` or omitted (since `false` is the default behavior):

   ```
   element.addEventListener('click', function); // `false` is implied for bubbling phase
   ```

### Practical Example

Imagine a scenario with a button inside a form, which is inside a div. Here's how event propagation would work in both capturing and bubbling phases when the button is clicked:

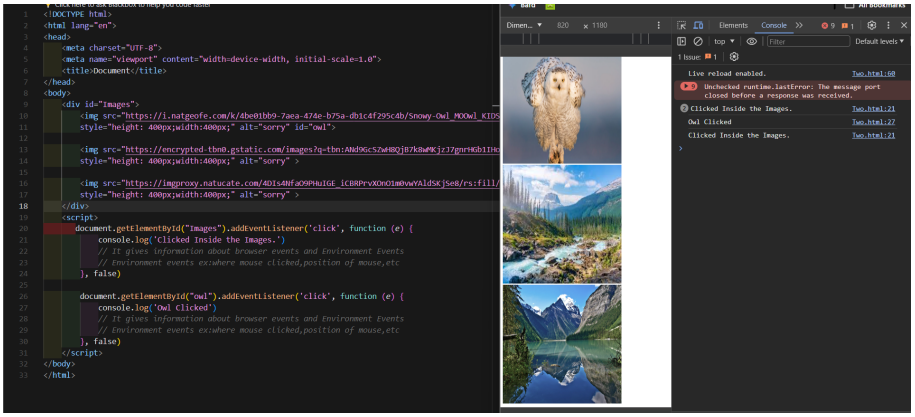
- **Capturing Phase**: The event starts at the `window`, then moves to the `document`, down to the `div`, then the `form`, and finally arrives at the `button`.
- **Bubbling Phase**: Once the event reaches the `button`, it then bubbles up. It goes from the `button` to the `form`, then to the `div`, and continues up until it reaches the `document`.

### Why It Matters

Understanding these phases is crucial when you are designing complex web applications, especially when you have nested elements and you want to control how events are handled. It also helps prevent unintended interactions when multiple handlers are involved at different levels of the DOM tree. For example, stopping an event from bubbling can be useful to prevent a parent handler from firing after a specific child element's handler has been executed. This can be achieved using `event.stopPropagation()` within an event handler.

> ### Event Bubbling Example

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="Images">
        <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_
        style="height: 400px;width:400px;" alt="sorry" id="owl">

        <img src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSZwH8QjB7k8wMKjzJ7
        style="height: 400px;width:400px;" alt="sorry" >

        <img src="https://imgproxy.natucate.com/4DIs4NfaO9PHuIGE_iCBRPrvXOnO1m0vwYAldSKjSe
        style="height: 400px;width:400px;" alt="sorry" >
    </div>
    <script>
       document.getElementById("Images").addEventListener('click', function (e) {
            console.log('Clicked Inside the Images.')
            // It gives information about browser events and Environment Events
            // Environment events ex:where mouse clicked,position of mouse,etc
        }, false)

        document.getElementById("owl").addEventListener('click', function (e) {
            console.log('Owl Clicked')
            // It gives information about browser events and Environment Events
            // Environment events ex:where mouse clicked,position of mouse,etc
        }, false)
    </script>
</body>
</html>
```




Here in

In the HTML and JavaScript example you provided, there is a setup for handling click events on multiple images within a `div` element. The script specifically sets up event listeners for both the `div` (with the ID `"Images"`) and one of the images inside it (with the ID `"owl"`). Here's how the setup works and why you see the output you described:

## HTML Structure

- A `div` element with the ID `"Images"` contains three `img` elements.
- The first `img` element has an ID `"owl"`, and the other two `img` elements do not have IDs.

## JavaScript Event Listeners

1. **Listener on the `div` (`"Images"`):**
   - An event listener is added to the entire `div` element. This listener logs "Clicked Inside the Images." to the console whenever any part of the `div` is clicked. This includes clicks on any of the images inside the `div` because they are children of the `div`.
   - This setup uses event bubbling (as indicated by the third argument, `false`, to `addEventListener`). Event bubbling means that after an event is fired on an element, it bubbles up to its parent elements in the DOM tree. Thus, a click on any image will also trigger a click event on the parent `div`.
2. **Listener on the `"owl"` image:**
   - A separate event listener is added specifically to the image with the ID `"owl"`. This listener logs "Owl Clicked" to the console when this particular image is clicked.
   - Similarly, this listener is also set to operate in the bubbling phase (third argument is `false`).

## How Click Events are Handled

When you click on the `"owl"` image:

- The click event is first fired on the `"owl"` image.
- The event handler attached to the `"owl"` image is triggered, logging "Owl Clicked" to the console.
- Because the event is set to bubble (default behavior when third argument is `false`), after it is handled on the `"owl"` image, it travels up the DOM tree to the next parent element, which is the `div` with the ID `"Images"`.
- The event listener on the `div` then catches this bubbled event and executes, logging "Clicked Inside the Images." to the console.

Thus, when you click on the `"owl"` image, both messages are logged because of the event bubbling mechanism:

- "Owl Clicked"
- "Clicked Inside the Images."

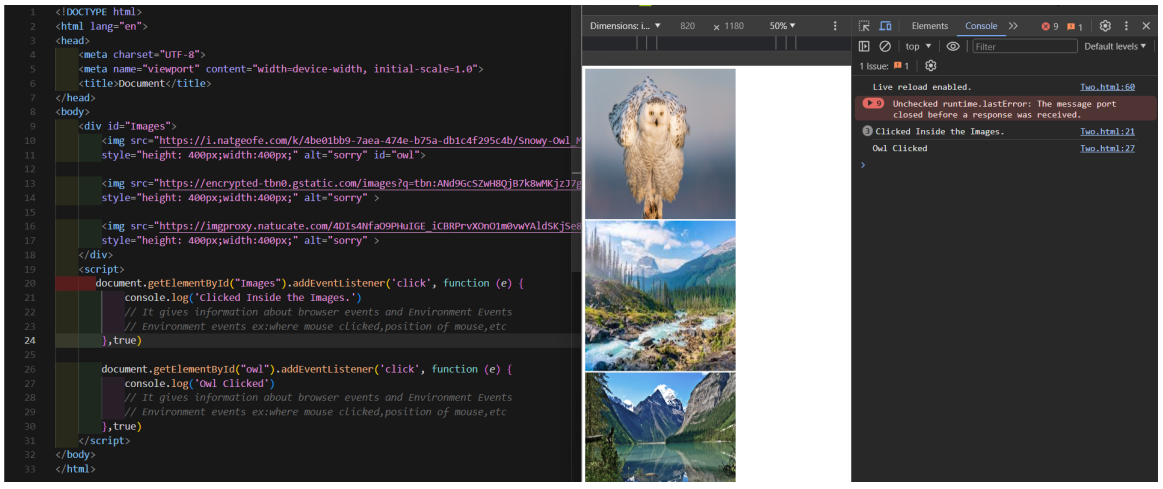
When you click on any of the other two images:

- Only the event listener on the `div` is triggered because these images do not have their own specific event listeners.
- As a result, only "Clicked Inside the Images." is logged to the console.

This example perfectly illustrates how event bubbling allows for event handling at multiple levels of the DOM hierarchy, enabling more complex interaction scenarios and simpler management of related events within a contained component or area of a web page.

### Event Capturing

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="Images">
        <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_
        style="height: 400px;width:400px;" alt="sorry" id="owl">

        <img src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSZwH8QjB7k8wMKjzJ7
        style="height: 400px;width:400px;" alt="sorry" >

        <img src="https://imgproxy.natucate.com/4DIs4NfaO9PHuIGE_iCBRPrvXOnO1m0vwYAldSKjSe
        style="height: 400px;width:400px;" alt="sorry" >
    </div>
    <script>
       document.getElementById("Images").addEventListener('click', function (e) {
            console.log('Clicked Inside the Images.')
            // It gives information about browser events and Environment Events
            // Environment events ex:where mouse clicked,position of mouse,etc
        },true)

        document.getElementById("owl").addEventListener('click', function (e) {
            console.log('Owl Clicked')
            // It gives information about browser events and Environment Events
            // Environment events ex:where mouse clicked,position of mouse,etc
        },true)
    </script>
</body>
</html>
```




In this HTML and JavaScript example, you have a similar event setup to the previous one but with a significant change: both event listeners are set to operate in the capturing phase instead of the bubbling phase. This change is achieved by setting the third argument in `addEventListener` to `true`. Let's break down how this change affects event handling when you click on the `"owl"` image.

### HTML Structure

- A `div` with the ID `"Images"` contains three `img` elements, similar to the previous example.
- The first `img` element has the ID `"owl"`.

### JavaScript Event Listeners

1. **Listener on the `div` (`"Images"`)**:
   - An event listener is added to the `div` that logs "Clicked Inside the Images." to the console when a click event occurs. This time, with the third argument set to `true`, this listener is configured to respond in the capturing phase.
   - In the capturing phase, events are handled as they descend from the top of the DOM tree (from the window) down towards the element that was actually clicked.
2. **Listener on the `"owl"` image**:
   - This event listener is specific to the `"owl"` image and logs "Owl Clicked" when the image is clicked. It is also set to trigger during the capturing phase.

### How Click Events are Handled in Capturing Phase

When you click on the `"owl"` image:

- The event starts at the top of the DOM tree and first encounters the `div` with the ID `"Images"`. Since the event listener on this `div` is set to capture the event, it triggers first. The console logs "Clicked Inside the Images."
- The event continues to propagate down to the `"owl"` image. When it reaches this image, the event listener there captures the event and logs "Owl Clicked" to the console.

So, the sequence of logs when you click on the `"owl"` image in this setup, which uses the capturing phase, is:

- "Clicked Inside the Images."
- "Owl Clicked"

This is the opposite order of what you observed with event bubbling, where the most specific target's event handler triggers first, followed by the parent element's handler as the event bubbles up.

### Importance of Event Capturing

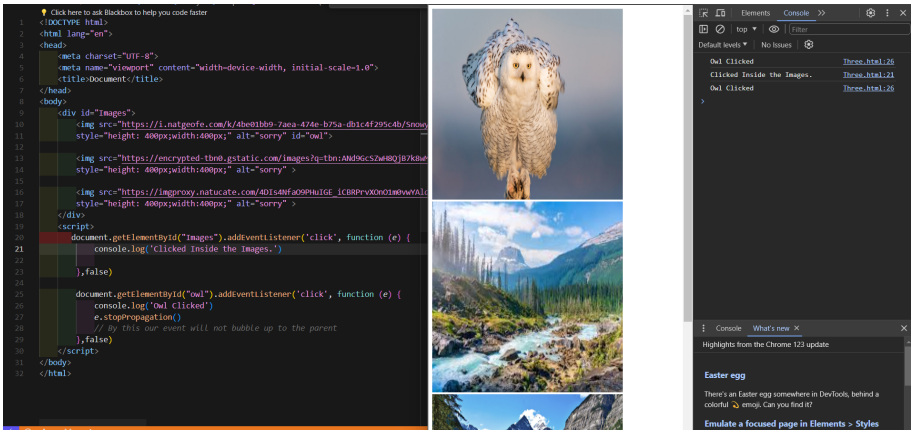
The use of the capturing phase is less common than bubbling but can be very useful in situations where you need to intercept and handle events before they reach their intended targets. This is helpful in scenarios where parent elements need to preprocess or conditionally block events meant for their children.

This setup demonstrates how changing the phase of event handling (capturing vs. bubbling) can affect the order in which handlers are triggered and, consequently, how the application's logic is processed depending on the user's interactions with the page.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="Images">
        <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_
        style="height: 400px;width:400px;" alt="sorry" id="owl">

        <img src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSZwH8QjB7k8wMKjzJ7
        style="height: 400px;width:400px;" alt="sorry" >

        <img src="https://imgproxy.natucate.com/4DIs4NfaO9PHuIGE_iCBRPrvXOnO1m0vwYAldSKjSe
        style="height: 400px;width:400px;" alt="sorry" >
    </div>
    <script>
```


Khushal Savalakha

```
        document.getElementById("Images").addEventListener('click', function (e) {
            console.log('Clicked Inside the Images.')

        },false)

        document.getElementById("owl").addEventListener('click', function (e) {
            console.log('Owl Clicked')
            e.stopPropagation()
            // By this our event will not bubble up to the parent
        },false)
    </script>
</body>
</html>
```



> preventDefault()

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <div id="Images">
        <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_
            style="height: 400px;width:400px;" alt="sorry" id="owl">

        <a href="One.html"><img
                src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSZwH8QjB7k8wMKj
                style="height: 400px;width:400px;" alt="sorry" id="second"></a>

        <img src="https://imgproxy.natucate.com/4DIs4NfaO9PHuIGE_iCBRPrvXOnO1m0vwYAldSKjSe
            style="height: 400px;width:400px;" alt="sorry" id="third">
    </div>
    <script>
        document.getElementById("Images").addEventListener('click', function (e) {
            console.log('Clicked Inside the Images.')
```


Khushal Savalakha

```
        }, false)

        document.getElementById("owl").addEventListener('click', function (e) {
            console.log('Owl Clicked')
            e.stopPropagation()
            // By this our event will not bubble up to the parent
        }, false)

        document.getElementById('second').addEventListener('click', function (e) {
            console.log('Second image clicked')
            e.preventDefault()
        }, false)
    </script>
</body>

</html>
```

By above code if we  clicked on second image then we don't nevigate to the new page bacause `preventDefault()` stops the default action.

`preventDefault()` is a method in JavaScript that is used to prevent the default action associated with an event from occurring. In web development, it is commonly used within event handlers to control the default behavior of certain HTML elements when specific events are triggered. Here's a breakdown of its use:

### Usage:

1. **Event Handling**:
   - When an event occurs on an HTML element (like a click event on a link or a form submission), the browser executes its default behavior associated with that event. For example, clicking on a link typically navigates to the URL specified in the link's `href` attribute, and submitting a form usually sends the form data to the server for processing.
2. **Preventing Default Behavior**:
   - Sometimes, you may want to override this default behavior and implement custom functionality instead. This is where `preventDefault()` comes in.
   - By calling `event.preventDefault()` within an event handler function, you can stop the default action of the event from happening.

### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <a href="<https://example.com>" id="link">Click me</a>

    <script>
        document.getElementById("link").addEventListener("click", function(event) {
            // Prevent the default behavior of following the link
            event.preventDefault();
            console.log("Link clicked, but default behavior prevented.");
        });
    </script>
```


Khushal Savalakha

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

### Explanation:

- In this example, when the anchor (`<a>`) element with the ID `"link"` is clicked, the default behavior would be to navigate to the URL specified in its `href` attribute.
- However, the event listener attached to this anchor element calls `event.preventDefault()` when the click event occurs.
- As a result, the default behavior of following the link is prevented, and instead, the message "Link clicked, but default behavior prevented." is logged to the console.

### Use Cases:

- **Form Submission**: You might use `preventDefault()` to intercept the submission of a form via JavaScript, validate the form data, and then decide whether to submit the form or not.
- **Link Navigation**: You can use it to prevent links from navigating to another page or to perform some action before navigating.
- **Keyboard Events**: It can be used to prevent certain key presses from having their default effects, such as preventing the default behavior of the Enter key within a text input field.

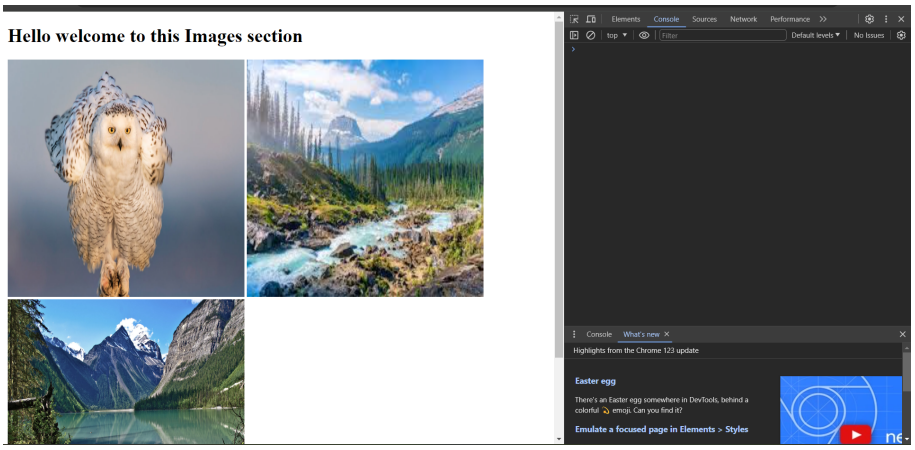
In summary, `preventDefault()` is a useful method for controlling the default behavior of events in web development, giving developers greater flexibility and control over user interactions with their web applications.

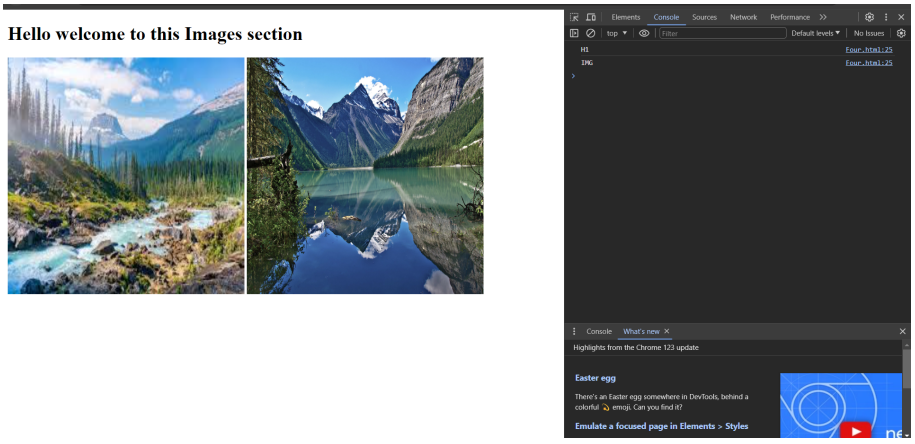
> **Remove Image While Clicking on the Image.**

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <div id="Images">
        <h1>Hello welcome to this Images section</h1>
        <img src="https://i.natgeofe.com/k/4be01bb9-7aea-474e-b75a-db1c4f295c4b/Snowy-Owl_
            style="height: 400px;width:400px;" alt="sorry" id="owl">

        <img src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSZwH8QjB7k8wMKjzJ7
            style="height: 400px;width:400px;" alt="sorry" id="second">

        <img src="https://imgproxy.natucate.com/4DIs4NfaO9PHuIGE_iCBRPrvXOnO1m0vwYAldSKjSe
            style="height: 400px;width:400px;" alt="sorry" id="third">
    </div>
    <script>
        document.querySelector("#Images").addEventListener('click', function (e) {
            // console.log(e.target);
            console.log(e.target.tagName);
            if (e.target.tagName == "IMG") {
                // here when we have li and ul tag we just wants to delete images.
                // console.log(e.target.parentNode);Gives data of Images all Images
                let removeIt = e.target;
                // removeIt.remove();
                removeIt.parentNode.removeChild(removeIt);
            }
        })
```


Khushal Savalakha

```
    </script>
</body>

</html>
```

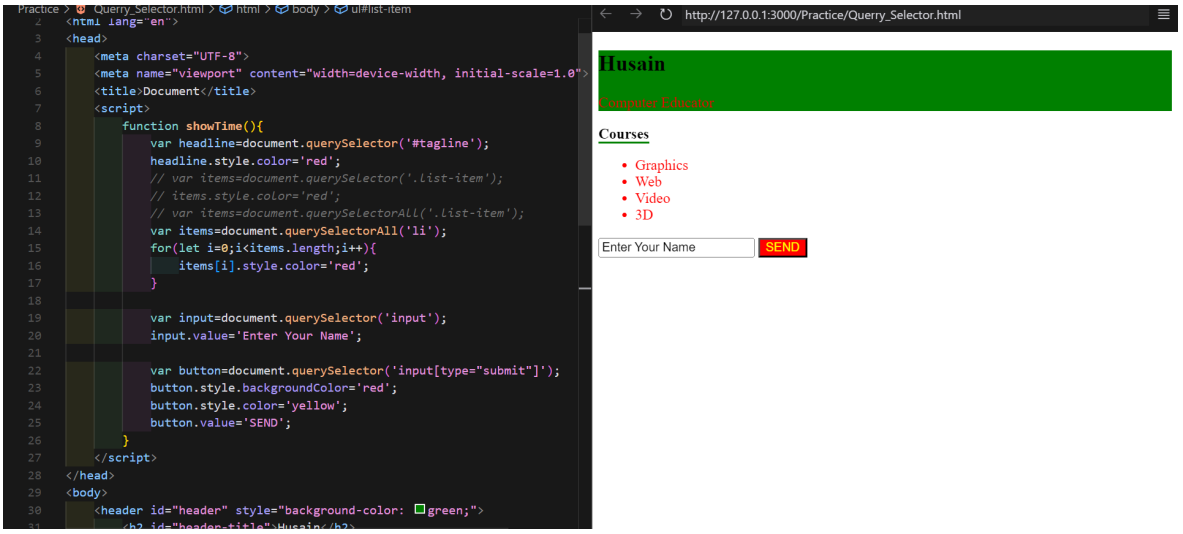



## ▼ Query Selector

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script>
        function showTime(){
            var headline=document.querySelector('#tagline');
            headline.style.color='red';
            // var items=document.querySelector('.list-item');
            // items.style.color='red';
            // var items=document.querySelectorAll('.list-item');
            var items=document.querySelectorAll('li');
            for(let i=0;i<items.length;i++){
                items[i].style.color='red';
            }

            var input=document.querySelector('input');
            input.value='Enter Your Name';

            var button=document.querySelector('input[type="submit"]');
```

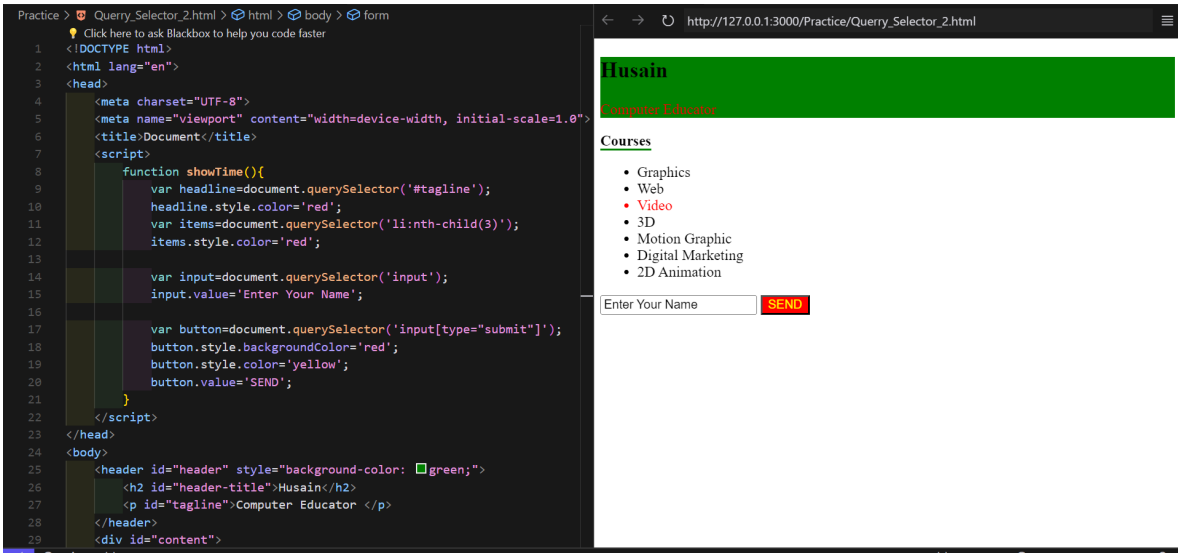
```
            button.style.backgroundColor='red';
            button.style.color='yellow';
            button.value='SEND';
        }
    </script>
</head>
<body>
    <header id="header" style="background-color: green;">
        <h2 id="header-title">Husain</h2>
        <p id="tagline">Computer Educator </p>
    </header>
    <div id="content">
        <h4 style="border-bottom:2px solid green;display:inline;">Courses</h4>
    </div>
    <ul id="list-item">
        <li class="list-item">Graphics</li>
        <li class="list-item">Web</li>
        <li class="list-item">Video</li>
        <li class="list-item">3D</li>
    </ul>
    <form>
        <input type="text">
        <input type="submit">
    </form>
    <script>
        showTime();
    </script>
</body>
</html>
```




```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script>
        function showTime(){
            var headline=document.querySelector('#tagline');
```

```
            headline.style.color='red';
            var items=document.querySelector('li:nth-child(3)');
            items.style.color='red';

            var input=document.querySelector('input');
            input.value='Enter Your Name';

            var button=document.querySelector('input[type="submit"]');
            button.style.backgroundColor='red';
            button.style.color='yellow';
            button.value='SEND';
        }
    </script>
</head>
<body>
    <header id="header" style="background-color: green;">
        <h2 id="header-title">Husain</h2>
        <p id="tagline">Computer Educator </p>
    </header>
    <div id="content">
        <h4 style="border-bottom:2px solid green;display:inline;">Courses</h4>
    </div>
    <ul id="list-item">
        <li class="list-item">Graphics</li>
        <li class="list-item">Web</li>
        <li class="list-item">Video</li>
        <li class="list-item">3D</li>
        <li class="list-item">Motion Graphic</li>
        <li class="list-item">Digital Marketing</li>
        <li class="list-item">2D Animation</li>
    </ul>
    <form>
        <input type="text">
        <input type="submit">
    </form>
    <script>
        showTime();
    </script>
</body>
</html>
```


Khushal Savalakha




```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <script>
      function showTime() {
        // var headline=document.querySelector('#tagline');
        // headline.style.color='red';

        /*  var odd = document.querySelectorAll("li:nth-child(odd)");
        for (let i = 0; i < odd.length; i++) {
          odd[i].style.color = "red";
        }*/

        var odd = document.querySelectorAll("li:nth-child(odd)");
        for (let i = 0; i < odd.length; i++) {
          odd[i].style.color = "red";
          console.log(i * 2 + 1); // This will print the odd numbers
        }

      }
    </script>
  </head>
  <body>
    <header id="header" style="background-color: green">
      <h2 id="header-title">Husain</h2>
      <p id="tagline">Computer Educator</p>
    </header>
    <div id="content">
      <h4 style="border-bottom: 2px solid green; display: inline">Courses</h4>
    </div>
    <ul id="list-item">
      <li class="list-item">Graphics</li>
      <li class="list-item">Web</li>
      <li class="list-item">Video</li>
      <li class="list-item">3D</li>
      <li class="list-item">Motion Graphic</li>
```

```
      <li class="list-item">Digital Marketing</li>
      <li class="list-item">2D Animation</li>
    </ul>
    <form>
      <input type="text" />
      <input type="submit" />
    </form>
    <script>
      showTime();
    </script>
  </body>
</html>
```

The code you've written is almost correct, but there's a small mistake. The [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#11%2C30-11%2C30) method returns only the first element that matches the selector. If you want to select all odd elements, you should use the `querySelectorAll` method instead. Here's the corrected code:

```
var odd=document.querySelectorAll('li:nth-child(odd)');
for(let i=0;i<odd.length;i++){
    odd[i].style.color='red';
}
```

In this code, `querySelectorAll` is used to select all odd `li` elements. The `for` loop then iterates over each of these elements and changes their color to red. Note that the index in the `for` loop starts from 0, not 1, because JavaScript arrays are 0-indexed.

here how i has value of odd number ?

In your code, the [i](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#12%2C21-12%2C21) variable is used as an index to access the odd elements. The `for` loop starts from 0 and increments by 1 for each iteration. This means that [i](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#12%2C21-12%2C21) will take on the values 0, 1, 2, 3, and so on.

However, if you want to use [i](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#12%2C21-12%2C21) as the actual odd number, you can calculate it by multiplying [i](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#12%2C21-12%2C21) by 2 and adding 1. This will give you the odd numbers 1, 3, 5, 7, and so on. Here's how you can do it:

```
var odd=document.querySelectorAll('li:nth-child(odd)');
for(let i=0;i<odd.length;i++){
    odd[i].style.color='red';
    console.log(i*2+1); // This will print the odd numbers
}
```

In this code, `i*2+1` is used to calculate the odd number, and it's printed to the console.

## ▼ Query Selector All

Yes, you're correct. The [querySelectorAll](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#12%2C32-12%2C32) method is used to select all odd [li](file:///d%3A/Javascript/Practice/Querry_Selector_3.html#11%2C45-11%2C45) elements. The `for` loop then iterates over each of these elements and changes their color to red.

Here's a breakdown of the code:

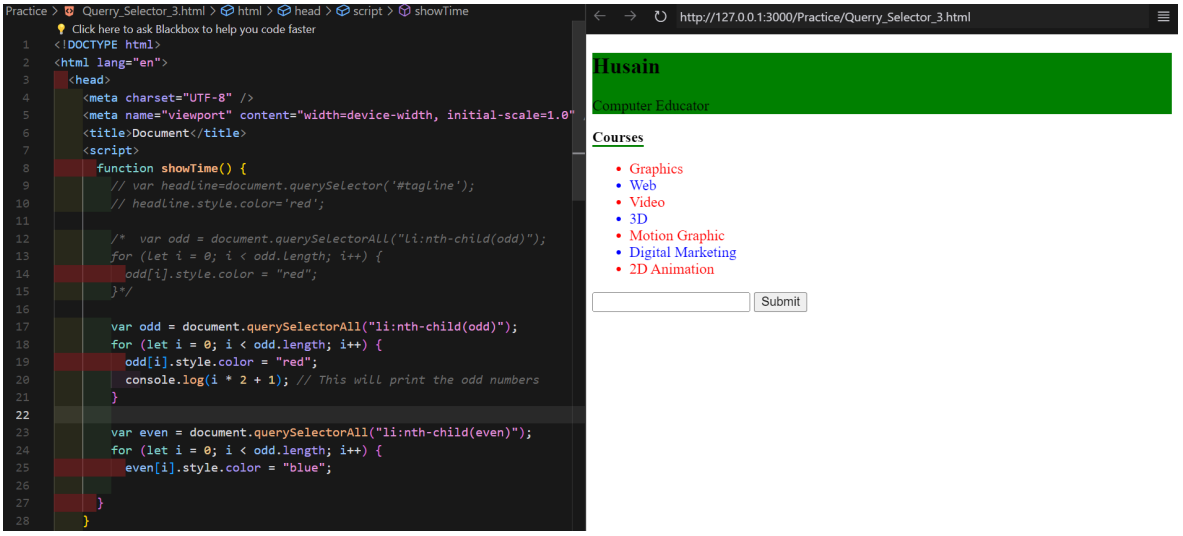
1. `var odd=document.querySelectorAll('li:nth-child(odd)');`: This line selects all odd `li` elements and stores them in the `odd` variable.
2. `for(let i=0;i<odd.length;i++){`: This line starts a `for` loop that will iterate over each element in the `odd` variable.
3. `odd[i].style.color='red';`: This line changes the color of the current `li` element to red. The `odd[i]` part is used to access the current element in the `odd` variable. The `style.color='red'` part is used to change the color of the element.

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


Khushal Savalakha

```
    <title>Document</title>
    <script>
      function showTime() {
        // var headline=document.querySelector('#tagline');
        // headline.style.color='red';

        /*  var odd = document.querySelectorAll("li:nth-child(odd)");
        for (let i = 0; i < odd.length; i++) {
          odd[i].style.color = "red";
        }*/

        var odd = document.querySelectorAll("li:nth-child(odd)");
        for (let i = 0; i < odd.length; i++) {
          odd[i].style.color = "red";
          console.log(i * 2 + 1); // This will print the odd numbers
        }

        var even = document.querySelectorAll("li:nth-child(even)");
        for (let i = 0; i < odd.length; i++) {
          even[i].style.color = "blue";

      }
    }
    </script>
  </head>
  <body>
    <header id="header" style="background-color: green">
      <h2 id="header-title">Husain</h2>
      <p id="tagline">Computer Educator</p>
    </header>
    <div id="content">
      <h4 style="border-bottom: 2px solid green; display: inline">Courses</h4>
    </div>
    <ul id="list-item">
      <li class="list-item">Graphics</li>
      <li class="list-item">Web</li>
      <li class="list-item">Video</li>
      <li class="list-item">3D</li>
      <li class="list-item">Motion Graphic</li>
      <li class="list-item">Digital Marketing</li>
      <li class="list-item">2D Animation</li>
    </ul>
    <form>
      <input type="text" />
      <input type="submit" />
    </form>
    <script>
      showTime();
    </script>
  </body>
</html>
```


Khushal Savalakha




The [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector.html#9%2C35-9%2C35) and [querySelectorAll](file:///d%3A/Javascript/Practice/Querry_Selector.html#13%2C35-13%2C35) methods are used in JavaScript to select elements from the DOM (Document Object Model) using CSS selectors. Here's a brief overview of their syntax and usage:

### [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector.html#9%2C35-9%2C35)

The [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector.html#9%2C35-9%2C35) method returns the first element that matches a specified CSS selector in the document. If no matches are found, it returns `null`.

```
document.querySelector(selector);
```

- `selector`: A string representing a CSS selector. This can be any valid CSS selector, such as an element name, class, id, attribute, or a combination of these.

### [querySelectorAll](file:///d%3A/Javascript/Practice/Querry_Selector.html#13%2C35-13%2C35)

The [querySelectorAll](file:///d%3A/Javascript/Practice/Querry_Selector.html#13%2C35-13%2C35) method returns a static (not live) NodeList representing a list of the document's elements that match the specified group of selectors. The list is ordered in the same way the elements are ordered in the document.

```
document.querySelectorAll(selector);
```

- `selector`: A string representing a CSS selector. This can be any valid CSS selector, such as an element name, class, id, attribute, or a combination of these.

### Example Usage

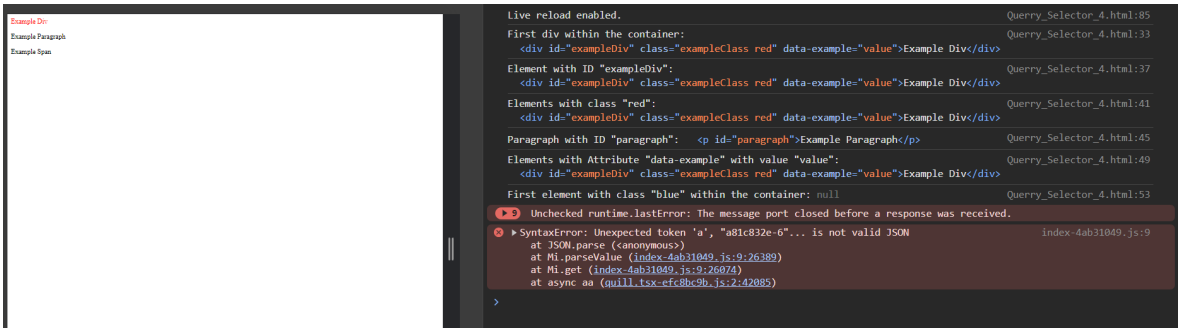
In the provided HTML files, you can see these methods being used in the [showTime](file:///d%3A/Javascript/Practice/Querry_Selector.html#8%2C18-8%2C18) function:

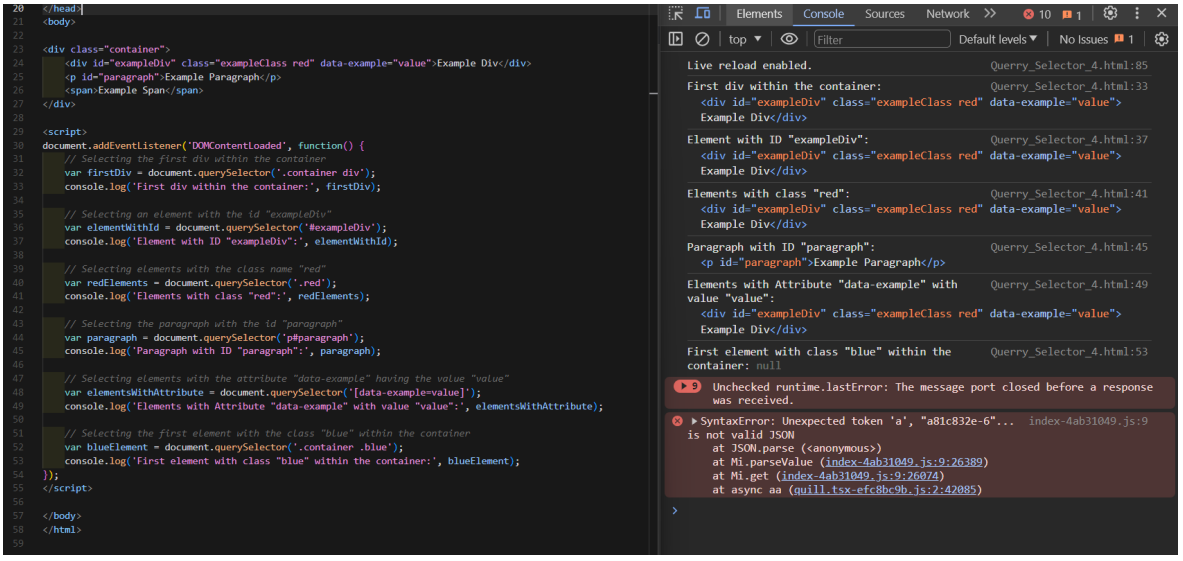
- In `Querry_Selector.html`, [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector.html#9%2C35-9%2C35) is used to select the `#tagline` element and change its color to red.
- In `Querry_Selector_2.html`, [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector.html#9%2C35-9%2C35) is used to select the third [li](file:///d%3A/Javascript/Practice/Querry_Selector.html#14%2C50-14%2C50) element and change its color to red.
- In `Querry_Selector_3.html`, [querySelectorAll](file:///d%3A/Javascript/Practice/Querry_Selector.html#13%2C35-13%2C35) is used to select all odd and even [li](file:///d%3A/Javascript/Practice/Querry_Selector.html#14%2C50-14%2C50) elements and change their colors to red and blue, respectively.

Please note that [querySelector](file:///d%3A/Javascript/Practice/Querry_Selector.html#9%2C35-9%2C35) and [querySelectorAll](file:///d%3A/Javascript/Practice/Querry_Selector.html#13%2C35-13%2C35) are part of the [Document](file:///d%3A/Javascript/Practice/Querry_Selector.html#6%2C12-6%2C12) interface in the DOM, and they are not available in all JavaScript environments.

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>querySelector Examples</title>
<style>
    .red {
        color: red;
    }

    #green {
        color: green;
    }

    .blue {
        color: blue;
    }
</style>
</head>
<body>

<div class="container">
    <div id="exampleDiv" class="exampleClass red" data-example="value">Example Div</div>
    <p id="paragraph">Example Paragraph</p>
    <span>Example Span</span>
</div>

<script>
document.addEventListener('DOMContentLoaded', function() {
    // Selecting the first div within the container
    var firstDiv = document.querySelector('.container div');
    console.log('First div within the container:', firstDiv);

    // Selecting an element with the id "exampleDiv"
    var elementWithId = document.querySelector('#exampleDiv');
    console.log('Element with ID "exampleDiv":', elementWithId);

    // Selecting elements with the class name "red"
    var redElements = document.querySelector('.red');
    console.log('Elements with class "red":', redElements);

    // Selecting the paragraph with the id "paragraph"
    var paragraph = document.querySelector('p#paragraph');
    console.log('Paragraph with ID "paragraph":', paragraph);

    // Selecting elements with the attribute "data-example" having the value "value"
    var elementsWithAttribute = document.querySelector('[data-example=value]');
    console.log('Elements with Attribute "data-example" with value "value":', elementsWith

    // Selecting the first element with the class "blue" within the container
    var blueElement = document.querySelector('.container .blue');
    console.log('First element with class "blue" within the container:', blueElement);
});
</script>
```

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




## ▼ Asynchronous(setTimeOut,setInterval)

### ▼ setTimeOut

```java
setTimeout(fun,1000)
console.log("This")

function fun(){
    console.log("Asynchronous")
}

console.log("Is");
```

I see, let me correct that.

In your code:

```java
setTimeout(fun, 0)
console.log("This")

function fun(){
    console.log("Asynchronous")
}
```


Khushal Savalakha

```
console.log("Is");
```

The output you observe, "This", "Is", and then "Asynchronous", can be explained by the event loop mechanism in JavaScript.

1. `setTimeout(fun, 0)` schedules the execution of the `fun` function to occur as soon as possible, but not immediately. Instead, it will be executed after the current synchronous code block completes.
2. `console.log("This")` is executed synchronously, printing "This" to the console.
3. The JavaScript engine continues executing the synchronous code and encounters `console.log("Is")`, printing "Is" to the console.
4. Now, the synchronous code execution is complete, and the JavaScript engine checks the event loop for any pending tasks. It finds the `fun` function scheduled by `setTimeout`.
5. The `fun` function is executed, printing "Asynchronous" to the console.

So, the output you see is:

```
This
Is
Asynchronous
```

Even though the `setTimeout` is scheduled with a timeout of 0 milliseconds, the actual execution of the callback function (`fun`) is deferred until the current synchronous code block is finished, which is why "Is" appears before "Asynchronous".

> Syntax:

The `setTimeout` function in JavaScript is used to schedule the execution of a function after a specified delay (in milliseconds). Here's the basic syntax of `setTimeout`:

```
setTimeout(callbackFunction, delayInMilliseconds, [param1, param2, ...]);
```

### Parameters

1. `callbackFunction`: The function to be executed after the delay.
2. `delayInMilliseconds`: The time to wait before executing the function, in milliseconds.
3. `param1, param2, ...` (optional): Additional arguments to pass to the `callbackFunction` when it is executed.

### Example

Let's look at a few examples to illustrate how `setTimeout` works.

### Basic Example

```
function greet() {
  console.log('Hello, world!');
}

// Execute `greet` after 2 seconds (2000 milliseconds)
setTimeout(greet, 2000);
```

In this example, the `greet` function will be called after a 2-second delay.

### Example with Anonymous Function

```
setTimeout(() => {
  console.log('This message is delayed by 3 seconds');
}, 3000);
```

Here, an anonymous function is scheduled to run after 3 seconds.


Khushal Savalakha

### Example with Additional Parameters

```
function sayMessage(message, name) {
  console.log(`${message}, ${name}!`);
}

// Execute `sayMessage` after 1 second, passing 'Hello' and 'Alice' as arguments
setTimeout(sayMessage, 1000, 'Hello', 'Alice');
```

In this example, the `sayMessage` function is called after 1 second with 'Hello' and 'Alice' as arguments, resulting in the output: `Hello, Alice!`

### Example of Cancelling a Timeout

You can cancel a timeout before it executes using `clearTimeout`.

```
const timeoutId = setTimeout(() => {
  console.log('This will not run');
}, 5000);

// Cancel the timeout
clearTimeout(timeoutId);
```

In this example, the scheduled function will never run because the timeout is cleared before it can execute.

### Summary

The `setTimeout` function is useful for scheduling tasks to run after a delay, creating pauses, or deferring execution of code. Understanding its syntax and how to use it effectively can help manage timing-related tasks in JavaScript applications.

```
// Define a function that will be executed later
function myFunction() {
    console.log("Function executed!");
}

// Set a timer to execute 'myFunction' after 3 seconds (3000ms)
const timer = setTimeout(myFunction, 3000);

// Cancel the timer after 1.5 seconds (1500ms)
setTimeout(() => {
    clearTimeout(timer); // This cancels the scheduled execution
    console.log("Function execution canceled!");
}, 1500);
```

> 💡 here in this code when code executes first timer comes so timer started and after 3 second it will executed but code does not stop for 3 second there he goes ahead it will see setTimeout with 1.5 sec because 1.5<3 sec so setTimeout with 1.5 sec will execute first and cancel 3 sec timeout am i true ?

Yes, you are absolutely correct! Here's a detailed explanation of how the code works:

1. **The First `setTimeout` (3 seconds)**:
   - When the first `setTimeout` is executed, it sets a timer for 3000 milliseconds (3 seconds) to run the `myFunction`.
   - However, the JavaScript code doesn't "pause" or wait for 3 seconds. JavaScript is asynchronous, so it moves on to execute the next line of code immediately.
2. **The Second `setTimeout` (1.5 seconds)**:


Khushal Savalakha

- After scheduling the 3-second timer, the next `setTimeout` is executed immediately, scheduling another timer that will run after 1500 milliseconds (1.5 seconds).
- This second timer is responsible for canceling the first timer (`clearTimeout(timer)`).

3. **Execution Flow**:
   - The second `setTimeout` (for 1.5 seconds) completes **before** the first one (for 3 seconds), because 1.5 seconds is less than 3 seconds.
   - After 1.5 seconds, the second timer triggers, calling `clearTimeout(timer)`, which cancels the 3-second timer.
   - Since the 3-second timer is canceled **before it completes**, `myFunction` is never executed, and the message `"Function execution canceled!"` is printed instead.

### Summary:

- The 1.5-second `setTimeout` cancels the 3-second `setTimeout` before it can execute.
- JavaScript continues to execute the code without waiting for the timers. It schedules them asynchronously and moves forward in the code.

> 💡 i want to understand how compiler or interpreter works here ?

To understand how the JavaScript compiler (or interpreter) works in conjunction with the event loop, timers, and asynchronous execution, let's break down the key concepts and processes involved in your code snippet:

### Key Concepts

1. **Compiler vs. Interpreter**:
   - **Compiler**: A program that translates high-level code into machine code (binary) before execution. In JavaScript, most engines like V8 (used in Chrome and Node.js) perform just-in-time (JIT) compilation, which means parts of the code are compiled as needed while executing.
   - **Interpreter**: A program that executes code line by line, translating it into machine code on the fly. JavaScript is primarily interpreted, which means the engine reads and executes the code directly.
2. **Event Loop**:
   - JavaScript has a single-threaded model, meaning it can execute one piece of code at a time. However, it can handle asynchronous operations via the event loop, which manages a queue of events and callbacks.
3. **Timers**:
   - Functions like `setTimeout` and `setInterval` are part of the Web API (or Node API). They schedule tasks to be executed after a certain time, but they don't block the main thread.

### How It Works Internally

Here's a step-by-step breakdown of how the JavaScript engine processes your code snippet:

1. **Code Execution Begins**:
   - The JavaScript engine starts executing your code from the top.
2. **Function Definition**:

```
function myFunction() {
    console.log("Function executed!");
}
```

   - The engine reads and stores the `myFunction` definition in memory but doesn't execute it yet.
3. **First `setTimeout` Call**:

```
const timer = setTimeout(myFunction, 3000);
```


Khushal Savalakha

- The engine sees the `setTimeout` function call:
  - It registers the function `myFunction` and the delay of 3000 milliseconds.
  - It creates a timer in the background and assigns a unique timer ID to `timer`.
  - The engine continues executing the next lines of code without waiting for 3 seconds.

4. **Second `setTimeout` Call**:

```
setTimeout(() => {
    clearTimeout(timer); // This cancels the scheduled execution
    console.log("Function execution canceled!");
}, 1500);
```

- The engine encounters another `setTimeout` call:
  - It registers an anonymous function with a delay of 1500 milliseconds.
  - It creates another timer in the background for this delay.
  - The engine continues execution to the next line.

5. **Waiting for Timers**:
   - Now, the engine has two timers running in the background:
     - **Timer 1** for `myFunction` (3000 ms).
     - **Timer 2** for the anonymous function (1500 ms).
   - The main thread is free to execute other code, but there is none left to execute at this point.
6. **Timer 2 Expires (after 1.5 seconds)**:
   - The timer for the anonymous function expires first:
     - The event loop picks this function from the task queue and executes it.
     - It calls `clearTimeout(timer)`, using the timer ID from the first `setTimeout` to cancel the scheduled execution of `myFunction`.
     - The log "Function execution canceled!" is printed to the console.
7. **Timer 1 Expires (after 3 seconds)**:
   - The timer for `myFunction` never executes because it was canceled by the previous step.
   - The event loop does not execute it, demonstrating the cancellation process.

## Summary of Compiler/Interpreter Role:

- **Parsing and Storing**: The engine parses the code, storing function definitions and setting up timers without executing them immediately.
- **Non-blocking Execution**: The engine uses an event loop to handle asynchronous tasks, allowing the code to run without blocking.
- **Task Queue Management**: When timers expire, their callbacks are placed in the task queue and executed when the main thread is available.

## Conclusion:

The JavaScript compiler and interpreter work together to manage code execution and asynchronous tasks efficiently. They enable non-blocking behavior, allowing developers to write code that can handle events and delays while keeping the user experience smooth. The combination of function registration, timer management, and the event loop plays a crucial role in how JavaScript executes code.

```
var cancellable = function(fn, args, t) {
    const cancelFn = function(){
        clearTimeout(timer)
    }
```


Khushal Savalakha

```
    const timer = setTimeout(() => {
        fn(...args)
    },t)
    return cancelFn
};//Leetcode
```

> 💡 here in this code cancelFn has no any timing constraints for delay of execution of clearTimeout
> but timer has timing constraints so here clearTimeout n always cancle the timer am i true?

Yes, you're absolutely right! Let's break it down further to clarify your understanding:

1. **The `cancellable` Function**:
   - The function `cancellable` takes three parameters: a function `fn`, an array of arguments `args`, and a delay `t` in milliseconds.
   - Inside this function, a `setTimeout` is used to schedule the execution of `fn(...args)` after `t` milliseconds.
2. **The `timer` Variable**:
   - The `timer` variable holds the ID returned by `setTimeout`. This ID represents the scheduled execution of `fn` after the specified delay `t`.
3. **The `cancelFn` Function**:
   - The `cancelFn` function is defined to call `clearTimeout(timer)`.
   - This means when `cancelFn` is invoked, it cancels the timer represented by `timer`, preventing `fn` from being executed if it hasn't been called yet.
4. **No Timing Constraints for `cancelFn`**:
   - You are correct that `cancelFn` itself has no timing constraints. It can be called at any time, and when it is called, it will cancel the scheduled execution of `fn` regardless of whether that execution is scheduled for 1 millisecond or 1000 milliseconds in the future.
   - So, `clearTimeout(timer)` will effectively stop the timer whenever `cancelFn` is called, as long as it is called before the delay of `t` milliseconds has completed.

### Summary:

- **Timing for `setTimeout`**: The execution of `fn(...args)` is constrained by the delay `t`, meaning it will only execute after that time has passed unless it is canceled.
- **Immediate Cancellation**: `cancelFn` can be called immediately (or at any point) to cancel that timer without any timing constraints, allowing it to stop the execution of `fn` before `t` milliseconds have elapsed.

In essence, as long as `cancelFn` is invoked before the timer reaches `t` milliseconds, it will cancel the execution of `fn`, demonstrating how `clearTimeout` works effectively to manage asynchronous operations in JavaScript.

## ▼ setInterval

The `setInterval` function in JavaScript is used to repeatedly execute a function at specified time intervals. Here's the basic syntax:

```
setInterval(callbackFunction, intervalInMilliseconds, [param1, param2, ...]);
```

### Parameters

1. `callbackFunction`: The function to be executed repeatedly.
2. `intervalInMilliseconds`: The time interval (in milliseconds) between each execution of the `callbackFunction`.
3. `param1, param2, ...` (optional): Additional arguments to pass to the `callbackFunction` when it is executed.


Khushal Savalakha

### Example

Let's look at a few examples to illustrate how `setInterval` works.

### Basic Example

```
function greet() {
  console.log('Hello, world!');
}

// Execute `greet` every 2 seconds (2000 milliseconds)
setInterval(greet, 2000);
```

In this example, the `greet` function will be called every 2 seconds.

### Example with Anonymous Function

```
setInterval(() => {
  console.log('This message is logged every 3 seconds');
}, 3000);
```

Here, an anonymous function is scheduled to run every 3 seconds.

### Example with Additional Parameters

```
function sayMessage(message, name) {
  console.log(`${message}, ${name}!`);
}

// Execute `sayMessage` every 1 second, passing 'Hello' and 'Alice' as arguments
setInterval(sayMessage, 1000, 'Hello', 'Alice');
```

In this example, the `sayMessage` function is called every 1 second with 'Hello' and 'Alice' as arguments, resulting in the output: `Hello, Alice!` every second.

### Example of Clearing an Interval

You can stop the repeated execution by using `clearInterval`.

```
const intervalId = setInterval(() => {
  console.log('This will be cleared soon');
}, 2000);

// Cancel the interval after 10 seconds
setTimeout(() => {
  clearInterval(intervalId);
  console.log('Interval cleared');
}, 10000);
```

In this example, the interval is cleared after 10 seconds, so the message "This will be cleared soon" will only be logged a few times before the interval is stopped.

### Summary

The `setInterval` function is useful for executing a function repeatedly at fixed time intervals. Understanding its syntax and how to use it effectively can help manage tasks that need to be performed periodically in JavaScript applications.

function Increment(a,b){
console.log(a+b);
}
a=9


Khushal Savalakha

b=4
// setInterval(Increment,1000,++a,++b)
setInterval(Increment,1000,[++a,++b])

PS D:\Javascript\College\Sem-4> node "d:\Javascript\College\Sem-4\Task7.js"
10,5undefined
10,5undefined
10,5undefi...

why above code gives undefined atleat ?

The behavior you're seeing is due to how `setInterval` and the increment operators (`++a` and `++b`) are being used in your code.

### Explanation

When you use `setInterval`, the parameters passed after the interval (in milliseconds) are passed to the callback function (`Increment` in this case) each time it is called. However, the way you've structured the code, specifically the way you're incrementing `a` and `b`, and the array usage, is causing the function to not behave as expected.

### Key Points:

1. **Increment Operators**: `++a` and `++b` increment the values of `a` and `b` before passing them to `setInterval`. This means `a` becomes `10` and `b` becomes `5` before `setInterval` is even called.
2. **Array Usage**: Passing `[++a, ++b]` as a single argument means `Increment` receives one argument which is an array.

### Correct Approach

You should pass `a` and `b` directly as separate arguments to `setInterval`. Also, avoid using an array unless you intend to handle an array within `Increment`.

### Example

Here's how you can fix the code:

```
function Increment(a, b) {
    console.log(a + b);
}

let a = 9;
let b = 4;

// Incrementing `a` and `b` before passing them to `setInterval`
setInterval(() => Increment(++a, ++b), 1000);
```

### Explanation of Corrected Code

- **Incrementing Inside the Callback**: By using an arrow function `() => Increment(++a, ++b)`, `a` and `b` are incremented each time before they are passed to `Increment`.
- **Separate Arguments**: `Increment` is called with two separate arguments, `a` and `b`, rather than an array.

### Full Example with Console Logging for Clarity

```
function Increment(a, b) {
    console.log(a + b);
}

let a = 9;
let b = 4;

setInterval(() => {
    console.log(`Incrementing a: ${++a}, b: ${++b}`);
```


Khushal Savalakha

```
    Increment(a, b);
}, 1000);
```

In this corrected example:

- `a` and `b` are incremented every second.
- The `Increment` function receives the updated values each time and logs the sum.
- The console log inside the interval helps track the values of `a` and `b`.

### Expected Output

```
Incrementing a: 10, b: 5
15
Incrementing a: 11, b: 6
17
Incrementing a: 12, b: 7
19
...
```

This should resolve the issue of `undefined` and ensure the values are incremented and logged correctly each second.

function Increment(a,b){
console.log(a+b);
}
a=9
b=4
setInterval(Increment,1000,++a,++b)

PS D:\Javascript\College\Sem-4> node "d:\Javascript\College\Sem-4\Task7.js"
15
15
15
15

why it is not incrementing ?

The issue is that the parameters passed to `setInterval` are evaluated only once at the time `setInterval` is called. In your case, the values `++a` and `++b` are incremented once, and then `setInterval` is repeatedly calling `Increment` with those same values.

To achieve the desired effect of incrementing `a` and `b` each time before calling `Increment`, you need to use a function that increments `a` and `b` inside the `setInterval` callback.

Here's how you can modify your code to increment `a` and `b` every time `Increment` is called:

### Corrected Code

```
function Increment(a, b) {
    console.log(a + b);
}

let a = 9;
let b = 4;

setInterval(() => {
    Increment(++a, ++b);
}, 1000);
```

### Explanation

- **Arrow Function**: By using an arrow function as the callback to `setInterval`, you can ensure that `a` and `b` are incremented every time the interval fires.


Khushal Savalakha

- **Increment Inside the Callback**: The `++a` and `++b` operations are now inside the callback function, so they are evaluated each time the interval fires, leading to `a` and `b` being incremented on each call.

### Expected Output

With the corrected code, the output should look something like this, incrementing the values of `a` and `b` each time:

```
15
17
19
21
...
```

### Detailed Breakdown

1. **Initial Values**:
   - `a = 9`
   - `b = 4`
2. **First Interval Execution**:
   - `++a` increments `a` to `10`
   - `++b` increments `b` to `5`
   - `Increment(10, 5)` is called, logging `15`.
3. **Second Interval Execution**:
   - `++a` increments `a` to `11`
   - `++b` increments `b` to `6`
   - `Increment(11, 6)` is called, logging `17`.
4. **Subsequent Executions**:
   - This pattern continues, with `a` and `b` being incremented before each call to `Increment`.

### Conclusion

Using an arrow function within `setInterval` allows you to update the values of `a` and `b` dynamically each time the interval runs, which is the intended behavior for incrementing the values on each call.

The issue is that the parameters passed to `setInterval` are evaluated only once at the time `setInterval` is called. In your case, the values `++a` and `++b` are incremented once, and then `setInterval` is repeatedly calling `Increment` with those same values.

To achieve the desired effect of incrementing `a` and `b` each time before calling `Increment`, you need to use a function that increments `a` and `b` inside the `setInterval` callback.

Here's how you can modify your code to increment `a` and `b` every time `Increment` is called:

### Corrected Code

```javascript
function Increment(a, b) {
    console.log(a + b);
}

let a = 9;
let b = 4;

setInterval(() => {
    Increment(++a, ++b);
}, 1000);
```


Khushal Savalakha

## Explanation

- **Arrow Function**: By using an arrow function as the callback to `setInterval`, you can ensure that `a` and `b` are incremented every time the interval fires.
- **Increment Inside the Callback**: The `++a` and `++b` operations are now inside the callback function, so they are evaluated each time the interval fires, leading to `a` and `b` being incremented on each call.

## Expected Output

With the corrected code, the output should look something like this, incrementing the values of `a` and `b` each time:

```
15
17
19
21
...
```

## Detailed Breakdown

1. **Initial Values**:
   - `a = 9`
   - `b = 4`
2. **First Interval Execution**:
   - `++a` increments `a` to `10`
   - `++b` increments `b` to `5`
   - `Increment(10, 5)` is called, logging `15`.
3. **Second Interval Execution**:
   - `++a` increments `a` to `11`
   - `++b` increments `b` to `6`
   - `Increment(11, 6)` is called, logging `17`.
4. **Subsequent Executions**:
   - This pattern continues, with `a` and `b` being incremented before each call to `Increment`.

## Conclusion

Using an arrow function within `setInterval` allows you to update the values of `a` and `b` dynamically each time the interval runs, which is the intended behavior for incrementing the values on each call.


Khushal Savalakha

Given a function `fn`, an array of arguments `args`, and an interval time `t`, return a cancel function `cancelFn`.

After a delay of `cancelTimeMs`, the returned cancel function `cancelFn` will be invoked.

```
setTimeout(cancelFn, cancelTimeMs)
```

The function `fn` should be called with `args` immediately and then called again every `t` milliseconds until `cancelFn` is called at `cancelTimeMs` ms.

**Example 1:**

```
Input: fn = (x) => x * 2, args = [4], t = 35
Output:
[
   {"time": 0, "returned": 8},
   {"time": 35, "returned": 8},
   {"time": 70, "returned": 8},
   {"time": 105, "returned": 8},
   {"time": 140, "returned": 8},
   {"time": 175, "returned": 8}
]
Explanation:
const cancelTimeMs = 190;
const cancelFn = cancellable((x) => x * 2, [4], 35);
setTimeout(cancelFn, cancelTimeMs);

Every 35ms, fn(4) is called. Until t=190ms, then it is cancelled.
1st fn call is at 0ms. fn(4) returns 8.
2nd fn call is at 35ms. fn(4) returns 8.
3rd fn call is at 70ms. fn(4) returns 8.
4th fn call is at 105ms. fn(4) returns 8.
5th fn call is at 140ms. fn(4) returns 8.
6th fn call is at 175ms. fn(4) returns 8.
Cancelled at 190ms
```

```j
var cancellable = function (fn, args, t) {
    fn(...args);

    const timer = setInterval(() => {
        fn(...args);
    }, t);
    const cancelFn = function () {
        clearInterval(timer);
    };
    return cancelFn;
};//Leetcode
```

### Explanation of the Code:

- The code defines a `cancellable` function that executes a provided function `fn` immediately and at regular intervals defined by `t`.
- It logs the execution time and results in an array, which is printed after the cancellation.
- The cancellation is done using the `clearInterval` function, stopping the repeated execution once the specified `cancelTimeMs` is reached.

This format provides a clear overview of the solution, its approach, and its complexities, along with example usage.


Khushal Savalakha

## ▼ Basic & Slice

In JavaScript, the `slice()` method is used to extract a section of a string and returns it as a new string. It doesn't modify the original string.

The basic syntax for the `slice()` method is:

```
string.slice(startIndex, endIndex)
```

- `startIndex`: The index at which to begin extraction. If negative, it specifies the position from the end of the string.
- `endIndex` (optional): The index at which to end extraction. The extracted string will include the character at this index unless the endIndex is omitted, in which case it extracts to the end of the string. If negative, it specifies the position from the end of the string.

Here's how it works:

- If both `startIndex` and `endIndex` are positive, `slice()` extracts characters from `startIndex` up to but not including `endIndex`.
- If `startIndex` is negative, `slice()` extracts characters from the end of the string, starting at `string.length + startIndex`, up to but not including `endIndex`.
- If `endIndex` is negative, it specifies the index of the character before the one at `string.length + endIndex`.

Examples:

```
let str = 'Hello, world!';

// Extracts characters from index 7 (inclusive) to the end
let sliced1 = str.slice(7); // "world!"

// Extracts characters from index 1 (inclusive) to index 5 (exclusive)
let sliced2 = str.slice(1, 5); // "ello"

// Extracts characters from index -6 (6th character from the end) to the end
let sliced3 = str.slice(-6); // "world!"

// Extracts characters from index -8 (8th character from the end) to index -1 (inclusiv
e)
let sliced4 = str.slice(-8, -1); // "lo, worl"
```

In summary, `slice()` provides a flexible way to extract substrings from a string in JavaScript.

I see, let's simplify the approach a bit and ensure that the `oldData` variable is properly updated and restored. We'll directly store the original data within the functions. Here's the modified 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>
  <script>
    function upper() {
      var pData = document.getElementById('data');
      var oldData = pData.textContent;
      pData.textContent = oldData.toUpperCase();
    }

    function restoreData() {
      var pData = document.getElementById('data');
      var oldData = pData.getAttribute('data-original');
      if (oldData) {
```


Khushal Savalakha

```
        pData.textContent = oldData;
      }
    }

    function lower() {
      var pData = document.getElementById('data');
      var oldData = pData.textContent;
      pData.textContent = oldData.toLowerCase();
    }
  </script>
</head>
<body>
  <p id="data" data-original="Lorem ipsum dolor sit amet consectetur adipisicing elit. E
st dolores tempore ab, voluptates numquam quaerat molestiae facere iusto minus sapient
e!">
    Lorem ipsum dolor sit amet consectetur adipisicing elit. Est dolores
    tempore ab, voluptates numquam quaerat molestiae facere iusto minus
    sapiente!
  </p>
  <!-- Add double-click event handler for restoring data -->
  <input type="button" value="Uppercase" onclick="upper()" ondblclick="restoreData()"/>
  <input type="button" value="lowercase" onclick="lower()" ondblclick="restoreData()" />
</body>
</html>
```

In this code, the original text is directly stored as a data attribute ( `data-original` ) in the paragraph element. When the `restoreData()` function is called, it retrieves the original text from this data attribute and sets the paragraph's text content accordingly. This approach should properly restore the original text when double-clicking the buttons.

The `data-original` attribute in the HTML code serves as a custom data attribute attached to the paragraph ( `<p>` ) element. Custom data attributes in HTML5 allow you to store extra information within your HTML elements that can be accessed and manipulated using JavaScript.

In this specific case, the `data-original` attribute is being used to store the original text content of the paragraph element. When the text content is modified (e.g., converted to uppercase or lowercase), the original text is stored in this attribute. Later, when the `restoreData()` function is called, it retrieves the original text from this attribute and restores it as the paragraph's text content.

So, essentially, `data-original` serves as a mechanism to retain the original text content of the paragraph element so that it can be restored later when needed.

This line of code is part of a JavaScript function that is triggered when a key is pressed in the input field with the id "Output". Here's a breakdown of what it does:

This line of code is part of a JavaScript function that is triggered when a key is pressed in the input field with the id "Output". Here's a breakdown of what it does:

1. var key=String.fromCharCode(event.which);: This line converts the key code of the pressed key (which is stored in event.which) to a character. For example, if the key code is 65, this line will convert it to 'A'.
2. Output=Output+key+'23';: This line appends the converted character to the Output string, followed by the characters '23'. The '23' is added as a delimiter, which might be used to separate different key presses in the Output string.
   So, if you press the 'A' key, the Output string will be updated to 'A23'. If you then press the 'B' key, the Output string will be updated to 'A23B23', and so on.

In JavaScript, you can create arrays using square brackets `[]`. Arrays can hold any type of data, including numbers, strings, objects, and even other arrays. Here's how you can create an array:

```
// Creating an array of numbers
let numbers = [1, 2, 3, 4, 5];

// Creating an array of strings
let fruits = ['apple', 'banana', 'orange'];
```


Khushal Savalakha

```
// Creating an array of mixed data types
let mixedArray = [1, 'hello', true, {name: 'John'}, [6, 7, 8]];
```

To convert data into lowercase in JavaScript, you can use the `toLowerCase()` method. This method converts all characters in a string to lowercase. Here's how you can use it:

```
let str = "Hello World";
let lowerCaseStr = str.toLowerCase(); // lowerCaseStr will be "hello world"
```

Similarly, if you want to convert the elements of an array to lowercase, you can iterate through the array and apply `toLowerCase()` to each element:

```
let fruits = ['Apple', 'Banana', 'Orange'];

// Convert all elements to lowercase
for (let i = 0; i < fruits.length; i++) {
    fruits[i] = fruits[i].toLowerCase();
}

console.log(fruits); // Output: ['apple', 'banana', 'orange']
```

Alternatively, you can use the `map()` function to create a new array with all elements converted to lowercase:

```
let fruits = ['Apple', 'Banana', 'Orange'];

let lowerCaseFruits = fruits.map(fruit => fruit.toLowerCase());

console.log(lowerCaseFruits); // Output: ['apple', 'banana', 'orange']
```

These are basic operations in JavaScript for creating arrays and converting data to lowercase. You can further explore JavaScript documentation and tutorials for more advanced concepts and operations.

The provided code is a JavaScript function named [conf()](file:///d%3A/Javascript/Collage/java_script_6.html#12%2C18-12%2C18). Here's a breakdown of what it does:

1. `var ret=confirm;` : This line is trying to assign the `confirm` function to the variable `ret`. However, there's a mistake here. The correct way to call the `confirm` function is `var ret=confirm();`. The `confirm` function is a built-in JavaScript function that displays a dialog box with a specified message, along with an OK and a Cancel button.
2. `("You want to continue ?");` : This line is trying to call the `confirm` function with the message "You want to continue ?". However, due to the mistake in the previous line, this line will not work as expected.
3. `if(ret==true){ document.write("Continue"); } else{ document.write("do not"); }` : This is an `if-else` statement that checks the value of `ret`. If `ret` is `true`, it means the user clicked the OK button in the confirm dialog, so it writes "Continue" to the document. If `ret` is `false`, it means the user clicked the Cancel button, so it writes "do not" to the document.

Here's the corrected code:

```
function conf(){
    var ret=confirm("You want to continue ?");
    if(ret==true){
        document.write("Continue");
    }
    else{
        document.write("do not");
    }
}
```

`var` and `let` are both keywords used for variable declaration in JavaScript, but they have some differences in terms of scope and hoisting.


Khushal Savalakha

1. `var`: Variables declared with `var` are function-scoped or globally-scoped. They are hoisted to the top of their scope and can be accessed before they are declared. However, they are not block-scoped, which means they can be re-declared and updated within the same scope.
2. `let`: Variables declared with `let` are block-scoped. They are hoisted to the top of their block, but not initialized. This means you cannot access them before the line where they are declared. Also, they cannot be re-declared within the same scope, but they can be updated.

Here's an example:

```
function example() {
    if (true) {
        var a = 10;
        let b = 20;
    }
    console.log(a); // 10
    console.log(b); // ReferenceError: b is not defined
}
```

In this code, `a` is accessible outside the `if` block because it's declared with `var`, which is function-scoped. However, `b` is not accessible outside the `if` block because it's declared with `let`, which is block-scoped.

In JavaScript, if you don't use the `var`, `let`, or `const` keyword when assigning a value to a variable, the variable is automatically declared as a global variable.

However, it's a good practice to always use `var`, `let`, or `const` when declaring variables to avoid polluting the global namespace.

Here's how you would declare a variable and assign a value to it:

- `var a = 'kvjdsk';` - This declares a variable `a` and assigns it the value `'kvjdsk'`. The `var` keyword is used here.
- `let a = 'kvjdsk';` - This declares a variable `a` and assigns it the value `'kvjdsk'`. The `let` keyword is used here.
- `const a = 'kvjdsk';` - This declares a constant `a` and assigns it the value `'kvjdsk'`. The `const` keyword is used here.

The `textContent` and `value` properties serve different purposes and are used with different types of HTML elements.

1. **`textContent`:**
   - The `textContent` property is used to get or set the text content of an HTML element and all its descendants.
   - It returns the text content as a string, including the text of any nested elements within the specified element.
   - It is commonly used with elements like `<div>`, `<p>`, `<span>`, etc., to manipulate their textual content.
2. **`value`:**
   - The `value` property is primarily used with form input elements (`<input>`, `<textarea>`, `<select>`) to get or set their current value.
   - For `<input>` elements, the `value` property represents the current value of the input field, which could be text, a number, or other types depending on the input type (e.g., text, checkbox, radio, etc.).
   - For `<textarea>` elements, the `value` property represents the text content of the textarea.
   - For `<select>` elements (dropdown lists), the `value` property represents the value of the selected option.
   - It's typically used to manipulate the data entered by users in form fields or to set default values dynamically.

In summary, `textContent` deals with the textual content within HTML elements, while `value` specifically deals with the value of form input elements.


Khushal Savalakha

### Changing HTML Style

To change the style of an HTML element, use this syntax:

```
document.getElementById(id).style.property = 'style'
```

**Example**

```
document.getElementById('p1').style.backgroundColor = "red";
```

### Changing HTML Content

The easiest way to modify the content of an HTML element is by using the innerHTML property.

To change the content of an HTML element, use this syntax:

```
document.getElementById(id).innerHTML = 'new HTML'
```

**Example**

```
document.getElementById('p1').innerHTML = "red";
```

### Changing the Value of an Attribute

To change the value of an HTML attribute, use this syntax:

```
document.getElementById(id).attribute = new value
```

**Example**

```
document.getElementById('p1').title = "demo";
```

## ▼ Variables in javascript

`var` : If you redeclare a variable using `var`, it's not an error. The new declaration simply overrides the old one. This is called variable hoisting. The variable is moved to the top of its scope and its value is set to `undefined` until the line where it's declared is reached.

`let` : If you redeclare a variable using `let`, it's an error. The JavaScript engine throws a [SyntaxError] (file:///d%3A/Javascript/Practice/Varibles_In_Javascript.html#33%2C19-33%2C19) because it's not allowed to redeclare a variable using `let` in the same scope. This is a feature of ES6 (ECMAScript 2015) and later versions of JavaScript.

```javascript
function exampleVar() {
  if (true) {
    var x = 10; // Declaring x with var
    var x = 90; // Redeclaring x with var, which is allowed
  }
  console.log('X:'+x); // Output: 90
}

function exampleLet() {
  if (true) {
    let y = 8; // Declaring y with let
    let y = 20; // Redeclaring y with let, which is not allowed
  }
  console.log(y); // ReferenceError: y is not defined
}
```

In the `exampleVar` function, you're redeclaring `x` with `var`, which is allowed in JavaScript. The last declaration is used, so `x` is 90 when you log it.

In the `exampleLet` function, you're trying to redeclare `y` with `let`, which is not allowed in JavaScript. This is why you're getting a `ReferenceError: y is not defined` when you try to log `y` after the second `let y = 20;`.

### ▼ Varible Hoisting

Variable hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compile phase. This means that you can use a variable or a function before it has been declared.

However, only the declaration is hoisted, not the initialization. If you try to access a variable before it has been declared and initialized, it will have the value `undefined`.


Khushal Savalakha

```
console.log(x); // Output: undefined
var x = 5;
```

In the above code, `x` is declared and initialized after the `console.log` statement. However, the `console.log` statement doesn't throw an error. This is because the declaration of `x` is hoisted to the top of its scope, so the code is actually interpreted like this:

```
var x;
console.log(x); // Output: undefined
x = 5;
```

This is not the case with `let` and `const`. They are also hoisted, but they are not initialized. So, if you try to access a `let` or `const` variable before it has been declared and initialized, you will get a `ReferenceError`.

**means i write 1 varible which like var a=9; here a is going to the top and tell thet there is a varible named a is coming inn particular code and it exists ?**

Yes, that's correct. When you declare a variable using `var`, the declaration is hoisted to the top of its scope. This means that the JavaScript engine knows about the variable [a](file:///d%3A/Javascript/Practice/Varibles_In_Javascript.html#31%2C14-31%2C14) as soon as it starts to compile the code, even if the line `var a = 9;` hasn't been reached yet.

```
console.log(a); // Output: undefined
var a = 9;
console.log(a); // Output: 9
```

In the above code, the first `console.log` statement doesn't throw an error. This is because the declaration of `a` is hoisted to the top of its scope, so the code is actually interpreted like this:

```
var a;
console.log(a); // Output: undefined
a = 9;
console.log(a); // Output: 9
```

This is not the case with `let` and `const`. They are also hoisted, but they are not initialized. So, if you try to access a `let` or `const` variable before it has been declared and initialized, you will get a `ReferenceError`.

## ▼ Data Types

Primitive data types and non-primitive (or reference) data types are fundamental concepts in programming languages like JavaScript. Here's a breakdown of their differences:

1. **Primitive Data Types:**
   - Primitive data types are simple, immutable data types.
   - They are directly stored in memory and accessed by their value.
   - In JavaScript, the primitive data types are:
     - `string`: Represents textual data.
     - `number`: Represents numeric values, both integer and floating-point.
     - `boolean`: Represents a logical value, `true` or `false`.
     - `undefined`: Represents an undefined value.
     - `null`: Represents an absence of any value.
     - `symbol`: Represents a unique identifier (introduced in ECMAScript 6).
   - Examples:

     ```
     let name = "John"; // string
     let age = 30; // number
     ```


Khushal Savalakha

```
let isStudent = true; // boolean
let score = null; // null
let height; // undefined
```

2. **Non-Primitive (Reference) Data Types:**
   - Non-primitive data types are more complex and mutable.
   - They are not stored directly in memory but instead stored as references (addresses) to the actual data.
   - Changes made to reference data types affect all references to the same data.
   - In JavaScript, non-primitive data types include:
     - `object`: Represents a collection of key-value pairs.
     - `array`: A special type of object that stores a collection of elements.
     - `function`: A special type of object that can be invoked.
   - Examples:

```
let person = { name: "Alice", age: 25 }; // object
let numbers = [1, 2, 3, 4, 5]; // array
function greet() { console.log("Hello!"); } // function
```

3. **Comparison:**
   - Primitive data types are compared by their value, while non-primitive data types are compared by their reference.
   - Example:

```
let a = 5;
let b = 5;
console.log(a === b); // true (values are the same)

let arr1 = [1, 2, 3];
let arr2 = [1, 2, 3];
console.log(arr1 === arr2); // false (different references)
```

let arr1 = [1, 2, 3];
let arr2 = [1, 2, 3];
console.log(arr1 == arr2); // false

it means here if we write == or === both time they only see reference ?

Yes, that's correct. In JavaScript, both the `==` and `===` operators check references when comparing objects such as arrays.

- The `==` operator performs type coercion before comparison and then checks if the references are the same.
- The `===` operator checks if both operands have the same value and are of the same type, but for objects (including arrays), it still compares their references.

Therefore, regardless of whether you use `==` or `===`, the comparison between two different array objects will always result in `false` because they refer to different memory locations.

4. **Copying:**
   - When you assign a primitive data type to a variable, a copy of the value is made.
   - When you assign a non-primitive data type to a variable, a reference to the original data is made.
   - Example:

```
let x = 10;
let y = x; // y now has a copy of the value of x
y = 20; // Changing y does not affect x
```


Khushal Savalakha

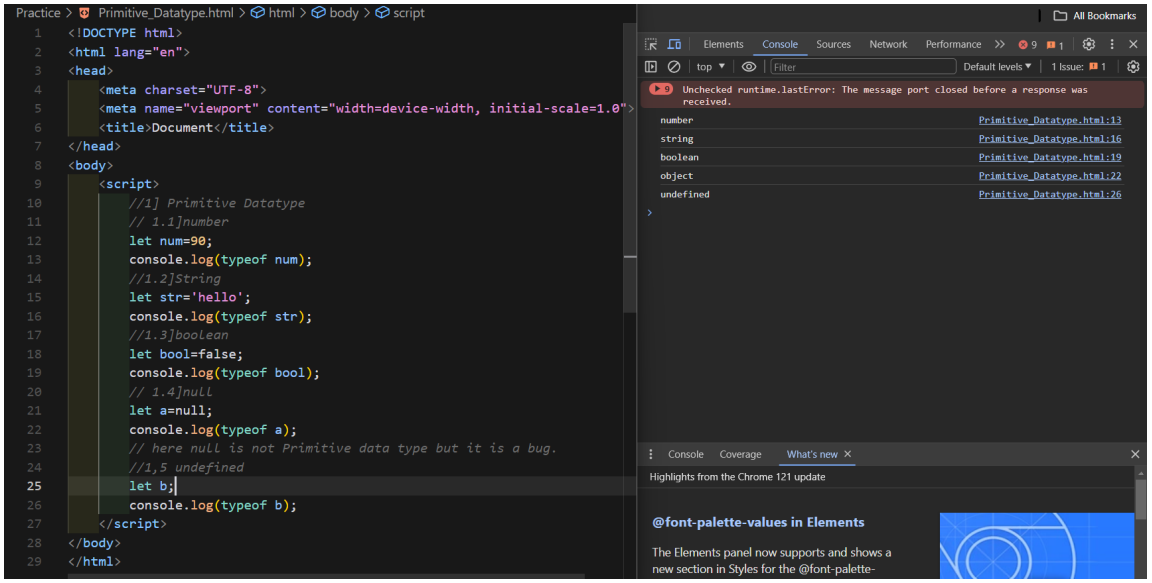
```
let obj1 = { name: "John" };
let obj2 = obj1; // obj2 now refers to the same object as obj1
obj2.name = "Alice"; // Changing obj2 also affects obj1
console.log(obj1.name); // Output: "Alice"
```

Understanding the distinction between primitive and non-primitive data types is crucial for writing efficient and bug-free code.

### ▼ Primitive Datatype

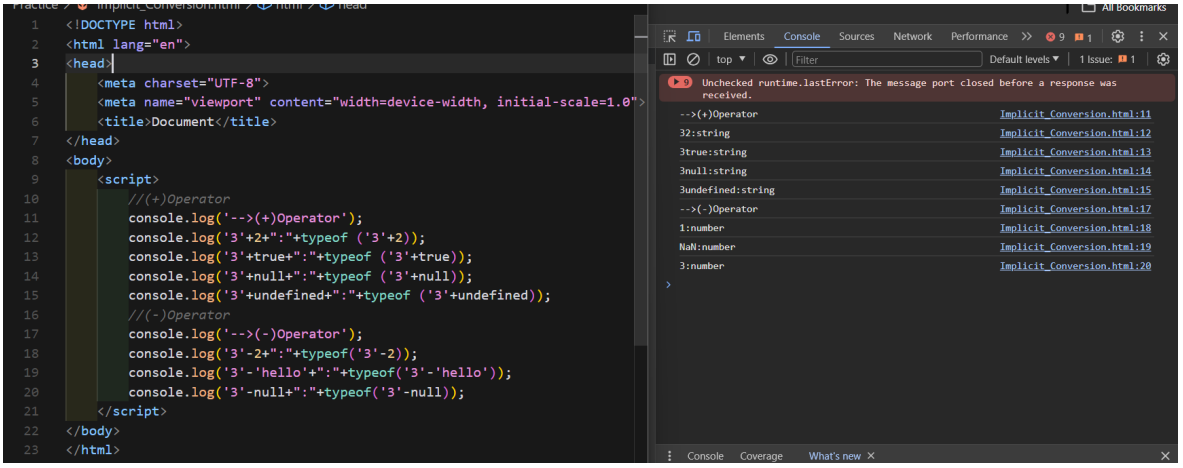
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        //1] Primitive Datatype
        // 1.1]number
        let num=90;
        console.log(typeof num);
        //1.2]String
        let str='hello';
        console.log(typeof str);
        //1.3]boolean
        let bool=false;
        console.log(typeof bool);
        // 1.4]null
        let a=null;
        console.log(typeof a);
        // here null is not Primitive data type but it is a bug.
        //1,5 undefined
        let b;
        console.log(typeof b);
    </script>
</body>
</html>
```


Khushal Savalakha



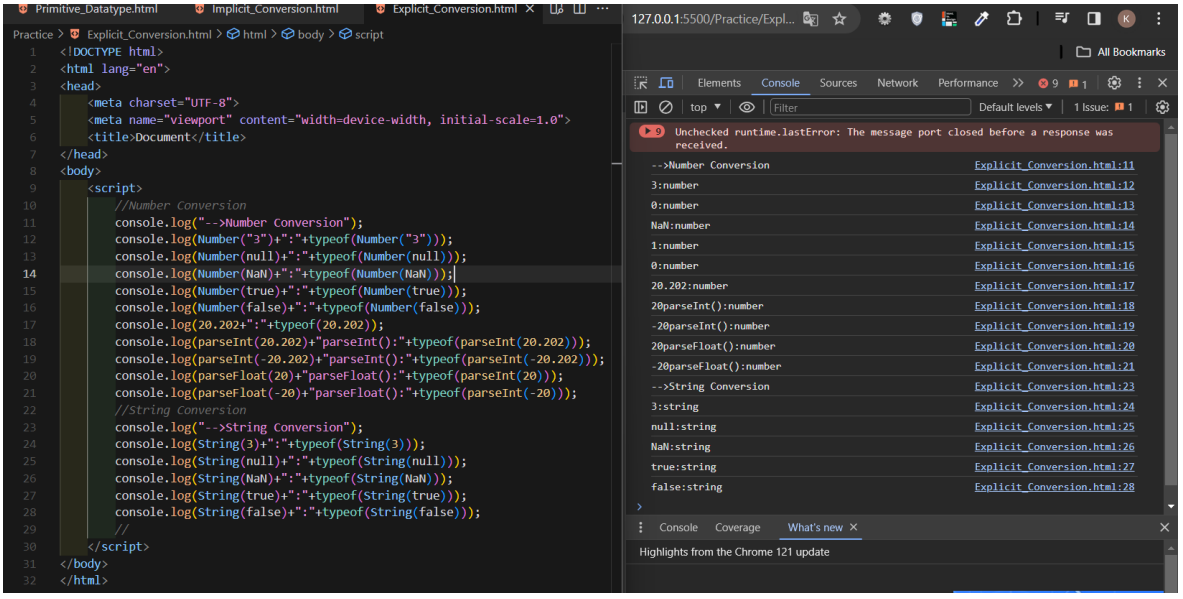

## ▼ Implicit Conversion

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        //(+)Operator
        console.log('-->(+)Operator');
        console.log('3'+2+":"+typeof ('3'+2));
        console.log('3'+true+":"+typeof ('3'+true));
        console.log('3'+null+":"+typeof ('3'+null));
        console.log('3'+undefined+":"+typeof ('3'+undefined));
        //(-)Operator
        console.log('-->(-)Operator');
        console.log('3'-2+":"+typeof('3'-2));
        console.log('3'-'hello'+":"+typeof('3'-'hello'));
        console.log('3'-null+":"+typeof('3'-null));
    </script>
</body>
</html>
```




## ▼ Explicit Conversion

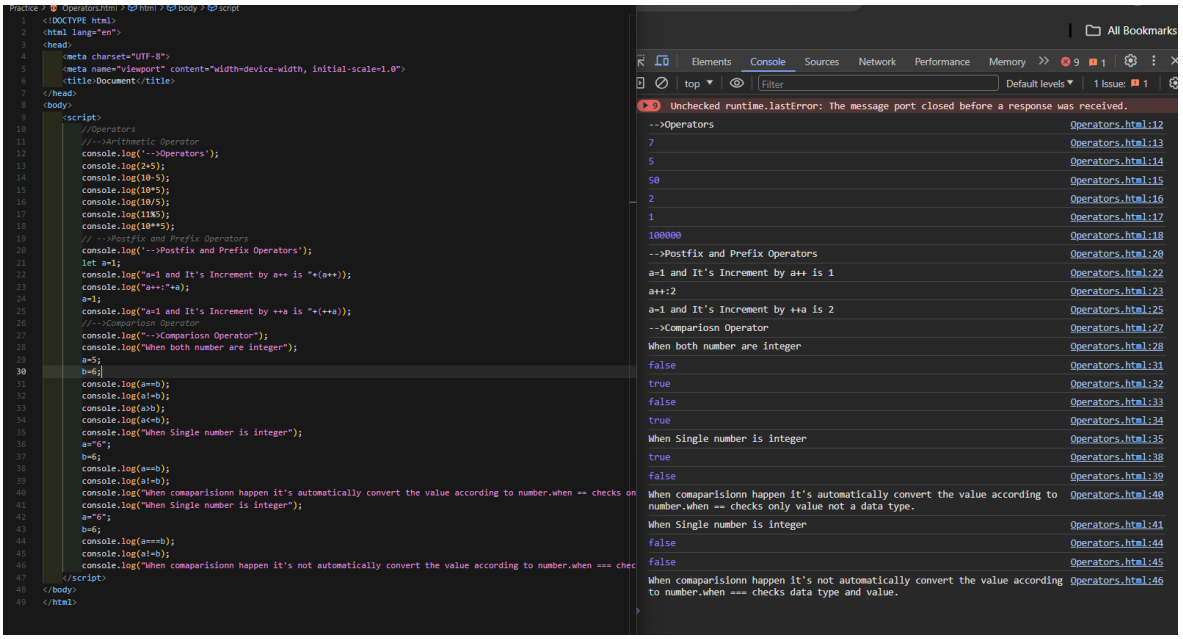
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        //Number Conversion
        console.log("-->Number Conversion");
        console.log(Number("3")+":"+typeof(Number("3")));
        console.log(Number(null)+":"+typeof(Number(null)));
        console.log(Number(NaN)+":"+typeof(Number(NaN)));
        console.log(Number(true)+":"+typeof(Number(true)));
        console.log(Number(false)+":"+typeof(Number(false)));
        console.log(20.202+":"+typeof(20.202));
        console.log(parseInt(20.202)+"parseInt():"+typeof(parseInt(20.202)));
        console.log(parseInt(-20.202)+"parseInt():"+typeof(parseInt(-20.202)));
        console.log(parseFloat(20)+"parseFloat():"+typeof(parseInt(20)));
        console.log(parseFloat(-20)+"parseFloat():"+typeof(parseInt(-20)));
        //String Conversion
        console.log("-->String Conversion");
        console.log(String(3)+":"+typeof(String(3)));
        console.log(String(null)+":"+typeof(String(null)));
        console.log(String(NaN)+":"+typeof(String(NaN)));
        console.log(String(true)+":"+typeof(String(true)));
        console.log(String(false)+":"+typeof(String(false)));
    </script>
</body>
</html>
```




## ▼ Operators

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        //Operators
        //-->Arithmetic Operator
        console.log('-->Operators');
        console.log(2+5);
        console.log(10-5);
        console.log(10*5);
        console.log(10/5);
        console.log(11%5);
        console.log(10**5);
        // -->Postfix and Prefix Operators
        console.log('-->Postfix and Prefix Operators');
        let a=1;
        console.log("a=1 and It's Increment by a++ is "+(a++));
        console.log("a++:"+a);
        a=1;
        console.log("a=1 and It's Increment by ++a is "+(++a));
        //-->Compariosn Operator
        console.log("-->Compariosn Operator");
        console.log("When both number are integer");
        a=5;
        a=6;
        console.log(a==b);
        console.log(a!=b);
        console.log(a>b);
        console.log(a<=b);
        console.log("When Single number is integer");
        a="6";
        a=6;
```

```
        console.log(a==b);
        console.log(a!=b);
        console.log("When comaparisionn happen it's automatically convert the value accord
        console.log("When Single number is integer");
        a="6";
        a=6;
        console.log(a===b);
        console.log(a!=b);
        console.log("When comaparisionn happen it's not automatically convert the value ac
    </script>
</body>
</html>
```




## LOGICAL OPERATORS

| A | B | A \|\| B | A && B | !A |
|---|---|---|---|---|
| False | False | False | False | True |
| True | False | True | False | False |
| False | True | True | False | True |
| True | True | True | True | False |

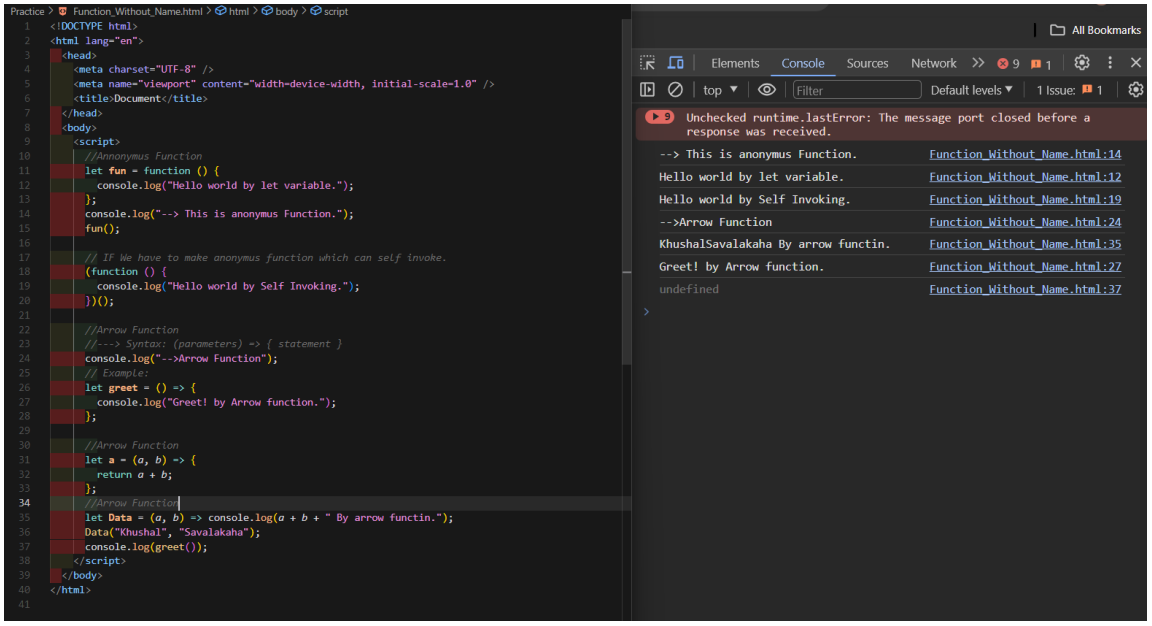
## ▼ Functions

### ▼ Arrow Function & For Each

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


Khushal Savalakha

```html
    <title>Document</title>
  </head>
  <body>
    <script>
      //Annonymus Function
      let fun = function () {
        console.log("Hello world by let variable.");
      };
      console.log("--> This is anonymus Function.");
      fun();

      // IF We have to make anonymus function which can self invoke.
      (function () {
        console.log("Hello world by Self Invoking.");
      })();

      //Arrow Function
      //---> Syntax: (parameters) => { statement }
      console.log("-->Arrow Function");
      // Example:
      let greet = () => {
        console.log("Greet! by Arrow function.");
      };

      //Arrow Function
      let a = (a, b) => {
        return a + b;
      };
      //Arrow Function
      let Data = (a, b) => console.log(a + b + " By arrow functin.");
      Data("Khushal", "Savalakaha");
      console.log(greet());
    </script>
  </body>
</html>
```



```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      let num = [10, 20, 35,10, 19, 25, 60];
      let a = num.filter((val, ind, arr) => {
        if (val >= 10 && val <= 25) {
          return val;
        }
      });

      let b = num.find((val, ind, arr) => {
        if (val==10) {
          return val;
        }
      });

      let c = num.findLastIndex((val, ind, arr) => {
        return val==10;
      });

      console.log(a);
      console.log(b);
      console.log(c);
    </script>
  </body>
</html>
```

The [findLastIndex](file:///d%3A/Javascript/Practice/Return_Particular_Range_value.html#23%2C19-23%2C19) method in JavaScript is used to find the index of the last element in the array that satisfies the provided testing function. If no such element is found, it returns -1.

In your code, `num.findLastIndex((val, ind, arr) => { return val==10; });` is looking for the last index of the element in the `num` array that is equal to 10.

Here's a breakdown of how it works:

1. The `findLastIndex` method iterates over each element in the `num` array, starting from the last element and moving towards the first.
2. For each element, it calls the provided function `(val, ind, arr) => { return val==10; }`.
   - `val` is the current element being processed.
   - `ind` is the index of the current element.
   - `arr` is the array on which `findLastIndex` was called.
3. The function checks if the current element `val` is equal to 10. If it is, the function returns `true`, which means the `findLastIndex` method will stop iterating and return the index of the current element.
4. If the function never returns `true` (i.e., no element in the array is equal to 10), `findLastIndex` will return -1.

In your case, `c` will be the index of the last element in the `num` array that is equal to 10. If there is no such element, `c` will be -1.

it means here (val, ind, arr) => {


Khushal Savalakha

return val==10;

}

returns true for particular value to findLastIndex() and it will find index from  array num am i true?

Yes, you're correct. The function `(val, ind, arr) => { return val==10; }` is a testing function that is passed to the `findLastIndex` method. This function is called for each element in the `num` array.

- `val` is the current element being processed.
- `ind` is the index of the current element.
- `arr` is the array on which `findLastIndex` was called.

The function checks if the current element `val` is equal to 10. If it is, the function returns `true`, which means the `findLastIndex` method will stop iterating and return the index of the current element.

So, if the array `num` is `[10, 20, 35, 10, 19, 25, 60]`, the `findLastIndex` method will return the index of the last element that is equal to 10. In this case, the last element that is equal to 10 is at index 3 (arrays are 0-indexed), so `findLastIndex` will return 3.

but here arr.forEach(element=>{

document.write(element+"<br>");

})

in this there is no particular format like (value,index,array)?

The [forEach](file:///d%3A/Javascript/Practice/Loops.html#28%2C11-28%2C11) method in JavaScript can take up to three arguments: the current value being processed, the index of that value in the array, and the array itself. However, it's not necessary to include all of these arguments in the function you pass to [forEach](file:///d%3A/Javascript/Practice/Loops.html#28%2C11-28%2C11).

In your code, you're only using the first argument, which is the current value being processed. This is why you're only seeing [element](file:///d%3A/Javascript/Practice/Loops.html#28%2C19-28%2C19) in the function definition.

Here's how you could modify your code to include the index and array arguments:

```
let str='10,20,30';
let arr=str.split(',');
arr.forEach((element, index, array) => {
  document.write(element + " at index " + index + " in array " + array + "<br>");
})
```

In this version of the code, the function passed to `forEach` takes three arguments: `element`, `index`, and `array`. The function then writes the current element, its index, and the entire array to the document.

here it means we can use element without braces ?

Yes, in JavaScript, if a function takes only one argument, you can omit the parentheses around the argument list. This is called "implicit parameter syntax".

So, in your [forEach](file:///d%3A/Javascript/Practice/Loops.html#28%2C11-28%2C11) loop, you can use [element](file:///d%3A/Javascript/Practice/Loops.html#28%2C19-28%2C19) without parentheses because the arrow function only has one parameter. If you had more than one parameter, or if you wanted to use the `arguments` object, you would need to use parentheses.

Here's an example:

```
let arr = [1, 2, 3, 4, 5];

// Using implicit parameter syntax
arr.forEach(element => {
  console.log(element);
});

// Using explicit parameter syntax
arr.forEach((element, index) => {
```


Khushal Savalakha

```
  console.log(element, index);
});
```

In the first `forEach` loop, `element` is the only parameter, so you can use it without parentheses. In the second `forEach` loop, there are two parameters, so you need to use parentheses.

> Arrow Function Implicit Return

In JavaScript, arrow functions have a concise syntax that allows you to omit the `return` keyword in certain situations. When an arrow function consists of only one expression, you can omit the curly braces `{}` and the `return` keyword, and the result of that expression will be implicitly returned. This shorthand syntax can make your code more concise and readable.

Here's an example to illustrate:

```
// Regular function
function add(a, b) {
  return a + b;
}

// Arrow function with explicit return
const addArrowExplicit = (a, b) => {
  return a + b;
};

// Arrow function with implicit return
const addArrowImplicit = (a, b) => a + b;

console.log(add(2, 3)); // Output: 5
console.log(addArrowExplicit(2, 3)); // Output: 5
console.log(addArrowImplicit(2, 3)); // Output: 5
```

In the above example, `add` is a regular function that explicitly returns the sum of two numbers using the `return` keyword. Both `addArrowExplicit` and `addArrowImplicit` are arrow functions that achieve the same result of adding two numbers. However, `addArrowImplicit` omits the curly braces `{}` and the `return` keyword, relying on the implicit return of the expression `a + b`.

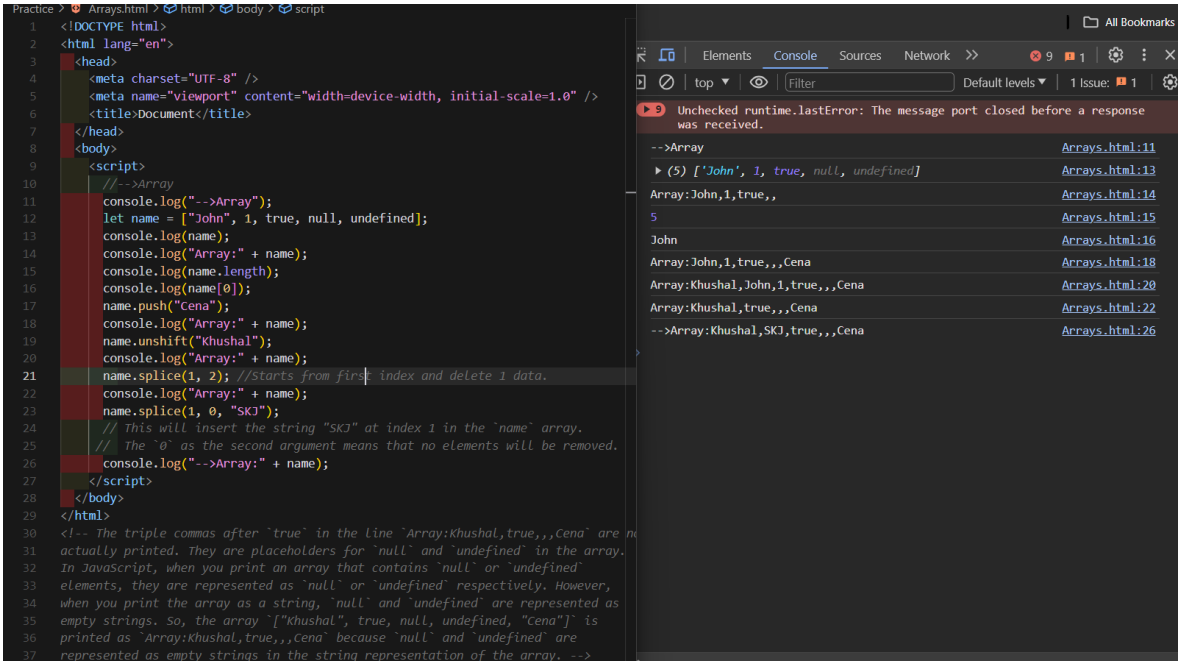
So, arrow functions provide a more concise way of writing functions, especially when the function body consists of a single expression. This shorthand syntax can make your code cleaner and easier to read.

## ▼ Arrays

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      //-->Array
      console.log("-->Array");
      let name = ["John", 1, true, null, undefined];
      console.log(name);
      console.log("Array:" + name);
      console.log(name.length);
      console.log(name[0]);
      name.push("Cena");
      console.log("Array:" + name);
      name.unshift("Khushal");
```


Khushal Savalakha

```
      console.log("Array:" + name);
      name.splice(1, 2); //Starts from first index and delete 1 data.
      console.log("Array:" + name);
      name.splice(1, 0, "SKJ");
      // This will insert the string "SKJ" at index 1 in the `name` array.
     //  The `0` as the second argument means that no elements will be removed.
      console.log("-->Array:" + name);
    </script>
  </body>
</html>
<!-- The triple commas after `true` in the line `Array:Khushal,true,,,Cena` are not
actually printed. They are placeholders for `null` and `undefined` in the array.
In JavaScript, when you print an array that contains `null` or `undefined`
elements, they are represented as `null` or `undefined` respectively. However,
when you print the array as a string, `null` and `undefined` are represented as
empty strings. So, the array `["Khushal", true, null, undefined, "Cena"]` is
printed as `Array:Khushal,true,,,Cena` because `null` and `undefined` are
represented as empty strings in the string representation of the array. -->
```




## ▼ Mutidimension Array

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      let arr = [
        ["john", 30], //0
        ["Thomas", 28], //1
        ["Robert", 22],
      ]; //2
```

```
      console.log(arr);
      console.log(arr[1]);
      console.log(arr[1][0]);
      console.log("------------------->Simple For loop");
      for (let i = 0; i < arr.length; i++) {
        for (let j = 0; j < arr[i].length; j++) {
          console.log(arr[i][j]);
        }
      }
      console.log("------------------->For Each Nested loop");
      arr.forEach((data) => {
        data.forEach((val) => {
          console.log(val);
        });
      });
      console.log("------------------->For For Of Nested loop");
      for(ar of arr){
        for(i of ar){
            console.log(i);
        }
      }
    </script>
  </body>
</html>
```

```
▼ (3) [Array(2), Array(2), Array(2)]                                   Multidimension Array.html:15
  ▶ 0: (2) ['john', 30]
  ▶ 1: (2) ['Thomas', 28]
  ▶ 2: (2) ['Robert', 22]
    length: 3
  ▶ [[Prototype]]: Array(0)
▼ (2) ['Thomas', 28]                                                   Multidimension Array.html:16
    0: "Thomas"
    1: 28
    length: 2
  ▶ [[Prototype]]: Array(0)
Thomas                                                                 Multidimension Array.html:17
------------------->Simple For loop                                    Multidimension Array.html:18
john                                                                   Multidimension Array.html:21
30                                                                     Multidimension Array.html:21
Thomas                                                                 Multidimension Array.html:21
28                                                                     Multidimension Array.html:21
Robert                                                                 Multidimension Array.html:21
22                                                                     Multidimension Array.html:21
------------------->For Each Nested loop                               Multidimension Array.html:24
john                                                                   Multidimension Array.html:27
30                                                                     Multidimension Array.html:27
Thomas                                                                 Multidimension Array.html:27
28                                                                     Multidimension Array.html:27
Robert                                                                 Multidimension Array.html:27
22                                                                     Multidimension Array.html:27
------------------->For For Of Nested loop                             Multidimension Array.html:30
john                                                                   Multidimension Array.html:33
30                                                                     Multidimension Array.html:33
Thomas                                                                 Multidimension Array.html:33
28                                                                     Multidimension Array.html:33
Robert                                                                 Multidimension Array.html:33
22                                                                     Multidimension Array.html:33
>
```

## ▼ Array Destructing

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        let array1=["Thomas","Chad","Neo","Magnus"];
        // let item1=array1[0];
        // let item2=array1[1];
        // console.log(item1);
        // console.log(item2);
        //It's not proper practice to print two elements of array1
        let [item1,item2,...item3]=array1;
        console.log(item1);
        console.log(item2);
        console.log(item3);
    </script>
</body>
</html>
```

```
Thomas                                  Array_distructing.html:17
Chad                                    Array_distructing.html:18
▶ (2) ['Neo', 'Magnus']                 Array_distructing.html:19
>
```

# ▼ Loops

## ▼ Foreach loop

The [forEach](file:///d%3A/Javascript/Practice/Loops.html#11%2C13-11%2C13) loop in JavaScript is a method that executes a provided function once for each array element. Here's the syntax:

```
array.forEach(function(currentValue, index, arr), thisValue)
```

- `currentValue`: The current element being processed in the array.
- `index` (optional): The index of the current element being processed in the array.
- `arr` (optional): The array `forEach()` was called upon.
- `thisValue` (optional): A value to use as `this` when executing the callback.

In your code, you're using an arrow function, which is a more concise way to write a function. The equivalent code using a regular function would be:

```
let num=[10,20,30,40,50];
num.forEach(function(val,index,array) {
    console.log(val+" "+index+" "+num);
})
```

This code will log each element of the `num` array, along with its index and the entire `num` array, to the console.

Let's break down the code snippet you provided and explain the usage of the `thisValue` parameter in the `forEach` method:


Khushal Savalakha

```
let numbers = [1, 2, 3, 4, 5];
let sum = 0;

numbers.forEach(function(number) {
    sum += number;
}, this);

console.log(sum); // Output: 15
```

1. **Array Declaration:**
   - An array named `numbers` is declared and initialized with some numeric values `[1, 2, 3, 4, 5]`.
2. **Variable Initialization:**
   - A variable named `sum` is declared and initialized to `0`. This variable will store the sum of the numbers in the `numbers` array.
3. `forEach` **Method:**
   - The `forEach` method is called on the `numbers` array.
   - It takes a callback function as its first argument. This callback function will be executed for each element of the array.
   - In this case, the callback function takes one argument `number`, which represents each element of the array as the loop iterates.
   - Inside the callback function, each `number` is added to the `sum` variable.
4. `thisValue` **Parameter:**
   - The second argument passed to the `forEach` method is `this`. This is the `thisValue` parameter.
   - In JavaScript, functions are objects, and they have their own `this` context. The `thisValue` parameter allows you to explicitly set the `this` value within the callback function.
   - In this example, `this` refers to the global object because we've passed `this` as the second argument to `forEach`. In a browser environment, the global object is `window`.
   - By passing `this` as the second argument, we are explicitly setting `this` to the global object within the callback function.
5. **Summation:**
   - As the `forEach` loop iterates through each element of the `numbers` array, the `number` is added to the `sum` variable.
6. **Output:**
   - After the `forEach` loop completes, the value of `sum` is logged to the console, which should be `15` in this case (sum of `1 + 2 + 3 + 4 + 5`).

In summary, the usage of `thisValue` in this example doesn't seem necessary because we're not using `this` inside the callback function. Passing `this` as the second argument to `forEach` is effectively the same as not passing any argument (which would also default `this` to the global object). However, it's good to understand how the `thisValue` parameter works in the `forEach` method for cases where you might need to explicitly set the `this` context within the callback function.

## ▼ For Of loop

**Example:**

```
const a = ["a", "b", "c"];
for (const [i, e] of a.entries()) {
  console.log(i, e);
}
```

**Output:**

```
0 'a'
1 'b'
2 'c'
```


Khushal Savalakha

Untitled

Page 18

```
<body>
  <script>
    obj = { a: "hello", b: "Good", c: "Morning" };
    for (let k in obj) {
      document.write(`the key is ${k}`+"<br>");
    }

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

the key is a
the key is b
the key is c

```
<script>
      obj = { a: "hello", b: "Good", c: "Morning" };
      for (let k in obj) {
        document.write(`the key is ${k}`+"<br>");
      }
      for (let k of obj) {
        document.write(`the key is ${k}`+"<br>");
      }
    </script>
```

In your code, you have an object `obj` with properties `a`, `b`, and `c`. You're trying to iterate over these properties using a `for...in` loop and a `for...of` loop. However, there are some issues with your code:

1. `for...in` loop is used to iterate over the properties of an object.
2. `for...of` loop is used to iterate over the values of an iterable object like arrays.

Let's correct your code:

```
let obj = { a: "hello", b: "Good", c: "Morning" };

// Using for...in loop to iterate over object properties
for (let k in obj) {
  document.write(`the key is ${k}`+"<br>");
}

// Using Object.values() to get an array of object values and then using for...of loo
p
for (let val of Object.values(obj)) {
  document.write(`the value is ${val}`+"<br>");
}
```

In this corrected version:

- The first loop iterates over the properties of the `obj` object using `for...in` loop, printing out each property key.
- The second loop uses `Object.values(obj)` to get an array of object values and then iterates over these values using a `for...of` loop, printing out each value.

In JavaScript, when we refer to the "properties" of an object in the context of a `for...in` loop, we are referring to the names (keys) of the enumerable properties of the object itself and, potentially, its prototype chain. Here's what is included:

1. **Own Properties**: These are the properties that are directly defined on the object itself.


Khushal Savalakha

2. **Inherited Properties**: If the object's prototype chain has additional enumerable properties, those properties will also be included. These properties are inherited from the object's prototype.

For example:

```
function Parent() {
  this.parentProp = 'parentProp';
}

Parent.prototype.parentProtoProp = 'parentProtoProp';

function Child() {
  this.childProp = 'childProp';
}

Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;
Child.prototype.childProtoProp = 'childProtoProp';

let childObj = new Child();

for (let key in childObj) {
  console.log(key);
}
```

In this example, iterating over `childObj` using `for...in` loop will include the following properties:

- `childProp`: An own property of `childObj`.
- `parentProp`: An inherited property from `Parent`.
- `parentProtoProp`: An inherited property from `Parent.prototype`.
- `childProtoProp`: An inherited property from `Child.prototype`.

`for...in` loop iterates over all enumerable properties, including both own properties and inherited properties from the prototype chain. If you only want to iterate over the own properties of an object, you can use `Object.hasOwnProperty()` method within the loop to filter out inherited properties.

# ▼ Spread and Copy

## ▼ Slice

The method in JavaScript is used to extract a section of an array and returns a new array. It does not modify the original array, but instead returns a shallow copy of elements from the original array.

The method takes two arguments:

1. The starting index (inclusive). If the start index is not specified, the default is 0.
2. The ending index (exclusive). If the end index is not specified, the default is the length of the array.

Here's a simple example:

```
let fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
let citrus = fruits.slice(1, 3);

// The slice() method returns a new array with the elements 'Orange' and 'Lemon'
// The original array is not modified
console.log(citrus); // Output: ['Orange', 'Lemon']
console.log(fruits); // Output: ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango']
```

In the example you provided, `arr4=arr3.slice(0);` creates a shallow copy of `arr3` and assigns it to `arr4`. This means that `arr4` is a new array with the same elements as `arr3`, but it is a separate array. If you modify `arr3` or `arr4`, the other array will not be affected.


Khushal Savalakha

The basic syntax of the method in JavaScript is as follows:

```
array.slice(start, end)
```

Here's what each part of the syntax means:

- `array`: The array you want to extract a section from.
- `start`: The index at which to begin the extraction. If `start` is undefined, `0` is used. If `start` is negative, it is treated as `array.length + start`. (For example, if `start` is `3`, it is treated as `array.length - 3`.)
- `end`: The index at which to end the extraction. `slice()` extracts up to but not including `end`. If `end` is omitted, `slice()` extracts through the end of the sequence. If `end` is negative, it is treated as `array.length + end`. (For example, if `end` is `2`, it is treated as `array.length - 2`.)

The `slice()` method does not modify the original array. It returns a new array with the extracted elements.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      arr1 = ["john", 202, true];
      arr2 = arr1;
      arr1.push("Berg");
      console.log(arr1);
      console.log(arr2);
      console.log("----------->");
      arr3 = ["skj", 768, true];
      arr4 = arr3.slice(0);
      arr3.push("Savalakha");
      console.log(arr3);
      console.log(arr4);
      console.log("----------->Spread Operator");
      //Spread Operator.
      arr4 = ["skj", 768, true];
      arr5 = [...arr4];
      arr4.push("Savalakha");
      console.log(arr4);
      console.log(arr5);
    </script>
  </body>
</html>
```

```
▶ (4) ['john', 202, true, 'Berg']          Spread_Copy.html:13
▶ (4) ['john', 202, true, 'Berg']          Spread_Copy.html:14
----------->                               Spread_Copy.html:15
▶ (4) ['skj', 768, true, 'Savalakha']      Spread_Copy.html:19
▶ (3) ['skj', 768, true]                   Spread_Copy.html:20
----------->Spread Operator                 Spread_Copy.html:21
▶ (4) ['skj', 768, true, 'Savalakha']      Spread_Copy.html:26
▶ (3) ['skj', 768, true]                   Spread_Copy.html:27
```


Khushal Savalakha

## ▼ Objects

**Just for the understanding**

The .entries() method returns an array containing arrays of an object's key-value pairs in the following format: [ [key, value], [key, value], ... ].

```
const object = { name: 'abc', age: 42 };
console.log(Object.entries(object));
```

**Output:** [[name, 'abc'],[ age, 42]]

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      let obj = {
        name: "Bravo",
        age: 21,
        "last name": "John",
        fun: function () {
          console.log("I am Function.");
        }
      };

      console.log(obj.name);
      console.log(obj.age);
      console.log(obj["name"]);
      console.log(obj["age"]);
      console.log(obj["last name"]);
      console.log(obj.fun());
      // console.log(obj["fun"]);This line gives error
      console.log(obj.fun.call(obj));
    </script>
  </body>
</html>
```

Output:

```
Bravo
21
Bravo
21
John
I am Function.
undefined
I am Function.
undefined
why here in the output
```


Khushal Savalakha

The `undefined` in the output is due to the `console.log(obj.fun());` line in your JavaScript code.

In JavaScript, when a function is called without the `return` statement, it implicitly returns `undefined`. This is why you see `undefined` in the output after "I am Function.".

If you want to see the output of the function, you should remove the `console.log(obj.fun());` line or replace it with `obj.fun();`.

why i have to write key ?

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      let obj = {
        name: "Bravo",
        age: 21,
        "last name": "John",
        fun: function () {
          console.log("I am Function.");
          return "Function executed successfully.";
        },
        arr:[20,30,40]
      };

      console.log(obj.name);
      console.log(obj.age);
      console.log(obj["name"]);
      console.log(obj["age"]);
      console.log(obj["last name"]);
      console.log(obj.fun());
      // console.log(obj["fun"]);This line gives error
    //   console.log(obj.fun.call(obj));
      obj.fun();
      console.log(obj.arr);
      console.log(obj);
      console.log(obj.fun);
      //obj.a="item1";it gives error
      obj.key='item1';
      console.log(obj.key);
      obj.key="item2";
      console.log(obj);
      obj[a]="item2";
      console.log(obj);
    </script>
  </body>
</html>
```

The error you're seeing is because you're trying to access a property [key](

In JavaScript, you can access object properties using either dot notation (like `obj.name`) or bracket notation (like `obj['name']`). The bracket notation is useful when the property name is dynamic or when it's a reserved word.

In your case, you're trying to access `obj.key`, but [key](file:///d%3A/Javascript/Practice/Objects.html#34%2C23-34%2C23) is not a property of the [obj](file:///d%3A/Javascript/Practice/Objects.html#10%2C11-10%2C11) object. That's why you're getting `undefined`.

If you want to add a new property to the [obj](file:///d%3A/Javascript/Practice/Objects.html#10%2C11-10%2C11) object, you can do it like this:


Khushal Savalakha

```
obj.key = "value";
```

This will add a new property `key` to the `obj` object with the value `"value"`.

In JavaScript, when you want to access a property of an object, you can use either dot notation or bracket notation.

In your case, you're trying to access a property that doesn't exist on the [obj](file:///d%3A/Javascript/Practice/Objects.html#10%2C11-10%2C11) object. If you want to add a new property to the [obj](file:///d%3A/Javascript/Practice/Objects.html#10%2C11-10%2C11) object, you can do it like this:

```
obj.key = "value";
```

This line of code is saying "add a new property to the `obj` object with the name `key` and the value `"value"`.

If you want to add a property with a dynamic name, you can use bracket notation:

```
let propertyName = "key";
obj[propertyName] = "value";
```

This will add a new property to the `obj` object with the name stored in the `propertyName` variable and the value `"value"`.

there is no any way to do naming by dot operator

In JavaScript, the dot operator is used to access properties of an object, not to create new properties. If you want to create a new property with a dynamic name, you have to use bracket notation.

The reason for this is that the dot operator is used for static property names, while the bracket notation is used for dynamic property names.

For example, if you know the property name in advance, you can use the dot operator:

```
obj.name = "John";
```

But if the property name is stored in a variable, you have to use the bracket notation:

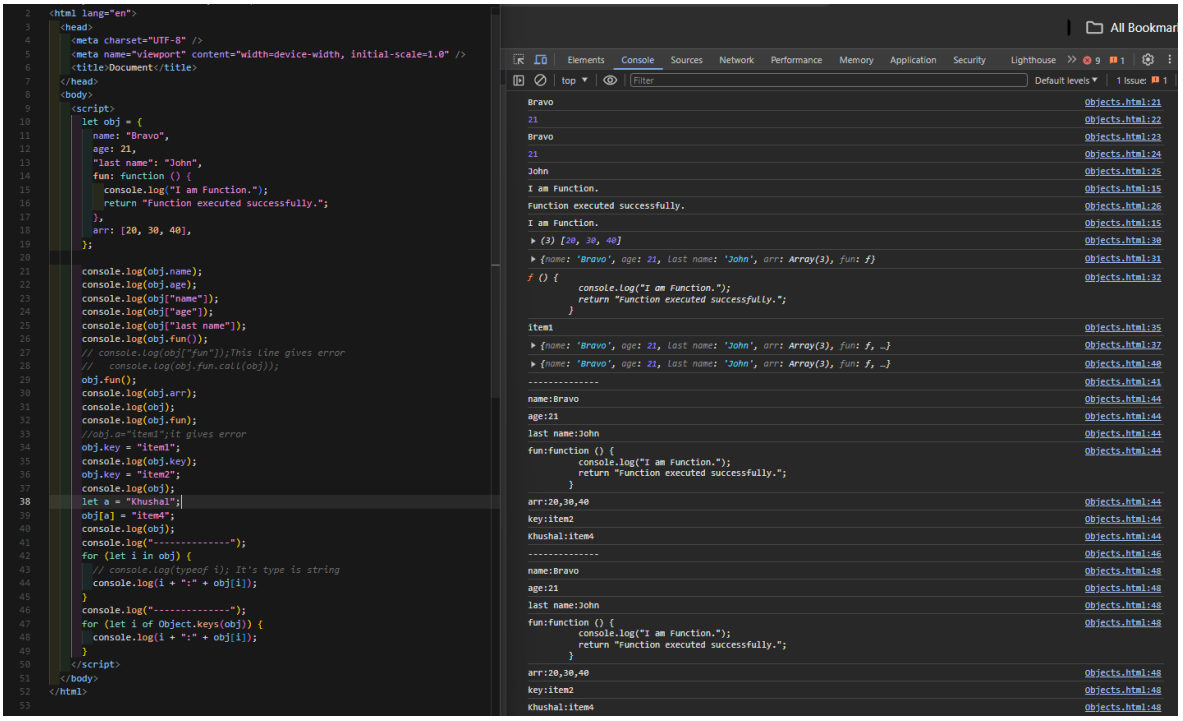
```
let propertyName = "name";
obj[propertyName] = "John";
```

This is because the dot operator expects a static property name, not a variable.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      let obj = {
        name: "Bravo",
        age: 21,
        "last name": "John",
        fun: function () {
          console.log("I am Function.");
          return "Function executed successfully.";
        },
        arr: [20, 30, 40],
      };

      console.log(obj.name);
      console.log(obj.age);
```


Khushal Savalakha

```
      console.log(obj["name"]);
      console.log(obj["age"]);
      console.log(obj["last name"]);
      console.log(obj.fun());
      // console.log(obj["fun"]);This line gives error
      //   console.log(obj.fun.call(obj));
      obj.fun();
      console.log(obj.arr);
      console.log(obj);
      console.log(obj.fun);
      //obj.a="item1";it gives error
      obj.key = "item1";
      console.log(obj.key);
      obj.key = "item2";
      console.log(obj);
      let a = "Khushal";
      obj[a] = "item4";
      console.log(obj);
      console.log("--------------");
      for (let i in obj) {
        // console.log(typeof i); It's type is string
        console.log(i + ":" + obj[i]);
      }
      console.log("--------------");
      for (let i of Object.keys(obj)) {
        console.log(i + ":" + obj[i]);
      }
    </script>
  </body>
</html>
```



**Example : Write Express JS script to request server to display json object (Array of Objects) in table form on browser.**

```
const expr=require("express")
const app=expr();
student={
   u1:[{name:"LJU",id:2},
   {name:"LJU1",id:3},
   {name:"LJU2",id:4},
   {name:"LJU3",id:5},
   {name:"LJU4",id:6}]}
      app.get("/student",(req,res)=>{
        res.set("content-type","text/html")
         res.write("<center><table cellspacing='5px' cellpadding='8px'
border='1px solid'><tr><th>Name</th><th>ID</th></tr>")
         for(i of student.u1){
            res.write("<tr><td>" + i.name + "</td>")
            res.write("<td>" + JSON.stringify(i.id) + "</td></tr>")
         }
        res.write("</table></center>")
        res.send()
      })
app.listen(6007)
```

Output:

| Name | ID |
|---|---|
| LJU | 2 |
| LJU1 | 3 |
| LJU2 | 4 |
| LJU3 | 5 |
| LJU4 | 6 |

Here it is fetching object directly that's why we have to write i.name

```
student.u1.forEach((item) => {
        res.write(`
            <tr>
                <td>${item.name}</td>
                <td>${item.id}</td>
            </tr>
        `);
        console.log(item);
    });
```

```
for (let k,v in obj) {
  // console.log(typeof i); It's type is string
  console.log(i + ":" + obj[i]);
}
```

here if we want to take key and value form obj ?

In JavaScript, you can't directly iterate over an object's properties using a `for...in` loop and get both the key and value at the same time. The `for...in` loop only gives you the key, not the value.

However, you can use the `Object.entries()` method to get an array of key-value pairs from an object, and then you can use a `for...of` loop to iterate over this array. Here's how you can do it:

```
for (let [key, value] of Object.entries(obj)) {
  console.log(key + ":" + value);
}
```

In this code, `Object.entries(obj)` returns an array of key-value pairs, and the `for...of` loop iterates over this array. The `key` and `value` variables are used to get the key and value of each pair.

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


Khushal Savalakha

```
    <title>Document</title>
  </head>
  <body>
    <script>
      // entries() method
      const a = ["a", "b", "c"];
      for (const [i, e] of a.entries()) {
        document.write(i, e + "<br>");
      }

      let obj = { a: "hello", b: "Good", c: "Morning" };

      for (const [key, value] of Object.entries(obj)) {
        document.write(`${key}: ${value}<br>`);
      }
    </script>
  </body>
</html>
Output:
0a
1b
2c
a: hello
b: Good
c: Morning
```

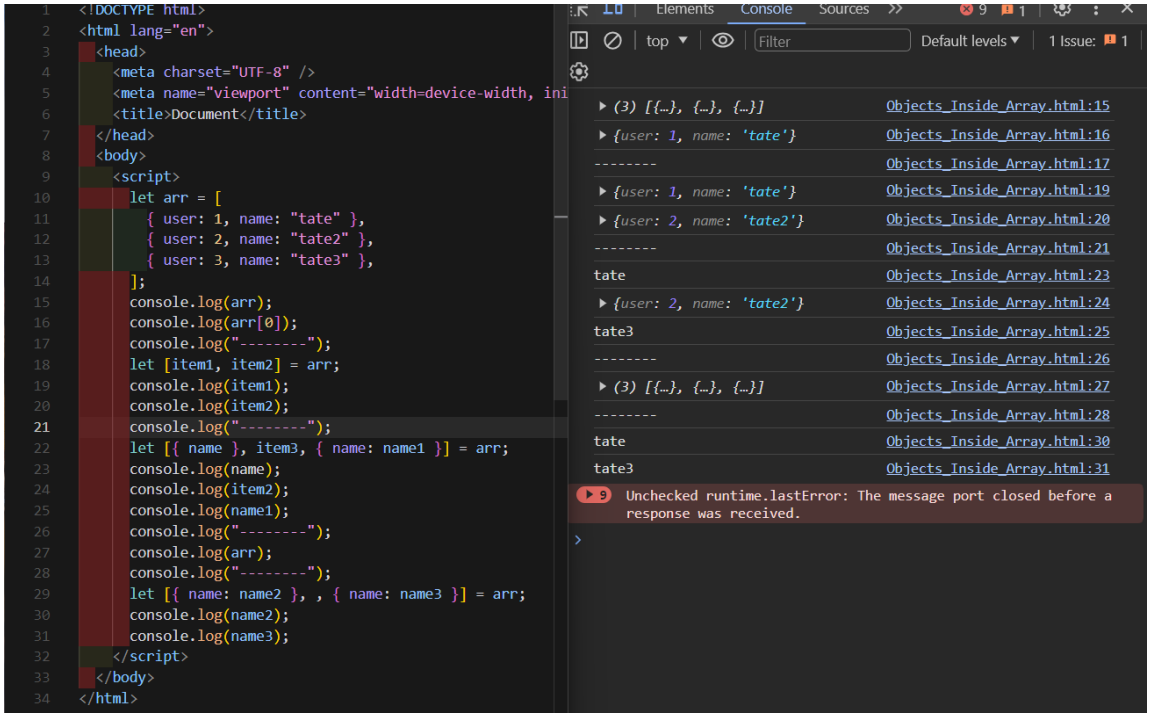
here we also applies object destructuring.

## ▼ Objects Inside Array

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      let arr = [
        { user: 1, name: "tate" },
        { user: 2, name: "tate2" },
        { user: 3, name: "tate3" },
      ];
      console.log(arr);
      console.log(arr[0]);
      console.log("--------");
      let [item1, item2] = arr;
      console.log(item1);
      console.log(item2);
      console.log("--------");
      let [{ name }, item3, { name: name1 }] = arr;
      console.log(name);
      console.log(item2);
      console.log(name1);
      console.log("--------");
      console.log(arr);
      console.log("--------");
```


Khushal Savalakha

```
      let [{ name: name2 }, , { name: name3 }] = arr;
      console.log(name2);
      console.log(name3);
    </script>
  </body>
</html>
```




The code you've provided is using JavaScript's array destructuring syntax. It's a way to extract data from arrays or objects into distinct variables.

Here's a breakdown of the line you're asking about:

```
let [{ name }, item3, { name: name1 }] = arr;
```

This line is declaring three new variables: `name`, `item3`, and `name1`. It's also using array destructuring to assign values to these variables from the `arr` array.

- `let [{ name }, item3, { name: name1 }]`: This part is using array destructuring to assign the first object's `name` property to the `name` variable, the third item in the array to the `item3` variable, and the second object's `name` property to the `name1` variable.
- `= arr;`: This part is specifying that the array to be destructured is `arr`.

So, the line is equivalent to:

```
let name = arr[0].name;
let item3 = arr[1];
let name1 = arr[2].name;
```

If you want to write this in a more traditional way, you would do it like this:

```
let name = arr[0].name;
let item3 = arr[1];
let name1 = arr[2].name;
console.log(name);
```

This will log the `name` property of the first object in the `arr` array.


Khushal Savalakha

## ▼ Map

```
  // method1:
let map2 = new Map();
 map2.set("ABC","FACULTY");
 map2.set("XYZ","ADMIN");
 console.log(map2);

//  method2:
 let map1=new Map([["ABC","FACULTY"], ["XYZ","ADMIN"]])
console.log(map1);
```

**Output:**

```
{'ABC' => 'FACULTY', 'XYZ' => 'ADMIN'}
{'ABC' => 'FACULTY', 'XYZ' => 'ADMIN'}
```

### Checking size of the map

The size property can be used to determine the number of values stored in the map.

Syntax

```
map_name.size
```

**Example:**

```
<script>
  let daysMap = new Map();
  daysMap.set('1', 'Monday');
  daysMap.set('2', 'Tuesday');
  daysMap.set('3', 'Wednesday');
  console.log(daysMap.size);
</script>
```

**Output:**

3

**Example**

```
let map=new Map();
map.set("ABC","FACULTY");
map.set("JKL","ADMIN");
console.log(map);
```



```
//Map(2) {'ABC' => 'FACULTY', 'JKL' => 'ADMIN'} =

map.delete("JKL"); //delete key "JKL"
console.log(map);
//Map(1) {'ABC' => 'FACULTY'}

map.set("Abc","Student"); //add key/value with key "Abc"
console.log(map);
// Map(2) {'ABC' => 'FACULTY', 'Abc' => 'Student'}

map.set("ABC","HOD"); //modify value of key "ABC"
console.log(map);
// Map(2) {'ABC' => 'HOD', 'Abc' => 'Student'}


map.clear(); // clears all elements
console.log(map);
// Map(0) {size: 0}

Output:
Map(2) {'ABC' => 'FACULTY', 'JKL' => 'ADMIN'}
Map(1) {'ABC' => 'FACULTY'}
Map(2) {'ABC' => 'FACULTY', 'Abc' => 'Student'}
Map(2) {'ABC' => 'HOD', 'Abc' => 'Student'}
Map(0) {size: 0}
```

# ▼ Destructuring of Object and Array

## ▼ Object

Object destructuring is a feature in programming languages like JavaScript and Python that allows you to extract values from objects and assign them to variables in a single statement.

In JavaScript, object destructuring uses a syntax similar to array destructuring, but with curly braces `{}`. Here's an example:

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};

// Using object destructuring
const { name, age, city } = person;

console.log(name); // Output: John
console.log(age); // Output: 30
console.log(city); // Output: New York
```

In this example, `{ name, age, city }` is the object destructuring syntax, and it assigns the value of the `name` property to the variable `name`, the value of the `age` property to the variable `age`, and so on.


Khushal Savalakha

Similarly, in Python, object destructuring is achieved using a similar syntax. Here's an example:

```
person = {
  'name': 'John',
  'age': 30,
  'city': 'New York'
}

# Using object destructuring
name, age, city = person.values()

print(name)  # Output: John
print(age)  # Output: 30
print(city)  # Output: New York
```

In Python, you can use methods like `.values()` to extract the values from the dictionary and perform destructuring. The order of assignment in object destructuring is based on the order of properties in the object (or dictionary), not their names. However, in JavaScript, the order is based on the property names matching the variable names.

## ▼ Array

Array destructuring is a feature in programming languages like JavaScript, Python, and others, that allows you to unpack values from arrays or iterables into distinct variables.

In JavaScript, for example, array destructuring allows you to extract values from an array and assign them to variables in a single statement. Here's an example:

```
const myArray = [1, 2, 3];

// Using array destructuring
const [a, b, c] = myArray;

console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3
```

In this example, `[a, b, c]` is the destructuring syntax, and it assigns the first value of `myArray` to `a`, the second value to `b`, and so on. This can be very useful when working with functions that return arrays or when dealing with structured data.

Similarly, in Python, array destructuring is achieved using a similar syntax. Here's an example:

```
my_list = [1, 2, 3]

# Using array destructuring
a, b, c = my_list

print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

Array destructuring can make code more concise and readable, especially when working with arrays or iterables with known structures.

Certainly! Let's break down the JavaScript code within the HTML `<script>` tag and explain the output:

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


Khushal Savalakha

```
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
</head>
<body>
  <script>
    // Array of objects
    let arr = [
      { user: 1, name: "tate" },
      { user: 2, name: "tate2" },
      { user: 3, name: "tate3" },
    ];
    console.log(arr); // Output: [{ user: 1, name: "tate" }, { user: 2, name: "tate
2" }, { user: 3, name: "tate3" }]
    console.log(arr[0]); // Output: { user: 1, name: "tate" }
    console.log("--------");

    // Destructuring assignment with array
    let [item1, item2] = arr;
    console.log(item1); // Output: { user: 1, name: "tate" }
    console.log(item2); // Output: { user: 2, name: "tate2" }
    console.log("--------");

    // Destructuring assignment with objects inside array
    let [{ name }, item3, { name: name1 }] = arr;
    console.log(name); // Output: "tate"
    console.log(item3); // Output: { user: 2, name: "tate2" }
    console.log(name1); // Output: "tate3"
    console.log("--------");

    console.log(arr); // Output: [{ user: 1, name: "tate" }, { user: 2, name: "tate
2" }, { user: 3, name: "tate3" }]
    console.log("--------");

    // Destructuring assignment with skipping elements
    let [{ name: name2 }, , { name: name3 }] = arr;
    console.log(name2); // Output: "tate"
    console.log(name3); // Output: "tate3"
  </script>
</body>
</html>
```

Explanation and Output:

- The `arr` variable is an array of objects with three elements.
- `console.log(arr)` prints the entire array.
- `console.log(arr[0])` prints the first element of the array.
- `console.log(item1)` and `console.log(item2)` after destructuring assignment prints the first and second elements of the array respectively.
- The destructuring assignment `{ name }, item3, { name: name1 }` assigns the `name` property of the first object to `name`, the second object to `item3`, and the `name` property of the third object to `name1`.
- `console.log(arr)` after destructuring prints the entire array again.
- The destructuring assignment `{ name: name2 }, , { name: name3 }` assigns the `name` property of the first and third objects to `name2` and `name3` respectively, skipping the second object.
- Each `console.log()` statement outputs the values described in the comments.


Khushal Savalakha

{ name } here how we decide to store in name ?

In the destructuring assignment `{ name }`, the decision to store the value in the variable `name` is based on the property name within the object being destructured. Let's break it down:

- `{ name }`: This syntax is an object destructuring pattern.
- Within the curly braces `{}`, we specify the property name (`name` in this case) that we want to extract from the object.

When you use this syntax in a destructuring assignment, JavaScript looks for a property named `name` in the object being destructured. It then extracts the value of that property and assigns it to a variable with the same name (`name` in this case).

For example, consider the object:

```
let obj = { name: "John", age: 30 };
```

If we use the destructuring assignment `{ name } = obj;`, JavaScript will look for the `name` property in the `obj` object and extract its value. Since the `obj` object has a `name` property with the value `"John"`, that value will be assigned to the variable `name`.

So, in `{ name } = obj;`, the decision to store the value in `name` is based on the property name specified in the object being destructured. If the property name matches a variable name within the destructuring pattern, the value of that property will be assigned to that variable.

# ▼ REACT METHOD

it means here they take name as property and assigning value am i true ?

Yes, you are correct. In the destructuring assignment `{ name }`, `name` is indeed taken as the property name, and the corresponding value of that property in the object being destructured is assigned to a variable with the same name (`name` in this case).

Let's clarify with an example:

Suppose we have an object:

```
let obj = { name: "John", age: 30 };
```

If we use the destructuring assignment `{ name } = obj;`, JavaScript interprets this as follows:

- Look for a property named `name` in the object `obj`.
- Extract the value associated with the `name` property.
- Assign this value to a variable named `name`.

Since the `obj` object does have a `name` property with the value `"John"`, the value `"John"` will be assigned to the variable `name`.

So, `{ name } = obj;` does indeed involve taking `name` as a property and assigning its value to the variable `name`.

let [{ name }, item3, { name: name1 }] = arr;
here if i want to to find for only just value then let{name}=arr can work or not ?

Yes, if you only want to extract the value of the `name` property from the first and third objects in the `arr` array, you can use the destructuring assignment `{ name } = arr;`. This will work fine as long as the structure of the `arr` array remains consistent with the pattern you're destructuring.

Let's see an example:

```
let arr = [
  { user: 1, name: "John" },
  { user: 2, name: "Thomas" },
  { user: 3, name: "Robert" }
];

let { name } = arr[0]; // Extracts the value of the 'name' property from the fir
st object
console.log(name); // Output: "John"
```


Khushal Savalakha

In this example, `{ name } = arr[0];` extracts the value of the `name` property from the first object in the `arr` array, which is `"John"`, and assigns it to the variable `name`.

However, if you only want to extract the value of the `name` property from the first object and not assign the other elements of the array, you should use the array destructuring syntax instead of the object destructuring syntax. Here's how you can do it:

```
let [ { name } ] = arr; // Extracts the value of the 'name' property from the fi
rst object
console.log(name); // Output: "John"
```

In this example, `[ { name } ] = arr;` extracts the value of the `name` property from the first object in the `arr` array and assigns it to the variable `name`. This approach is more suitable if you're only interested in extracting a single property from the first object of the array.

No, that's not quite accurate. In JavaScript, the function parameter syntax `({ company })` is using object destructuring. It expects an object as an argument and extracts the `company` property from that object. This means that the function parameter is expected to be an object with a `company` property.

Let's clarify with an example:

```
const navbar = ({ company }) => {
  console.log(company);
};

navbar({ company: "hitesh" }); // Output: "hitesh"
```

In this example, the `navbar` function expects an object as an argument. When you call `navbar({ company: "hitesh" })`, you're passing an object with a `company` property set to `"hitesh"`. The function then extracts the `company` property from this object using object destructuring and logs its value to the console.

If you try to call the `navbar` function with a different type of argument, like a string or a number, it will result in an error because object destructuring expects an object as its argument. For example:

```
navbar("hitesh"); // TypeError: Cannot destructure property 'company' of 'undefi
ned' or 'null'.
```

In this case, JavaScript will attempt to destructure the string `"hitesh"` as an object, which is not possible, leading to a TypeError. Therefore, the parameter is not automatically converted into an object; it's just expecting an object with a specific structure.

## ▼ Difference Between This and event

Certainly, let's discuss the major differences in how they work:

1. **`onclick='fun(this)'`:**
   - When you use `this`, you're passing a reference to the element that triggered the event.
   - This allows you to directly manipulate properties or attributes of the element within the event handler function.
   - For example, you can change the style, content, or any other attribute of the element using `this` within the function.
2. **`onclick='fun(event)'`:**
   - When you use `event`, you're passing a reference to the event object itself.
   - This allows you to access information about the event itself, including details like the type of event, the target element, and any additional data related to the event.
   - You can use this information to perform more complex operations, such as determining which element triggered the event, handling keyboard or mouse input, preventing default behavior, or stopping event propagation.

In summary, the major difference lies in what information you want to work with inside the event handler function:

- Use `this` when you primarily need to manipulate properties or attributes of the element that triggered the event.
- Use `event` when you need access to information about the event itself, such as its type, target element, or additional data.


Khushal Savalakha

In the context of JavaScript and HTML, "attributes" refer to additional information or settings associated with an HTML element. These attributes are defined within the HTML tag and provide additional information about the element or specify its behavior.

For example, consider an HTML `<input>` element:

```
<input type="text" id="myInput" value="Hello">
```

In this `<input>` element:

- `type`, `id`, and `value` are attributes.
- `type` specifies the type of input (text, checkbox, etc.).
- `id` specifies a unique identifier for the element.
- `value` specifies the initial value of the input field.

When we say "manipulating attributes of an element" within an event handler function, it means changing or accessing these attribute values dynamically using JavaScript. For instance:

```
// Changing the value attribute of an input element
document.getElementById("myInput").value = "New Value";

// Accessing the id attribute of an element
var elementId = document.getElementById("myInput").id;
```

So, when I mentioned "manipulating properties or attributes of the element within the event handler function," it means you can dynamically change or access these attributes of the HTML element that triggered the event using JavaScript.

but here we can change the value by this like obj.value='ddsdsf' ?

Yes, you're absolutely correct. In JavaScript, when you have a reference to an HTML element, you can directly manipulate its properties, including attributes like `value`, using dot notation.

For example, if you have an input element with the id "myInput":

```
<input type="text" id="myInput" value="Initial Value">
```

You can change its value using JavaScript like this:

```
document.getElementById("myInput").value = "New Value";
```

In this case, `document.getElementById("myInput")` returns a reference to the input element, and `.value` allows you to directly set or get the value of the input element.

So, indeed, you can use `obj.value = 'new value';` to change the value attribute of an input element, where `obj` is a reference to the input element obtained, for example, by using `document.getElementById()` or any other method to obtain a reference to the element.

## ▼ Onsubmit

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

<form id="myForm" onsubmit="return validateForm()">
    <label for="fname">First Name:</label>
    <input type="text" id="fname" name="fname"><br><br>
    <label for="lname">Last Name:</label>
```


Khushal Savalakha

```
    <input type="text" id="lname" name="lname"><br><br>
    <input type="submit" value="Submit">
</form>

<script>
    function validateForm() {
        var fname = document.getElementById("fname").value;
        var lname = document.getElementById("lname").value;

        if (fname == "" || lname == "") {
            alert("Both first name and last name must be filled out");
            return false; // Prevent form submission
        }

        // If validation passes, you can allow form submission
        return true;
    }
</script>

</body>
</html>
```

## ▼ Validation

### JavaScript Validation

It is important to validate the form submitted by the user because it can have inappropriate values. So, validation is must to authenticate user.

JavaScript provides facility to validate the form on the client-side so data processing will be faster than server-side validation. Most of the web developers prefer JavaScript form validation.

Through JavaScript, we can validate name, password, email, date, mobile numbers and more fields.

### Ways to access form elements

obj = document.forms[“form_name”][“element_name”].value;

**or**

obj = document.form_name.element_name.value;

**or**

obj = document.getElementByID(id).value;

Sure, here are the proper examples for each method:

1. Using `document.forms`:

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


Khushal Savalakha

```
<form name="myForm">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username"><br><br>
    <input type="button" value="Submit" onclick="getValue()">
</form>

<script>
    function getValue() {
        var username = document.forms["myForm"]["username"].value;
        alert("Username: " + username);
    }
</script>

</body>
</html>
```

1. Using `document.form_name.element_name`:

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

<form name="myForm">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username"><br><br>
    <input type="button" value="Submit" onclick="getValue()">
</form>

<script>
    function getValue() {
        var username = document.myForm.username.value;
        alert("Username: " + username);
    }
</script>

</body>
</html>
```

1. Using `document.getElementById`:

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

<form>
    <label for="username">Username:</label>
```


Khushal Savalakha

```
    <input type="text" id="username" name="username"><br><br>
    <input type="button" value="Submit" onclick="getValue()">
</form>

<script>
    function getValue() {
        var username = document.getElementById("username").value;
        alert("Username: " + username);
    }
</script>

</body>
</html>
```

Each of these examples demonstrates how to retrieve the value of the "username" input field from a form using a different method.

## ▼ Set Attribute

Certainly! Here's a simple example demonstrating the usage of the `setAttribute()` method in JavaScript:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>setAttribute Example</title>
    <style>
        .highlight {
            background-color: yellow;
        }
    </style>
</head>
<body>

<div id="myDiv">Original content</div>
<button onclick="addAttributes()">Add Attributes</button>

<script>
    function addAttributes() {
        // Get the element by its ID
        var element = document.getElementById("myDiv");

        // Set the "class" attribute to "highlight"
        element.setAttribute("class", "highlight");

        // Set the "data-info" attribute to "Some additional info"
        element.setAttribute("data-info", "Some additional info");
    }
</script>

</body>
</html>
```

In this example:

- There's a `<div>` element with the ID "myDiv" and some initial content.
- There's also a `<button>` element with an `onclick` attribute that calls the `addAttributes()` function when clicked.
- Inside the `addAttributes()` function:
    - The `getElementById()` method retrieves the `<div>` element with the ID "myDiv".


Khushal Savalakha

- The `setAttribute()` method is then used to set two attributes on the `<div>` element:
  - The `class` attribute is set to "highlight", applying a CSS style that highlights the element.
  - The `data-info` attribute is set to "Some additional info", which could be used for additional data or information.

- When you click the button, the `addAttributes()` function is executed, and the specified attributes are added to the `<div>` element.

You can see the changes in the appearance of the `<div>` element after clicking the button, as the class "highlight" is applied to it. Additionally, you can inspect the element in the browser's developer tools to see the added attributes.

in which type of attribute we do this.

In JavaScript, the `setAttribute()` method can be used to set any attribute on an HTML element. This includes both standard attributes like `class`, `id`, `src`, `href`, `style`, etc., as well as custom attributes.

Here are a few examples of attributes that you might commonly set using `setAttribute()`:

1. Standard HTML attributes:
   - `class`: to assign CSS classes to an element.
   - `id`: to assign a unique identifier to an element.
   - `src`: to specify the URL of an image or script.
   - `href`: to specify the URL of a hyperlink.
   - `style`: to apply inline CSS styles to an element.
2. Custom data attributes:
   - `data-*`: to store custom data associated with an element. For example, `data-info`, `data-username`, etc.
3. Event attributes:
   - `onclick`, `onmouseover`, `onsubmit`, etc.: to attach JavaScript event handlers to elements.
4. Form-related attributes:
   - `name`, `value`, `type`, etc.: for form input elements.
5. Any other standard or custom attributes specific to your application or use case.

So, `setAttribute()` can be used with a wide range of attributes, depending on your requirements for manipulating the elements in the DOM (Document Object Model).

Certainly! Here's an example demonstrating how to use the `setAttribute()` method to set a style attribute on an HTML element:

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

<div id="myDiv">Original content</div>
<button onclick="addStyle()">Add Style</button>

<script>
    function addStyle() {
        // Get the element by its ID
        var element = document.getElementById("myDiv");

        // Set the "style" attribute to apply a background color and padding
        element.setAttribute("style", "background-color: lightblue; padding: 10px;");
    }
</script>
```


Khushal Savalakha

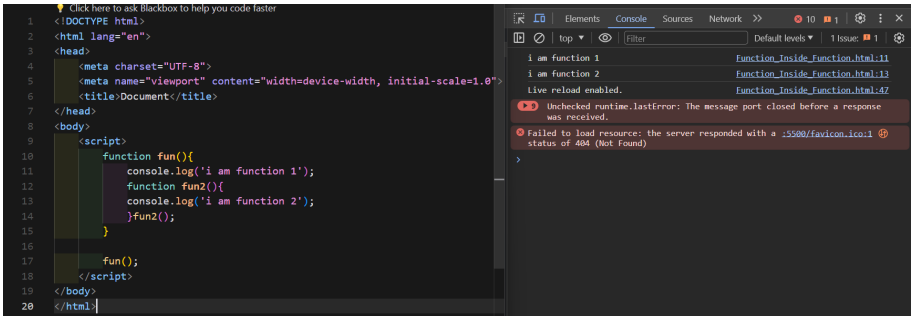
```
</body>
</html>
```

In this example:

- There's a `<div>` element with the ID "myDiv" and some initial content.
- There's also a `<button>` element with an `onclick` attribute that calls the `addStyle()` function when clicked.
- Inside the `addStyle()` function:
  - The `getElementById()` method retrieves the `<div>` element with the ID "myDiv".
  - The `setAttribute()` method is then used to set the `style` attribute on the `<div>` element. The value assigned to the `style` attribute is a string containing CSS styles. In this case, we're setting the background color to light blue and adding some padding to the element.

When you click the button, the `addStyle()` function is executed, and the specified styles are applied to the `<div>` element, changing its appearance accordingly.

## ▼ Function Inside Function

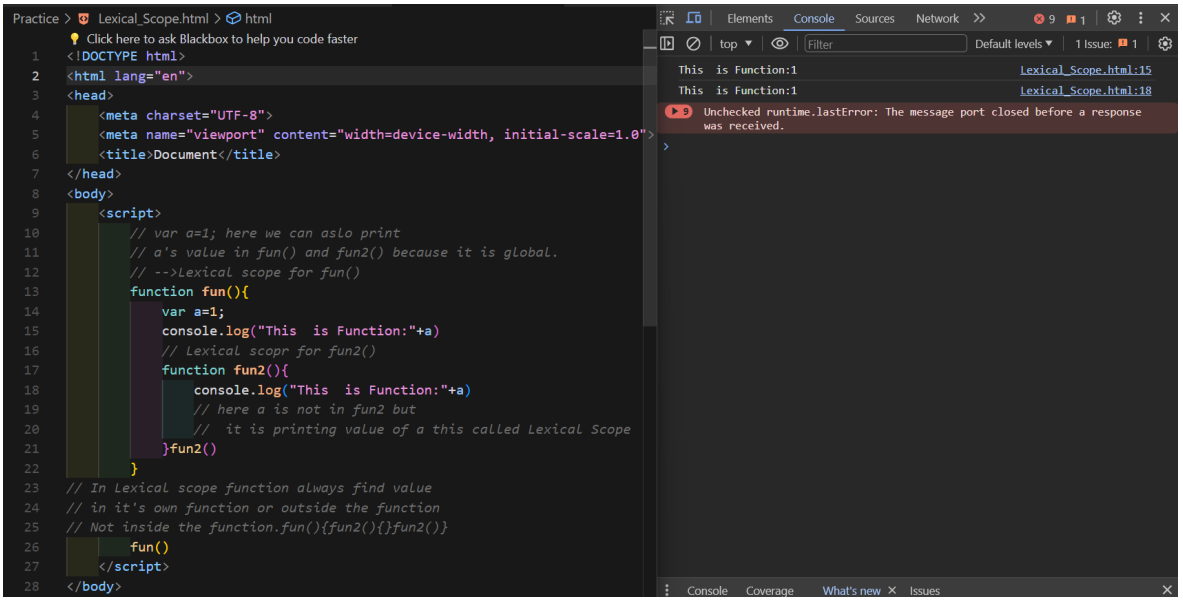



## ▼ Lexical Scope

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // var a=1; here we can aslo print
        // a's value in fun() and fun2() because it is global.
        // -->Lexical scope for fun()
        function fun(){
            var a=1;
            console.log("This  is Function:"+a)
            // Lexical scopr for fun2()
            function fun2(){
                console.log("This  is Function:"+a)
                // here a is not in fun2 but
                //  it is printing value of a this called Lexical Scope
            }fun2()
        }
// In Lexical scope function always find value
// in it's own function or outside the function
```


Khushal Savalakha

```
// Not inside the function.fun(){fun2(){}fun2()}
        fun()
    </script>
</body>
</html>
```




## ▼ Debugger

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        function fun(){
            debugger;
            for(let i=1;i<=5;i++){
                console.log(i);
            }
        }
        fun()
    </script>
</body>
</html>
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        function fun(){
            debugger;
            for(let i=1;i<=5;i++){
                console.log(i);
            }
        }
        fun()
    </script>
</body>
</html>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR

```
1
2
3
```

## ▼ Sets

```javascript
const arr=[10,20,30,40,50,40];
console.log(arr);

const s=new Set([10,20,30,40,50,40,50]);
console.log(s)
s.add('Hello');
console.log(s.length);
console.log(s);
length=0;
for(let element of s){
    length++;
}

console.log('set length:'+length)
```

```
Click here to ask Blackbox to help you code faster
const arr=[10,20,30,40,50,40];
console.log(arr);

const s=new Set([10,20,30,40,50,40,50]);
console.log(s)
s.add('Hello');
console.log(s.length);
console.log(s);
length=0;
for(let element of s){
    length++;
}

console.log('set length: '+length);
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
PS D:\Javascript> node "d:\Javascript\Practice\Sets.js"
[ 10, 20, 30, 40, 50, 40 ]
Set(5) { 10, 20, 30, 40, 50 }
undefined
Set(6) { 10, 20, 30, 40, 50, 'Hello' }
set length:6
PS D:\Javascript>
```

```java
const arr = [10, 20, 30, 40, 50, 40];
console.log(arr);

const s = new Set([10, 20, 30, 40, 50, 40, 50]);
console.log(s);

s.add('Hello');
console.log(s.size); // Use size instead of length

console.log(s);

let length = 0;
for (let element of s) {
    length++;
}

console.log('set length:' + length);
```

[ 10, 20, 30, 40, 50, 40 ]
Set(5) { 10, 20, 30, 40, 50 }
6
Set(6) { 10, 20, 30, 40, 50, 'Hello' }
set length:6

it means set is an object and object does not have length ?

Yes, that's correct. In JavaScript, Sets are not arrays; they are a different data structure. While arrays have a `length` property, which gives the number of elements in the array, Sets have a `size` property instead, which gives the number of elements in the Set.


Khushal Savalakha

Arrays are ordered collections of elements accessed by numerical indices, whereas Sets are collections of unique elements, and the order of elements is not guaranteed.

In JavaScript, both arrays and Sets are objects, but they have different methods and properties suited to their respective purposes.

## ▼ Maps

```
//Map
var map1=new Map([[1,"One"],["fname","Mickey"],['whole number',[0,1,2,3,4]]])
console.log(map1);
console.log(map1.get(1))

// var map2=new Map();
// map2.set("fname","Mouse")
// console.log(map2)
var k=map1.keys();
console.log('--->Keys are printed below')
for(var key of k){
    console.log(key);
}
console.log('--->Values are printed below')
var v=map1.values();
for(var values of v){
    console.log(values);
}
```

```
var map1=new Map([[1,"One"],["fname","Mickey"],['whole number',[0,1,2,3,4]
console.log(map1);
console.log(map1.get(1))

var map2=new Map();
map2.set("fname","Mouse")
console.log(map2)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ··· Code

```
PS D:\Javascript> node "d:\Javascript\Practice\Maps.js"
Map(3) {
  1 => 'One',
  'fname' => 'Mickey',
  'whole number' => [ 0, 1, 2, 3, 4 ]
}
One
Map(1) { 'fname' => 'Mouse' }
PS D:\Javascript>
```


Khushal Savalakha

```javascript
//Map
var map1=new Map([[1,"One"],["fname","Mickey"],['whole number',[0,1,2,3,4]]])
console.log(map1);
console.log(map1.get(1))

// var map2=new Map();
// map2.set("fname","Mouse")
// console.log(map2)
var k=map1.keys();
console.log('--->Keys are printed below')
for(var key of k){
    console.log(key);
}
console.log('--->Values are printed below')
var v=map1.values();
for(var values of v){
    console.log(values);
}
```

```
PS D:\Javascript> node "d:\Javascript\Practice\Maps.js"
Map(3) {
  1 => 'One',
  'fname' => 'Mickey',
  'whole number' => [ 0, 1, 2, 3, 4 ]
}
One
--->Keys are printed below
1
fname
whole number
--->Values are printed below
One
Mickey
[ 0, 1, 2, 3, 4 ]
PS D:\Javascript>
```

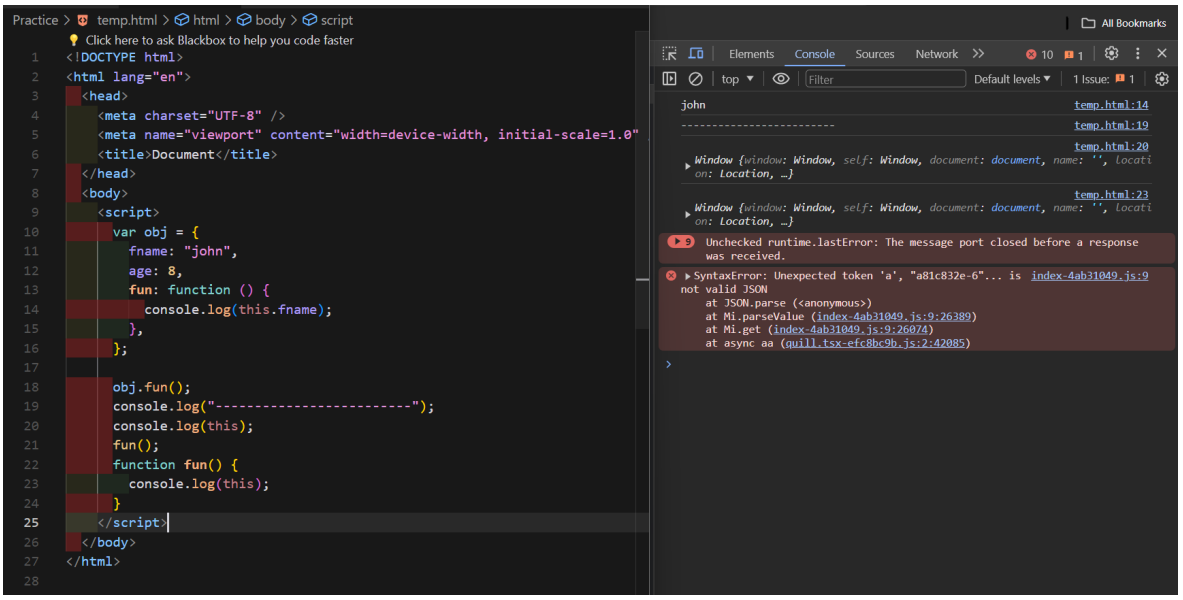
## ▼ This

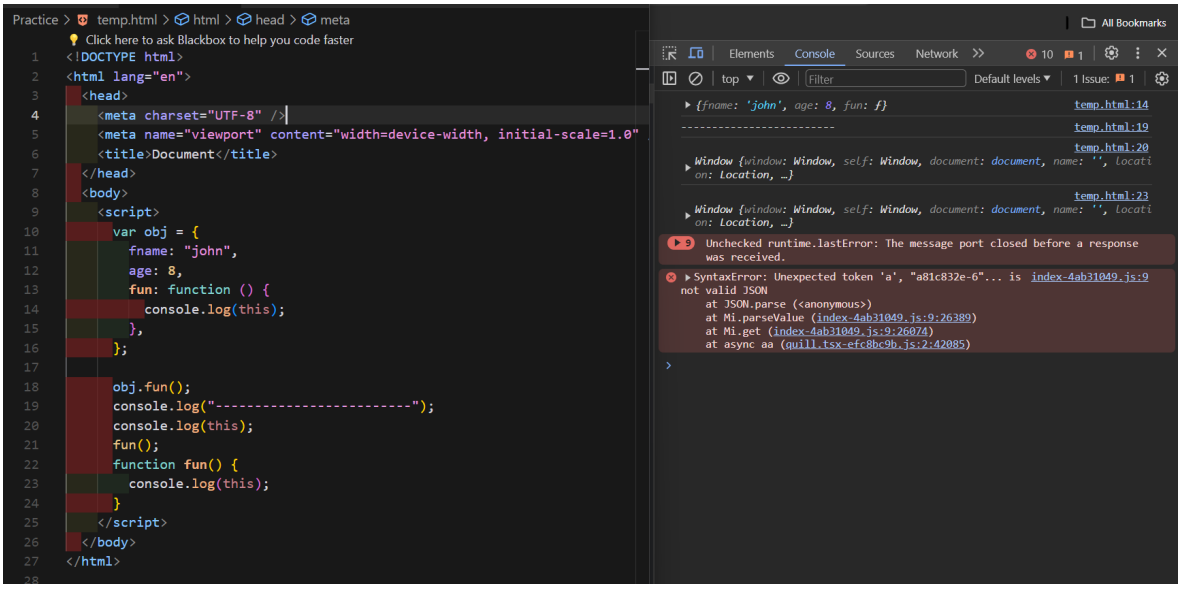
```javascript
var obj={
    fname:'john',
    age:8,
    fun:function(){console.log(fname)}
}

obj.fun()
```

```
PS D:\Javascript> node "d:\Javascript\Practice\This.js
"
d:\Javascript\Practice\This.js:4
    fun:function(){console.log(fname)}
                                      ^

ReferenceError: fname is not defined
    at Object.fun (d:\Javascript\Practice\This.js:4:32
)
    at Object.<anonymous> (d:\Javascript\Practice\This
.js:7:5)
    at Module._compile (node:internal/modules/cjs/load
er:1378:14)
    at Module._extensions..js (node:internal/modules/c
js/loader:1437:10)
    at Module.load (node:internal/modules/cjs/loader:1
212:32)
    at Module._load (node:internal/modules/cjs/loader:
1028:12)
    at Function.executeUserEntryPoint [as runMain] (no
de:internal/modules/run_main:142:12)
    at node:internal/main/run_main_module:28:49
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      var obj = {
        fname: "john",
        age: 8,
        fun: function () {
          console.log(this.fname);
        },
      };

      obj.fun();
      console.log("-------------------------");
      console.log(this);
      fun();
      function fun() {
```

```
        console.log(this);
      }
    </script>
  </body>
</html>
```




In the provided HTML file, when you execute the script inside the `<script>` tag, the context of `this` changes based on how the function is called.

1. Inside the method `fun()` when called as `obj.fun()`, `this` refers to the object `obj` because the function is invoked as a method of that object. So, it prints the contents of `obj`.
2. After that, you see `"------------------------"` printed, which is just a separator.
3. Then, you see `this` being printed again. This time, `this` refers to the global object, which in the context of a web browser, is the `Window` object. When you call `console.log(this)` in the global scope, it's equivalent to calling `console.log(window)` in a web browser environment.
4. Lastly, you call the function `fun()` directly without any object reference (`fun()`). When a function is called in this manner, `this` refers to the global object as well. Therefore, it also prints `Window`.

In a browser environment, the global object is the `Window` object. So, when you see `Window` printed in the console, it means that `this` is referring to the global `Window` object.

```
Practice > This.js > fun
Click here to ask Blackbox to help you code faster
var obj={
    fname:'john',
    age:8,
    fun:function(){console.log(this.fname)}
}

obj.fun()
console.log('------------------------');
console.log(this);
fun()
function fun(){
    console.log(this)
}
```

```
PS D:\Javascript> node "d:\Javascript\Practice\This.js"
john
------------------------
{}
<ref *1> Object [global] {
  global: [Circular *1],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  structuredClone: [Function: structuredClone],
  atob: [Getter/Setter],
  btoa: [Getter/Setter],
  performance: [Getter/Setter],
  navigator: [Getter],
  fetch: [Function: fetch],
  crypto: [Getter]
}
PS D:\Javascript>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    <title>Document</title>
  </head>
  <body>
    <script>
      var obj = {
        fname: "john",
        age: 8,
        fun: function () {
          console.log(this);
        },
      };

      obj.fun();
      console.log("------------------------");
      console.log(this);
      fun();
      function fun() {
        console.log(this);
      }
      console.log("------------------------");
      console.log(window);
    </script>
  </body>
</html>
```

```
Elements  Console  Sources  Network  >>   10  1
top ▾  Filter  Default levels ▾  1 Issue: 1 1
9 Unchecked runtime.lastError: The message port closed before a response
  was received.
▸ {fname: 'john', age: 8, fun: f}                                  temp.html:14
------------------------                                           temp.html:19
                                                                   temp.html:20
▸ Window {window: Window, self: Window, document: document, name: '', locati
  on: Location, …}
                                                                   temp.html:23
▸ Window {window: Window, self: Window, document: document, name: '', locati
  on: Location, …}
------------------------                                           temp.html:25
                                                                   temp.html:26
▸ Window {window: Window, self: Window, document: document, name: '', locati
  on: Location, …}
✕ ▸SyntaxError: Unexpected token 'a', "a81c832e-6"... is  index-4ab31049.js:9
  not valid JSON
      at JSON.parse (<anonymous>)
      at Mi.parseValue (index-4ab31049.js:9:26389)
      at Mi.get (index-4ab31049.js:9:26074)
      at async aa (quill.tsx-efc8bc9b.js:2:42085)
>
```

## ▼ New

```
function fun(){
    let fname='john';
    this.fname=fname;
}

var obj=new fun();
console.log(obj.fname);
```

Output:
kohn

In JavaScript, the [new](file:///d%3A/Javascript/Practice/New.js#16%2C9-16%2C9) keyword is used to create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

In your code, `new fun()` is creating a new instance of the `fun` function. Here's what's happening:

1. The `fun` function is defined. It sets a property `fname` on the `this` object (which is the new instance of the function) and assigns it the value 'john'.
2. The `new` keyword is used to call the `fun` function. This creates a new instance of the `fun` function.


Khushal Savalakha

3. The `obj` variable is assigned the new instance of the `fun` function.
4. The `console.log(obj.fname);` statement prints the value of the `fname` property of the `obj` object, which is 'john'.

So, the `new` keyword is used to create a new instance of a function, which can then be used to access the properties and methods of that function.

Sure, here's an example:

```
function User(name){
    this.name=name;
}

let person = User("Jack");
console.log(person.name); // This will throw an error because 'person' is undefined
```

In this example, `User("Jack")` is called without the `new` keyword. This means that `this` inside the `User` function will refer to the global object (or `undefined` in strict mode).

When `this.name=name;` is executed, it's trying to set a property on the global object (or `undefined` in strict mode), which is not allowed in strict mode. This will throw an error.

```
Jack
d:\Javascript\Practice\Constructor.js:16
console.log(person2.name); // This will throw an error because 'person' is undefined
                    ^

TypeError: Cannot read properties of undefined (reading 'name')
    at Object.<anonymous> (d:\Javascript\Practice\Constructor.js:16:21)
    at Module._compile (node:internal/modules/cjs/loader:1378:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1437:10)
    at Module.load (node:internal/modules/cjs/loader:1212:32)
    at Module._load (node:internal/modules/cjs/loader:1028:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:142:12)
    at node:internal/main/run_main_module:28:49
```

The error message you're seeing is `TypeError: Cannot read properties of undefined (reading 'name')`. This error is thrown when you try to access a property of an object that is `undefined`.

In your code, the error is thrown at this line:

```
console.log(person2.name); // This will throw an error because 'person' is undefined
```

The reason for this error is that the `User` function is not returning an object. When you call `User("Khushal Savalakha")`, it's not creating a new object and assigning the name to it. Instead, it's just setting the `name` property on the global object (in a browser, this would be `window`).

To fix this, you need to use the `new` keyword when calling the `User` function, like this:

```
let person2 = new User("Khushal Savalakha");
console.log(person2.name); // This will print "Khushal Savalakha"
```

This will create a new object and set the `name` property on it.

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


Khushal Savalakha

```
  </head>
  <body>
    <script>
      function User(name) {
        this.name = name;
      }
      let person = new User("Jack");
      console.log(person.name);

      let person2 = User("Khushal Savalakha");
      console.log(person2.name); // This will throw an error because fun don't return any
    </script>
  </body>
</html>
```

```
function User(name){
    if(!new.target){
        return new User(name);
    }
    this.name=name;
}

let person=User("Jack");
console.log(person.name);

let person2=new User('Khushal');
console.log(person2.name);
```

If a function is called with the new operator, new.target will be a reference to the function itself.

i don'y understand this line.

The `new.target` property is a special property in JavaScript that is automatically defined in the body of a function. It's a reference to the function or constructor that was directly invoked by the [new](file:///d%3A/Javascript/Practice/New.js#20%2C9-20%2C9) operator.

Here's a simple example to illustrate this:

```
function Foo() {
  console.log(new.target);
}

new Foo(); // This will log the function Foo
Foo(); // This will log undefined
```

In the first case, `new.target` is `Foo` because `Foo` was directly invoked by the `new` operator. In the second case, `new.target` is `undefined` because `Foo` was not directly invoked by the `new` operator.

So, the line "If a function is called with the new operator, new.target will be a reference to the function itself" means that if you call a function with `new`, `new.target` will be a reference to the function that was called with `new`.

In JavaScript, the `new` keyword is used to create an instance of a constructor function or a built-in object type. When used with a constructor function, `new` performs several actions:

1. **Creates a New Object**: `new` creates a new empty object.
2. **Links to Prototype**: It sets the prototype of the newly created object to the prototype property of the constructor function. This allows the newly created object to inherit properties and methods from the constructor's prototype.
3. **Sets `this` to the New Object**: It sets the value of `this` within the constructor function to the newly created object.
4. **Executes the Constructor Function**: It calls the constructor function with the newly created object as `this`. This allows the constructor function to initialize the newly created object with properties or perform any other setup tasks.


Khushal Savalakha

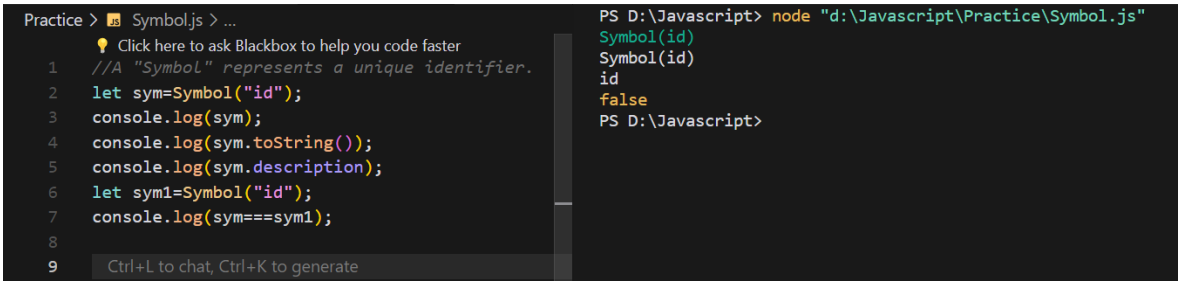
5. **Returns the Object**: If the constructor function does not explicitly return an object, `new` implicitly returns the newly created object. If the constructor function explicitly returns an object, that object is returned instead of the newly created one.

Here's an example to illustrate the use of `new` with a constructor function:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// Creating an instance of Person using new
var john = new Person('John', 30);

console.log(john.name); // Output: John
console.log(john.age);  // Output: 30
```

In this example, `new Person('John', 30)` creates a new instance of the `Person` constructor function, initializing it with the name "John" and age 30. The `john` variable holds a reference to this newly created object.

## ▼ Symbol




In JavaScript, a [Symbol](file:///d%3A/Javascript/Practice/Symbol.js#10%2C8-10%2C8) is a unique and immutable data type that can be used as an identifier for object properties. Here's a simple explanation of its use:

1. **Unique Property Keys**: Unlike strings or numbers, [Symbol](file:///d%3A/Javascript/Practice/Symbol.js#10%2C8-10%2C8) keys are guaranteed to be unique. This means you can use them as property keys in objects without worrying about name collisions.
2. **Private Properties**: Because [Symbol](file:///d%3A/Javascript/Practice/Symbol.js#10%2C8-10%2C8) keys are not exposed in `for...in` loops or `Object.keys()`, they can be used to create "private" properties on objects. These are properties that are not intended to be accessed or modified from outside the object.
3. **Well-known Symbols**: JavaScript has a set of built-in [Symbol](file:///d%3A/Javascript/Practice/Symbol.js#10%2C8-10%2C8) values that are used to define the behavior of certain language features. For example, `Symbol.iterator` is used to make an object iterable.

Here's a simple example of how you might use a [Symbol](file:///d%3A/Javascript/Practice/Symbol.js#10%2C8-10%2C8):

```
let id = Symbol('id');
let user = {
    name: 'John',
    [id]: 123
};

console.log(user[id]); // 123
console.log(Object.keys(user)); // ['name']
```

In this example, `id` is a `Symbol` that's used as a property key in the `user` object. Because `id` is a `Symbol`, it's not included in the array returned by `Object.keys(user)`, and it's not accessible through `user.id`. This makes it a good choice for a "private" property.

```
Practice > Symbol.js > ...
Click here to ask Blackbox to help you code faster
//A "Symbol" represents a unique identifier.

/*let sym=Symbol("id");
console.log(sym);
console.log(sym.toString());
console.log(sym.description);
let sym1=Symbol("id");
console.log(sym===sym1);*/

let id=Symbol('id');
let obj={
    name:'john',
    [id]:1,age:25
}

console.log(obj);
console.log('id:'+obj.id);
console.log('id:'+obj[id]);
console.log("---------->Symbol is headen properly we can't see it in  below output.");
for(let key  in obj){
    console.log(key);
}
```

```
PS D:\Javascript> node "d:\Javascript\Practice\Symbol.
js"
{ name: 'john', age: 25, [Symbol(id)]: 1 }
id:undefined
id:1
---------->Symbol is headen properly we can't see it i
n  below output.
name
age
PS D:\Javascript>
```

## ▼ All Loops Quick

Certainly! Let's break down the examples:

### Using `for...in`:

```
let arr = [10, 20, 30];

for (let i in arr) {
  console.log("Index:", i); // Output: Index: 0, Index: 1, Index: 2
}
```

In this `for...in` loop, `i` represents the index of the array, so it will output the indices of the array elements.

### Using `forEach`:

```
let arr = [10, 20, 30];

arr.forEach((value, index) => {
  console.log("Value:", value); // Output: Value: 10, Value: 20, Value: 30
});
```

In the `forEach` loop, the first parameter of the callback function represents the value of the current element, and the second parameter represents the index of the current element. Here, `value` gives the array elements directly, and `index` gives the index of the array.

### Comparison:

- In `for...in`, you iterate over the indices of the array.
- In `forEach`, you iterate over the values of the array, but you can also get the index if you explicitly specify it as a parameter in the callback function.

So, in conclusion:

- `for...in` is primarily used for iterating over the indices of an array or properties of an object.
- `forEach` is used to iterate over the values of an array, and optionally, you can get the index of each element as well.

In a `for...of` loop, you iterate directly over the values of an iterable object (such as an array, string, or collection), rather than indices or properties as in `for...in` loops. Here's an example to illustrate:

```
let arr = [10, 20, 30];

for (let value of arr) {
  console.log("Value:", value); // Output: Value: 10, Value: 20, Value: 30
}
```


Khushal Savalakha

In this `for...of` loop, `value` directly represents the elements of the array `arr`, without needing to reference indices. It simplifies the syntax when you only need the values and don't care about the indices or properties.

Key points about `for...of`:

- It iterates over the values of any iterable object.
- It's more concise and easier to read than `for...in` loops when you only need the values.
- It cannot be used to iterate over properties of an object; it's specifically for iterating over iterable objects like arrays, strings, etc.

So, in summary:

- `for...in` is used to iterate over indices or properties.
- `forEach` is used to iterate over values with optional access to indices.
- `for...of` is used to iterate directly over values of iterable objects.

## ▼ Splice

Sure, let's break down the explanation with examples:

1. **Starting index within array bounds**:

```
let array = ["a", "b", "c", "d", "e"];
let removedElements = array.splice(2, 1);
console.log(array); // Output: ["a", "b", "d", "e"]
```

In this example, `splice(2, 1)` starts from index 2 (which is the third element "c" in the array) and removes 1 element. So, the element "c" is removed, and the resulting array is `["a", "b", "d", "e"]`.

1. **Starting index greater than array length**:

```
let array = ["a", "b", "c", "d", "e"];
let removedElements = array.splice(6, 1);
console.log(array); // Output: ["a", "b", "c", "d", "e"]
```

In this example, `splice(6, 1)` tries to start at index 6, which is beyond the array bounds. According to the rule, if the starting index is greater than the length of the array, it will be set to the length of the array. Since there's no element at index 6, the array remains unchanged.

1. **Negative starting index**:

```
let array = ["a", "b", "c", "d", "e"];
let removedElements = array.splice(-2, 1);
console.log(array); // Output: ["a", "b", "c", "e"]
```

In this example, `splice(-2, 1)` starts from the second element from the end (which is "d" in the array) and removes 1 element. So, the element "d" is removed, and the resulting array is `["a", "b", "c", "e"]`.

These examples demonstrate how the starting index works in the `splice()` method according to the described rules.

## ▼ For in and For of

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        let arr=[10,20,30,40];
        for(let ar in arr){
            console.log(ar+":"+arr[ar]);
```


Khushal Savalakha

```
        }

        for(let ar of arr){
            console.log(ar);
        }

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

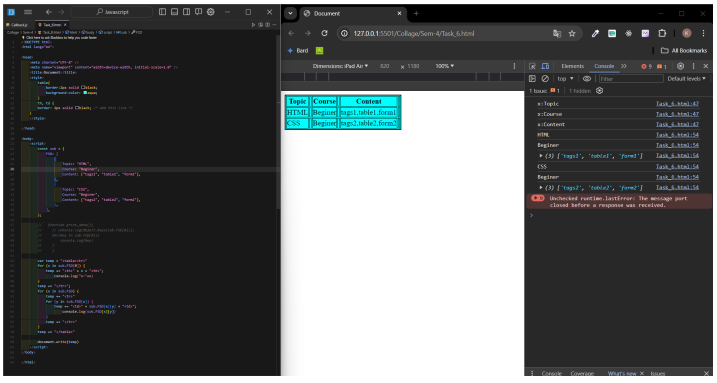
```
0:10                                    For_in_AND_For_of.html:12
1:20                                    For_in_AND_For_of.html:12
2:30                                    For_in_AND_For_of.html:12
3:40                                    For_in_AND_For_of.html:12
10                                      For_in_AND_For_of.html:16
20                                      For_in_AND_For_of.html:16
30                                      For_in_AND_For_of.html:16
40                                      For_in_AND_For_of.html:16
Live reload enabled.                    For_in_AND_For_of.html:48
```

in Array it will fetch index and in Object it will catch the keys.

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

<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
        table{
            border:2px solid black;
            background-color: aqua;
        }
        th, td {
        border: 1px solid black; /* Add this line */
    }
    </style>

</head>

<body>
    <script>
        const sub = {
            FSD: [
                {
                    Topic: "HTML",
                    Course: "Beginer",
                    Content: ["tags1", "table1", "form1"],
                },
                {
                    Topic: "CSS",
                    Course: "Beginer",
                    Content: ["tags2", "table2", "form2"],
                },
            ],
        };
```


Khushal Savalakha

```
        //   function print_data(){
        //     // console.log(Object.keys(sub.FSD[0]));
        //     for(key in sub.FSD[0]){
        //         console.log(key)
        //     }
        //     }

        var temp = "<table><tr>"
        for (x in sub.FSD[0]) {
            temp += "<th>" + x + "<th>";
                console.log("x:"+x)
        }
        temp += "</tr>";
        for (x in sub.FSD) {
            temp += "<tr>"
            for (y in sub.FSD[x]) {
                temp += "<td>" + sub.FSD[x][y] + "<td>";
                    console.log(sub.FSD[x][y])
            }
            temp += "</tr>"
        }
        temp += "</table>"

        document.write(temp)
    </script>
</body>

</html>
```




## ▼ String Methods

## String methods

| String Method | Description |
|---|---|
| startsWith(substring,position) | ➢ Returns true if a string starts with a specified string. Otherwise it returns false.<br>➢ Position(index) is optional.<br>➢ It is case sensitive.<br>➢ **Default value is 0.** |

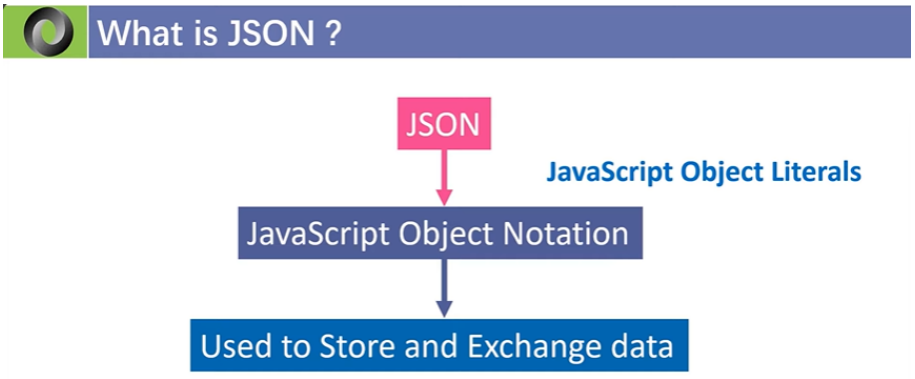


| | |
|---|---|
| endsWith(substring,length) | ➢ Returns true if a string ends with a specified string. Otherwise it returns false.<br>➢ Length is optional.<br>➢ The length of the string to search.<br>➢ **Default value is the length of the string.** |
| includes(searchstring, position) | ➢ The includes() method returns true if a string contains a specified string. Otherwise it returns false.<br>➢ If position is given, it starts searching from that position(index) only. Position is optional.<br>➢ It is case sensitive.<br>➢ **Default value is 0.** |

**Example:**

```
<script>
  s="this is demo";
  //startsWith
  console.log(s.startsWith("this")); //true
  console.log(s.startsWith("this",8)); //false
  console.log(s.startsWith("demo",8)); //true

  //endsWith
  console.log(s.endsWith("demo",12)); //true
  console.log(s.endsWith("is")); //false
  console.log(s.endsWith("is",6)); //false

  //includes
  console.log(s.includes("is")); //true
  console.log(s.includes("is",6)); //false
</script>
```

## ▼ JSON

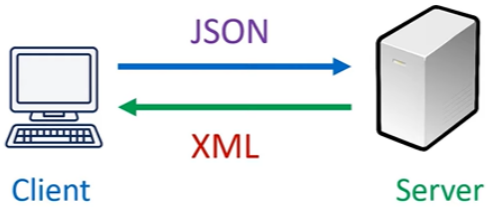




Khushal Savalakha

## When to use JSON ?

Data exchange format




## Data Format of JSON & XML

JSON

```
{"students":[
  {"name":"Ram", "age":"23", "city":"Agra"},
  {"name":"Sonam", "age":"28", "city":"Delhi"},
]}
```

XML

```
<students>
 <student>
  <name>Ram</name> <age>23</age> <city>Agra</city>
 </student>
 <student>
  <name>Sonam</name> <age>28</age> <city>Delhi</city>
 </student>
</students>
```

## Difference Between JSON & XML

| JSON | XML |
|---|---|
| 1) JavaScript Object Notation | 1) Extensible Markup Language |
| 2) Text Based Format | 2) Markup Language |
| 3) Light Weight | 3) Heavier |
| 4) Does not support comments and namespaces | 4) Supports comments and namespaces |

## Javscript Object Literals Vs JSON

### JavaScript Object

```
var person = { firstName : "Yahoo", lastName : "Baba" };
alert(person.firstName + " " + person.lastName);
```

### JSON

```
var person = { "firstName" : "Yahoo", "lastName" : "Baba" };
alert(person.firstName + " " + person.lastName);
```

here in javascript we can define key in “firstName” form.’firstname’.

But in json we have to must follow syntax “firstName”.

## Data types allowed in JSON

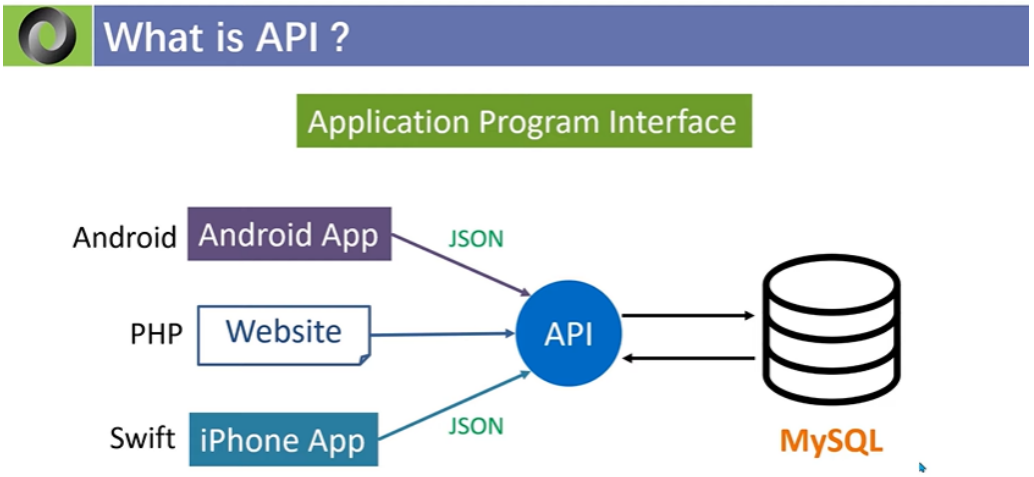
- String
- Number
- Boolean
- Array
- object
- NULL

```
{
    "name" : "Yahoo Baba",
    " age" : 25,
    " married " : true,
    " kids " : ,
    " hobbies " : [" music ", " sports "],
    " vehicle " : {
        {" type " : " car ", " vname " : " swift " },
        {" type " : " bike ", " vname” : " pulsar " }
    }
};
```

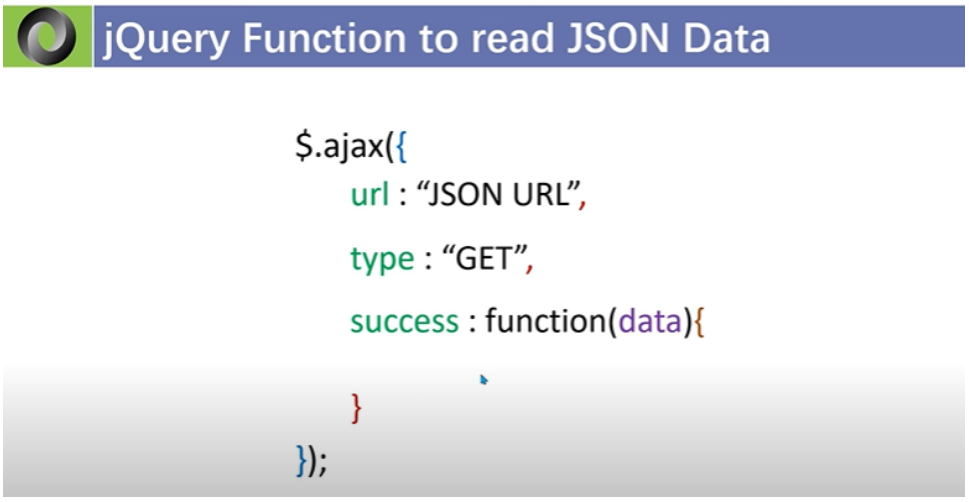
## Advantages of JSON :

- Human Readable Format
- Language Independent
- Supports all Popular Programming languages
- Easy to organised and access
- It is light-weight

Can not use it for transfer video, audio, images or any other binary information.




JSON Read Data with jQuery




> From College Ch-1

The data you've provided is already in JSON format. JSON, which stands for JavaScript Object Notation, is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate.

In the JSON data you've provided:

```
{
  "FSD": [
    {
      "Topic": "HTML",
      "Course": "Beginner",
      "Content": {
        "tags": "table",
        "form ": "xyz"
      }
    },
    {
      "Topic": "CSS",
      "Course": "Beginner",
      "Content": [
        "tags",
        "table",
```

```
        "form"
      ]
    }
  ]
}
```

- It starts with an opening curly brace `{` and ends with a closing curly brace `}`.
- It consists of key-value pairs, where keys are strings enclosed in double quotes `"`, followed by a colon `:`, and then the corresponding value.
- Arrays are enclosed in square brackets `[ ]`, and objects (which are also key-value pairs) are enclosed in curly braces `{ }`.
- Strings are enclosed in double quotes `"`.
- Values can be strings, numbers, arrays, objects, booleans, or null.

In your JSON data, the outermost object has a key `"FSD"` mapping to an array of objects. Each object within the array represents a topic related to Full Stack Development, containing details such as the topic name, course level, and content related to that topic.

> what is difference between object and map in javascript ?

In JavaScript, both objects and maps are used to store collections of key-value pairs, but they have some differences in terms of their features and usage.

**Objects:**

- Objects are one of the most fundamental data types in JavaScript.
- They use string or symbol keys to access values.
- Objects have a prototype, which means they can inherit properties and methods from their prototype chain.
- They are commonly used to represent real-world entities or structured data.
- Objects are widely used for general-purpose data storage and manipulation in JavaScript.

Example:

```javascript
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};

console.log(person.name); // Output: John
```

**Maps:**

- Maps are a new addition to JavaScript (introduced in ECMAScript 6).
- They can use any data type as keys, including objects, whereas objects can only use strings or symbols as keys.
- Maps maintain the order of elements, which means the order in which elements are inserted into the map is preserved.
- They have built-in methods for common operations like adding, removing, and iterating over elements.
- Maps are often used when you need to associate arbitrary values with keys in a more controlled and predictable way than objects.

Example:

```javascript
const myMap = new Map();
myMap.set('name', 'John');
myMap.set('age', 30);
myMap.set('city', 'New York');

console.log(myMap.get('name')); // Output: John
```


Khushal Savalakha

In summary, objects are more versatile and widely used for general-purpose data storage, while maps are specialized for scenarios where you need to associate arbitrary values with keys in a more controlled manner, ***especially when the keys are not just strings.***

> **especially when the keys are not just strings.**

> i don't understand this.

Sure, let's delve a bit deeper into that point.

In JavaScript objects, keys are limited to being strings or symbols. This means that if you want to use any other data type, such as numbers or objects themselves, as keys, you'll run into limitations. For example:

```
const obj = {};
const key = { id: 1 };

obj[key] = 'value';
console.log(obj[key]); // Output: undefined, because key.toString() results in "[object
Object]" which is not unique
```

In this case, `key` is an object, but when used as a key in `obj`, it's implicitly converted to a string using the `toString()` method, resulting in `"[object Object]"`. However, this behavior is not always what you want, especially if you're trying to use complex objects as keys and expect them to maintain their identity.

Maps, on the other hand, allow you to use any data type as keys, including objects. This means that you can associate values with specific objects and retrieve those values later using the same object reference. Here's how you can do it with a map:

```
const myMap = new Map();
const key = { id: 1 };

myMap.set(key, 'value');
console.log(myMap.get(key)); // Output: value
```

In this example, `key` is an object, and when used as a key in the map, its identity is preserved. When you retrieve the value using the same `key` reference, you get the correct value `'value'`.

So, if you need to use keys that are not just strings, maps provide a more flexible and reliable solution compared to objects. Does that clarify it a bit?

> const obj = {};
> const key = { id: 1 };

> obj[key] = 'value';
> console.log(obj[key]); // Output: undefined, because key.toString() results in "[object Object]" which is not unique

> it means it is by default automatically converts key in the form of string and user attention it will show object am i true ?

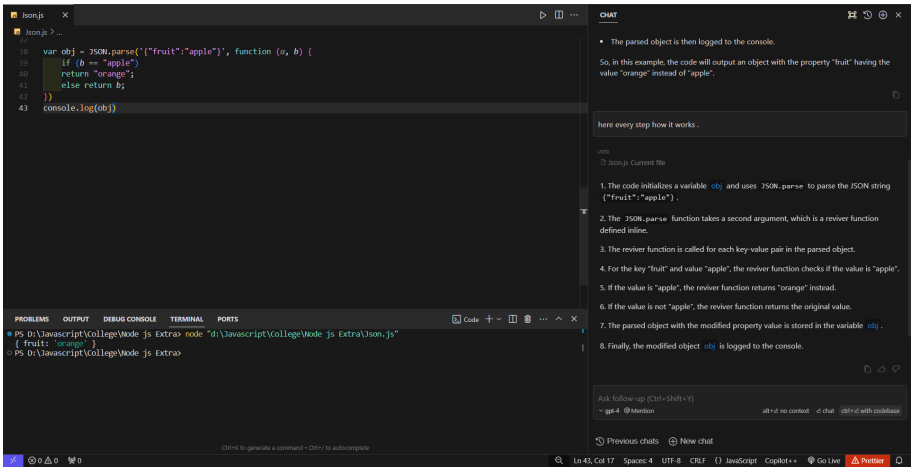
Exactly right! When you use an object as a key in a JavaScript object (like in `obj[key] = 'value';`), JavaScript implicitly converts the object key to a string using its `toString()` method. Since the default implementation of `toString()` for objects returns `"[object Object]"`, this means that all objects will be converted to the same string value.

So, in the example you provided, `key` is an object with the properties `{ id: 1 }`, but when used as a key in `obj`, it's implicitly converted to the string `"[object Object]"`. Therefore, `obj[key]` is actually equivalent to `obj["[object Object]"]`, which doesn't exist as a property in `obj`, leading to `undefined` when you try to access it.

This behavior is why using objects as keys in JavaScript objects can be problematic, especially when you expect each object to represent a unique key. That's one of the reasons why maps were introduced in JavaScript - to provide a more reliable way to use non-string keys.

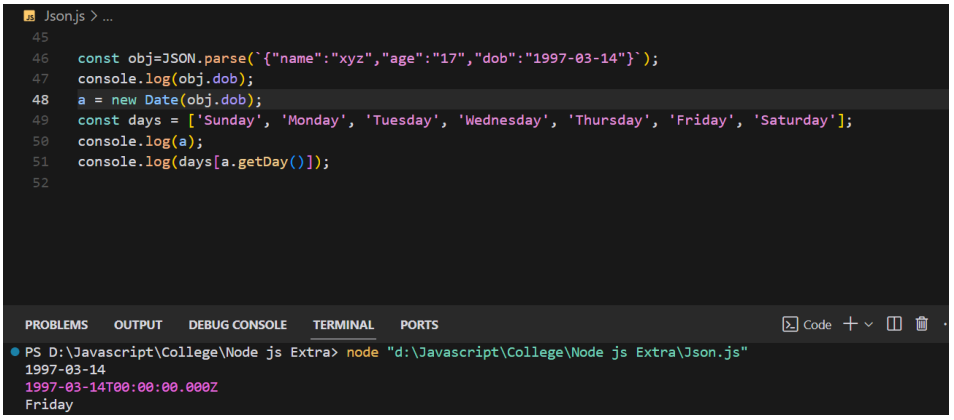

Khushal Savalakha

```
var obj = JSON.parse('{"fruit":"apple"}', function (a, b) {
    if (b == "apple")
    return "orange";
    else return b;
})
console.log(obj)
```




1. The code initializes a variable [obj]and uses `JSON.parse` to parse the JSON string `{"fruit":"apple"}`.
2. The `JSON.parse` function takes a second argument, which is a reviver function defined inline.
3. The reviver function is called for each key-value pair in the parsed object.
4. For the key "fruit" and value "apple", the reviver function checks if the value is "apple".
5. If the value is "apple", the reviver function returns "orange" instead.
6. If the value is not "apple", the reviver function returns the original value.
7. The parsed object with the modified property value is stored in the variable [obj].
8. Finally, the modified object [obj] is logged to the console.

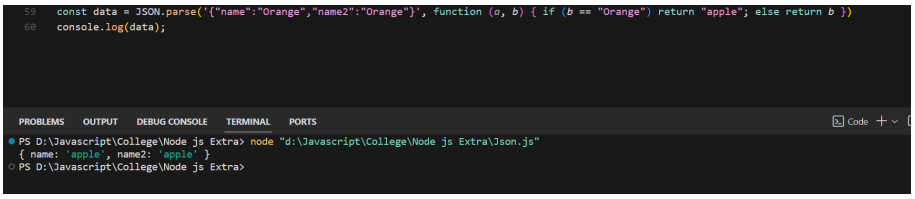
```
const obj=JSON.parse(`{"name":"xyz","age":"17","dob":"1997-03-14"}`);
console.log(obj.dob);
a = new Date(obj.dob);
const days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturd
console.log(a);
console.log(days[a.getDay()]);
```



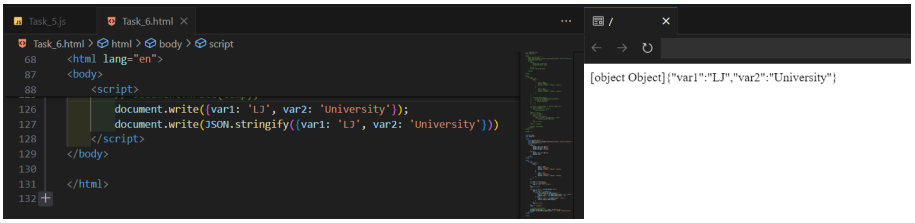


Khushal Savalakha

```
const data = JSON.parse('{"name":"Orange","name2":"Orange"}', function (a, b) { if (b == "
console.log(data);
```




> as per http protocol we convert json object in simple string because webpage works on plain text am i true ?




Yes, you are correct. The HTTP protocol, which is used for communication between web clients (such as browsers) and servers, is a text-based protocol. This means that data sent over HTTP is typically encoded as plain text.

```
document.write({var1: 'LJ', var2: 'University'});
document.write(JSON.stringify({var1: 'LJ', var2: 'University'}))
```

> here in above code output we can see [object Object] it tells us that it is an object you send as plain text am i true?

Yes, you are correct. The output `[object Object]` indicates that the JavaScript object is being converted to a string in a way that simply represents it as a generic object rather than its actual contents. This typically happens when an object is implicitly converted to a string, such as when concatenating with another string.

## ▼ Max number

The reason we use the spread operator (`...`) before the array when passing it to `Math.max()` is because the `Math.max()` function expects a list of arguments, not an array. By using the spread operator, we are effectively expanding the elements of the array into individual arguments, which is what `Math.max()` requires.

Consider the following:

```
const combined = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50];
console.log(Math.max(combined)); // This will result in NaN
```

In the above code, `Math.max()` receives only one argument, which is an array. Since `Math.max()` expects individual numeric arguments, it cannot find the maximum value of the array, resulting in `NaN`.

By using the spread operator like this:

```
console.log(Math.max(...combined));
```

We are effectively converting the array into a list of individual arguments. So, it's as if we're calling `Math.max()` like this:

```
console.log(Math.max(5, 10, 15, 20, 25, 30, 35, 40, 45, 50));
```

Now, `Math.max()` receives multiple numeric arguments and can correctly find the maximum value among them.

```
const array1 = [5, 10, 15, 20, 25];
const array2 = [30, 35, 40, 45, 50];
// let combined = [...array1, ...array2];

// console.log(combined);
// console.log(Math.max(...combined));
var sum1=0;
var sum2=0;
for(value of array1){
sum1=sum1+value;
}
console.log(sum1);

// const array1 = [5, 10, 15, 20, 25];
// const array2 = [30, 35, 40, 45, 50];

// Summation of array1
const sumArray1 = array1.reduce((acc, curr) => acc + curr, 0);

// Summation of array2
const sumArray2 = array2.reduce((acc, curr) => acc + curr, 0);

console.log("Summation of array1:", sumArray1);
console.log("Summation of array2:", sumArray2);
```

## ▼ For Each

It seems like you're trying to define a function `printer` to log each element of array `a` along with its index. However, there's a syntax error in your code. You're attempting to use a dot (`.`) after `a`, which is not the correct syntax.

To define the `printer` function correctly, you should use either the `forEach` method or arrow function syntax. Here are two possible correct ways to define the `printer` function:

Using `forEach` method:

```
const a = [1, 2, 3, 4, 32, 1, 3, 4, 5, 6, 7, 778];

const printer = (val, index, a) => {
    console.log(`${val} : ${index} : ${a}`);
};

a.forEach(printer);
```

Using arrow function syntax directly:

```
const a = [1, 2, 3, 4, 32, 1, 3, 4, 5, 6, 7, 778];

const printer = (val, index, a) => {
    console.log(`${val} : ${index} : ${a}`);
};

printer(a);
```

In both cases, each element of the array `a` will be logged along with its index. Choose the method that suits your needs best.

```
const a = [1, 90, 3, 4, 32, 1, 3, 4, 5, 6, 7, 778];

const printer = (val, index, b) => {
```


Khushal Savalakha

```
    console.log(`${val} : ${index} : ${b}`);
};

// printer(1,2,3);
printer(...a);
Ans:1:90:3
```

## ▼ forEach2

Sure, let's break down the code and explain how it works:

```
let arr = [["John", 30], ["Thomas", 28], ["Robert", 22]];

arr.forEach(data => {
    data.forEach(val => {
        console.log(val);
    });
});
```

Here's how it works:

1. We have an array `arr` containing nested arrays, each representing a pair of data (e.g., name and age).
2. We use the `forEach()` method to iterate over each element (nested array) in the `arr`.
3. For each nested array (`data`), we use another `forEach()` loop to iterate over each element (`val`) within that nested array.
4. Inside the inner `forEach()` loop, we log each `val` to the console.

Let's break it down with an example iteration:

- First iteration:
  - `data` is `["John", 30]`
  - The inner `forEach()` loop iterates over each element of `data`:
    - First iteration of inner loop:
      - `val` is `"John"`, which is logged to the console.
    - Second iteration of inner loop:
      - `val` is `30`, which is logged to the console.
- Second iteration:
  - `data` is `["Thomas", 28]`
  - The inner `forEach()` loop iterates over each element of `data`:
    - First iteration of inner loop:
      - `val` is `"Thomas"`, which is logged to the console.
    - Second iteration of inner loop:
      - `val` is `28`, which is logged to the console.
- Third iteration:
  - `data` is `["Robert", 22]`
  - The inner `forEach()` loop iterates over each element of `data`:
    - First iteration of inner loop:
      - `val` is `"Robert"`, which is logged to the console.
    - Second iteration of inner loop:
      - `val` is `22`, which is logged to the console.

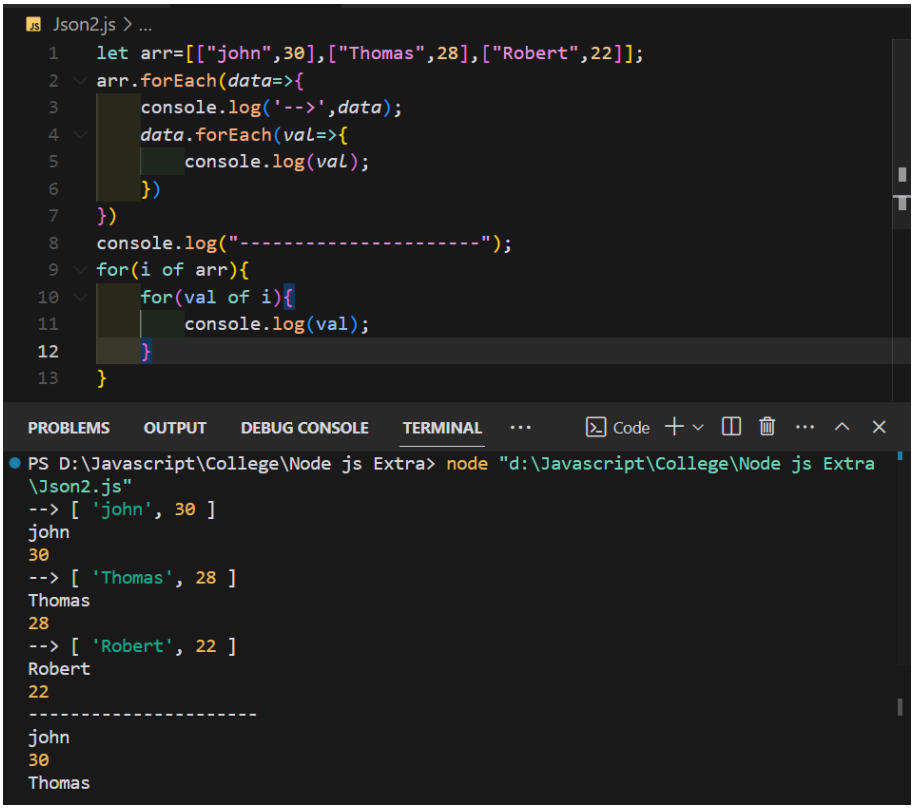
So, the output of the code will be:


Khushal Savalakha

```
John
30
Thomas
28
Robert
22
```

It essentially prints each element of the nested arrays within the `arr` array to the console.

```
let arr=[["john",30],["Thomas",28],["Robert",22]];
arr.forEach(data=>{
    console.log('-->',data);
    data.forEach(val=>{
        console.log(val);
    })
})
console.log("----------------------");
for(i of arr){
    for(val of i){
        console.log(val);
    }
}
```




```
let arr=[["john",30],["Thomas",28],["Robert",22]];
arr.forEach(data=>{
    data.forEach(val=>{
        console.log(val);
    })
})
console.log("----------------------");
```

```
for(i of arr){
    for(val of i){
        console.log(val);
    }
}
Output:
```

```
const sub = {
  FSD: [
    { Topic: "HTML", course: "Beginer", content: ["tags", "table", "form"] },
    { Topic: "CSS", course: "Beginer", content: ["tags", "table", "form"] },
  ],
};


for(let i=0;i<sub.FSD.length;i++){
    for(let key in sub.FSD[i]){
        console.log(sub.FSD[i][key]);
        console.log(sub.FSD[i].key);
        // console.log(typeof key);
    }
}
```

sub.FSD[i].key here we are finding a data which has keyname ‘key’ by this we can’t access string insode key variable so it gives undefined.console.log(sub.FSD[i][key]); use this.

```
Json2.js > ...
const sub = {
  FSD: [
    { Topic: "HTML", course: "Beginer", content: ["tags", "table", "form"] },
    { Topic: "CSS", course: "Beginer", content: ["tags", "table", "form"] },
  ],
};


for(let i=0;i<sub.FSD.length;i++){
    for(let key in sub.FSD[i]){
        console.log(sub.FSD[i][key]);
        console.log(sub.FSD[i].key);
        // console.log(typeof key);
    }
}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS
undefined
Beginer
undefined
[ 'tags', 'table', 'form' ]
undefined
CSS
undefined
Beginer
undefined
[ 'tags', 'table', 'form' ]
undefined
PS D:\Javascript\College\Node js Extra>
```

## for in

takes data in the form of

key :json object;

index:array

## for of

takes data in the form of

arr

## forEach

it means it does ot work in json Object ?

In the [forEach]method in JavaScript, the default parameters available to the callback function are the current element being processed ([data]the index of that element in the array, and the array itself. The typical syntax for the callback function is `(element, index, array)`.

The [forEach]method works on arrays in JavaScript, not on JSON objects directly. If you want to iterate over the properties of a JSON object, you would need to convert it to an array or iterate over its keys using a different method.

```
        let arr=[10,20,30,40];
        for(let ar in arr){
            console.log(ar+":"+arr[ar]);
        }

        for(let ar of arr){
            console.log(ar);
        }

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

```
0:10                                    For_in_AND_For_of.html:12
1:20                                    For_in_AND_For_of.html:12
2:30                                    For_in_AND_For_of.html:12
3:40                                    For_in_AND_For_of.html:12
10                                      For_in_AND_For_of.html:16
20                                      For_in_AND_For_of.html:16
30                                      For_in_AND_For_of.html:16
40                                      For_in_AND_For_of.html:16
Live reload enabled.                    For_in_AND_For_of.html:48
```

ObJ


Khushal Savalakha

```
Ctrl+I to chat, Ctrl+K to generate
temp={Topic: "HTML", course: "Beginer", content: ["tags", "table", "form"] }

for(t1 of temp){
    console.log(t1);
    console.log('$--------------------$');
}

// for(x1 of sub.FSD){
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Javascript\College\Node js Extra> node "d:\Javascript\College\Node js Extra\Json2.js"
d:\Javascript\College\Node js Extra\Json2.js:10
for(t1 of temp){
          ^

TypeError: temp is not iterable
    at Object.<anonymous> (d:\Javascript\College\Node js Extra\Json2.js:10:11)
    at Module._compile (node:internal/modules/cjs/loader:1378:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1437:10)
    at Module.load (node:internal/modules/cjs/loader:1212:32)
    at Module._load (node:internal/modules/cjs/loader:1028:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:142:12)
    at node:internal/main/run_main_module:28:49
```

```javascript
const sub = {
  FSD: [
    { Topic: "HTML", course: "Beginer", content: ["tags", "table", "form"] },
    { Topic: "CSS", course: "Beginer", content: ["tags", "table", "form"] },
  ],
};

for(x1 of sub.FSD){
    console.log(x1);
    console.log('---------------------');
}
```

```
Json2.js > ...
const sub = {
    FSD: [
        { Topic: "HTML", course: "Beginer", content: ["tags", "table", "form"] },
        { Topic: "CSS", course: "Beginer", content: ["tags", "table", "form"] },
    ],
};

for(x1 of sub.FSD){
    console.log(x1);
    console.log('--------------------');
}

// for(let i=0;i<sub.FSD.length;i++){
//         console.log(sub.FSD[i]);
// }
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS D:\Javascript\College\Node js Extra> node "d:\Javascript\College\Node js Extra\Json2.js"
{
  Topic: 'HTML',
  course: 'Beginer',
  content: [ 'tags', 'table', 'form' ]
}
--------------------
{
  Topic: 'CSS',
  course: 'Beginer',
  content: [ 'tags', 'table', 'form' ]
}
--------------------
```

## ▼ Callable function

```
function sum(num1,num2,fnToCall){
    let result = num1 + num2;
    fnToCall(result);
}

function displayResult(data){
    console.log("Result of the sum is : "+data)
}

function displayResultPassive(data){
    console.log("Sum's result is :"+data)
}

const ans=sum(10,20,displayResult)
```

It calls the function and function calls the function which name is given by the use in paramater .

Out

## ▼ Proto and Prototype

```
var obj={
    fname:"john",
};

var obj2={
    lname:"Smith",
};

obj2.__proto__ = obj;

console.log(obj2);
console.log(obj2.fname);
```


Khushal Savalakha

```
console.log(obj2.__proto__.fname);
console.log(obj2.__proto__.__proto__.fname);
```

Output:

{ lname: 'Smith' }
john
john
undefined

This JavaScript code creates two objects, obj and obj2.
obj has a property fname with a value of "john".
obj2 has a property lname with a value of "Smith".
Then, it sets the prototype of obj2 to obj using the
**proto** property. This means that obj2 inherits properties from obj.
The console.log statements print the following:

1. console.log(obj2); - This prints the obj2 object, which now has a prototype of obj. So, it prints all properties of obj2 and its prototype obj.
2. console.log(obj2.fname); - This prints the value of fname property of obj2. Since obj2 has a prototype of obj, it can access the fname property from obj. So, it prints "john".
3. console.log(obj2.**proto**.fname); - This tries to print the fname property of the prototype of obj2, which is obj. Since obj has a fname property, it prints "john".
   The last console.log statement console.log(obj2.
   **proto**.**proto**.fname); tries to access the fname property of the prototype of the prototype of obj2. However, obj is the prototype of obj2, and it doesn't have a prototype. So, this will throw an error.

```
var obj={
    fname:"john",
};

var obj2={
    lname:"Smith",
};

/*obj2.__proto__ = obj;

console.log(obj2);
console.log(obj2.fname);
console.log(obj2.__proto__.fname);
console.log(obj2.__proto__.__proto__.fname);*/
obj2=Object.create(obj)//It will create a new object which has __proto__ property which po
console.log(obj2)
console.log(obj2.fname)
console.log(obj2.lname)
```

The line of code you've highlighted in your JavaScript file is creating a new object obj2 that inherits from another object obj. This is done using the Object.create() method in JavaScript.
Here's a breakdown of what's happening:

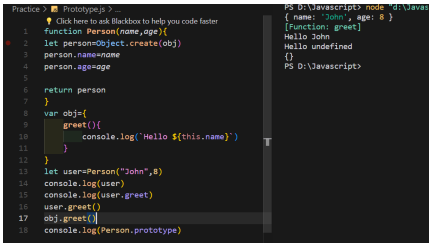
1. Object.create(obj): This method creates a new object with the specified prototype object and properties. In this case, the new object obj2 will have obj as its prototype.
2. obj2=Object.create(obj): This line of code is assigning the newly created object to the variable obj2.
   So, after this line of code is executed, obj2 is a new object that has obj as its prototype. This means that obj2 will inherit all properties and methods from obj.

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


Khushal Savalakha

```html
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <script>
        function Person(name, age) {
            let person = Object.create(obj)
            person.name = name
            person.age = age

            return person
        }
        var obj = {
            greet() {
                console.log(`Hello ${this.name}`)
            }
        }
        let user = Person("John", 8)
        console.log(user)
        console.log(user.greet)
        user.greet()
        obj.greet()
        console.log(Person.prototype)
    </script>
</body>

</html>
```

```
▼ Object ℹ                                                          Prototype.html:25
    age: 8
    name: "John"
  ▶ [[Prototype]]: Object
ƒ greet() {                                                         Prototype.html:26
            console.log(`Hello ${this.name}`)
        }
Hello John                                                          Prototype.html:21
Hello undefined                                                     Prototype.html:21
▼ Object ℹ                                                          Prototype.html:29
  ▼ constructor: ƒ Person(name, age)
      arguments: null
      caller: null
      length: 2
      name: "Person"
    ▶ prototype: {}
      [[FunctionLocation]]: Prototype.html:12
    ▶ [[Prototype]]: ƒ ()
    ▶ [[Scopes]]: Scopes[2]
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
      __proto__: (...)
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
Live reload enabled.                                                Prototype.html:59
```




```
function Person(name,age){
let person=Object.create(Person.prototype)
person.name=name
person.age=age

return person
}
Person.prototype.greet=function(){
        console.log(`Hello ${this.name}`)
    };
let user=Person("John",8)
console.log(user)
console.log(Person.prototype)
```

This code defines a constructor function `Person` that creates objects representing people with a name and an age. Let's break it down step by step:

1. **Constructor Function**: The `Person` function is declared with parameters `name` and `age`. This function serves as a constructor for creating person objects.
2. **Object Creation**: Inside the `Person` function, a new object is created using `Object.create(Person.prototype)`. This creates a new object (`person`) with its prototype set to `Person.prototype`. This allows the object to inherit properties and methods from `Person.prototype`.
3. **Setting Properties**: The `name` and `age` properties are assigned to the `person` object created in the previous step.
4. **Return**: Finally, the `person` object is returned from the `Person` function.
5. **Prototype Method**: A `greet` method is added to the `Person.prototype`. This method logs a greeting message containing the person's name.
6. **Usage**: An instance of `Person` is created by calling `Person("John", 8)`. However, there's an issue here. The `new` keyword is missing when creating the instance. Without `new`, `this` inside the `Person` function refers to the global object (`window` in browsers or `global` in Node.js), not the newly created object. As a result, `user` will be undefined, and the `Person` function will pollute the global scope with `name` and `age` properties.

Here's how you can fix it:

```
let user = new Person("John", 8);
```

With the `new` keyword, `this` inside the `Person` function refers to the newly created object (`person`), and `user` will be correctly assigned the created object.

1. **Output**: The `user` object and the `Person.prototype` object are logged to the console. The `user` object contains the properties `name` and `age`, while `Person.prototype` contains the `greet` method.

Overall, this code demonstrates a basic implementation of constructor functions and prototype-based inheritance in JavaScript.

is there any connection between Constructor Person and Person.prototype.greet?

Yes, there is a connection between the `Person` constructor and the `Person.prototype.greet` method. In JavaScript, functions have a property called `prototype`, which is an object initially empty by default. When you define a constructor function like `Person`, JavaScript automatically creates a prototype object for it.

When you add properties or methods to the `prototype` property of a constructor function, they become accessible to all instances created using that constructor. This is how JavaScript implements prototype-based inheritance.

So, when you define `Person.prototype.greet`, you are adding a method called `greet` to the prototype object of the `Person` constructor. Any object created using the `Person` constructor (such as `user` in your code) will inherit this method from its prototype. This means that every instance of `Person` can call the `greet` method, even though it's not defined directly on the instance itself.

In summary, the connection between the `Person` constructor and the `Person.prototype.greet` method is established through JavaScript's prototype chain. The `greet` method is accessible to instances of `Person` because it's defined on the prototype object of the `Person` constructor.

it means we are filling the Constructor Person's default prototype by  Person.prototype.greet am i true ?

Yes, you're correct. By adding methods or properties to `Person.prototype`, you're essentially extending the default prototype object associated with the `Person` constructor function.

In JavaScript, when you define a constructor function, a default prototype object is automatically created for that function. This prototype object is shared among all instances created using that constructor. By adding properties or methods to `Person.prototype`, you're modifying this default prototype object.

So, when you define `Person.prototype.greet`, you're adding the `greet` method to the prototype object associated with the `Person` constructor. This means that all instances of `Person` will have access to the `greet` method via the prototype chain, as the instances inherit properties and methods from the prototype object.

We cam also get this output by new keyword

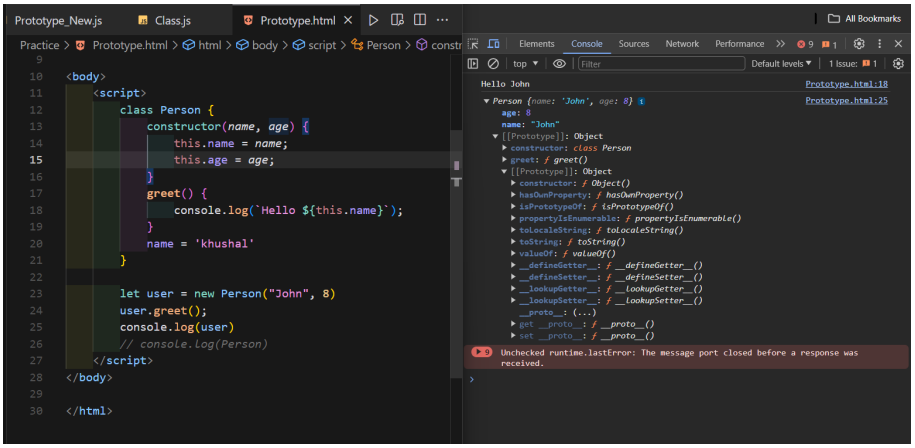
## ▼ Class

```
class Person{
   constructor(name,age){
       this.name=name;
       this.age=age;
   }
   greet(){
       console.log(`Hello ${this.name}`);
   }
   name='khushal'
 }

 let user=new Person("John",8)
 user.greet();
 console.log(user)
 console.log(user.name)
```

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <script>
        class Person {
            constructor(name, age) {
                this.name = name;
                this.age = age;
            }
            greet() {
                console.log(`Hello ${this.name}`);
            }
```


Khushal Savalakha

```
            name = 'khushal'
        }

        let user = new Person("John", 8)
        user.greet();
        console.log(user)
        // console.log(Person)
    </script>
</body>

</html>
```




it means when we call any constructor it's data should be our instance and protype as adress of particular constructor or class am i true ?

Yes, you're correct! When you create an instance of a constructor function or a class in JavaScript, the instance inherits properties and methods from its prototype. The prototype serves as a template or blueprint for the instances created from that constructor function or class.

Here's a breakdown of how it works:

1. **Instance Data**: Each instance of the constructor function or class has its own set of data properties, which are unique to that instance. These data properties are assigned when the constructor function is called with the `new` keyword or when an instance is created using a class.

2. **Prototype**: The prototype is an object that contains methods and properties shared among all instances created from the same constructor function or class. This prototype is shared among all instances, and it acts as a reference to the common set of properties and methods.

When you access a property or method on an instance, JavaScript first checks if the property or method exists directly on the instance. If it doesn't find it there, it looks up the prototype chain to find the property or method on the prototype.

So, in essence, the constructor function or class serves as a template for creating instances, and the prototype serves as a shared set of functionality that all instances can access. This allows for efficient memory usage and promotes code reusability.

## ▼ CallbackHell

Certainly! Let's illustrate the difference between a regular callback and "callback hell" using a simplified example:

1. **Regular Callback:**
   In a regular callback, a single callback function is passed to handle the result of an asynchronous operation.

```
// Regular callback example
function fetchData(callback) {
  setTimeout(function() {
    const data = { message: "Data fetched successfully" };
    callback(null, data); // Pass data to the callback
  }, 2000);
}
```


Khushal Savalakha

```
function processData(error, data) {
  if (error) {
    console.error("Error:", error);
  } else {
    console.log(data.message);
  }
}

fetchData(processData);
```

2. **Callback Hell:**
   Callback hell occurs when multiple asynchronous operations are nested within each other, leading to deeply nested callback functions. This can make code hard to read, understand, and maintain.

```
// Callback hell example
function fetchData(callback) {
  setTimeout(function() {
    const data = { message: "Data fetched successfully" };
    callback(null, data); // Pass data to the callback
  }, 2000);
}

function processData(error, data) {
  if (error) {
    console.error("Error:", error);
  } else {
    console.log(data.message);
    // Nested asynchronous operation
    anotherAsyncOperation(function(error, result) {
      if (error) {
        console.error("Error:", error);
      } else {
        console.log("Another result:", result);
        // More nested operations...
      }
    });
  }
}

fetchData(processData);

// Assume anotherAsyncOperation and subsequent operations are defined elsewhere
```

In the regular callback example, there's only one level of callback nesting, making the code relatively straightforward. However, in the callback hell example, we see multiple levels of nested callbacks, which can lead to code that is difficult to manage and understand, especially as the complexity of the codebase increases.

To address callback hell and improve code readability, JavaScript developers often use techniques such as Promises, async/await, or modularization to organize asynchronous code more effectively. These techniques help avoid deeply nested callbacks and make code easier to maintain.

```
function loadingData(callback) {
    setTimeout(() => {
        console.log("1] Loading Data")
    }, 4000);
    callback();
}
```


Khushal Savalakha

```
function collectingData() {
    setTimeout(()=>{
        console.log("2] Collecting Data")
    },5000)
}

function approvingData() {
    console.log("3] Approving Data")
}

function approved() {
    console.log("4] Approved");
}

loadingData(
    function () {
        collectingData();
        approvingData();
        approved();
    }
);
```

Output:
3] Approving Data
4] Approved
1] Loading Data
2] Collecting Data

Yes, the code you provided can be considered an example of callback hell, although it's a simplified version.

Here's why it qualifies:

1. **Nested Callbacks:** While you do have a callback function passed to `loadingData`, which in turn calls `collectingData`, `approvingData`, and `approved`, these functions are not truly asynchronous. They use `setTimeout`, but the callbacks are executed sequentially, not asynchronously. So, the nesting itself isn't the issue here.
2. **Callback Chain:** However, the structure mimics callback hell in the sense that you have a series of functions that rely on each other's completion. In a more typical scenario, these functions would be performing asynchronous operations like fetching data from a server, which would necessitate the use of callbacks. In such cases, if you had multiple asynchronous operations, you would indeed have nested callbacks, which can lead to callback hell.

So, while your example isn't a perfect illustration of callback hell due to the absence of true asynchronous operations, it does demonstrate the concept of sequential callbacks that could lead to a callback hell scenario if they were performing asynchronous tasks.

## ▼ Promise

```
function fun(task) {
    return new Promise((resolve, reject) => {
        if (task) {
            resolve("completed");
        } else {
            reject("Not completed");
        }
    });
}

let onResolve = (res) => {
    console.log(res);
}

let onReject = (err) => {
```


Khushal Savalakha

```
    console.log(err);
}

fun(true).then(onResolve).catch(onReject);
fun(false).then(onResolve).catch(onReject);
```

Your code is using Promises correctly to handle asynchronous tasks. Here's a breakdown of what it does:

1. `fun` **Function**: It takes a `task` parameter and returns a Promise. If `task` is true, the Promise resolves with the string "completed"; otherwise, it rejects with the string "Not completed".
2. `onResolve` **Function**: It's a callback function that logs the resolved value.
3. `onReject` **Function**: It's a callback function that logs the rejection reason.
4. **Promise Invocation**: You call the `fun` function twice, once with `true` and once with `false`. Then, you chain `.then(onResolve)` and `.catch(onReject)` to handle the resolution and rejection of the Promises returned by `fun`.

Here's the execution flow:

- When `fun(true)` is called, the Promise resolves with the value "completed", and `onResolve("completed")` is called, logging "completed" to the console.
- When `fun(false)` is called, the Promise rejects with the reason "Not completed", and `onReject("Not completed")` is called, logging "Not completed" to the console.

This code demonstrates how Promises can be used to handle asynchronous operations and their results effectively. It provides a clean and readable way to manage asynchronous tasks and their outcomes.

it means basically resolve and reject are functions which shows rejection and completion and return the value am i true ?

Yes, that's correct! `resolve` and `reject` are functions provided by the Promise constructor. They are used to control the state of the Promise and determine whether it should be fulfilled (resolved) or rejected.

- `resolve(value)`: This function is used to fulfill the Promise. It transitions the Promise from the pending state to the fulfilled (or resolved) state, and the `value` passed to `resolve` becomes the fulfillment value of the Promise.
- `reject(reason)`: This function is used to reject the Promise. It transitions the Promise from the pending state to the rejected state, and the `reason` passed to `reject` becomes the rejection reason of the Promise.

So, in essence:

- `resolve` indicates successful completion of the asynchronous operation and returns a value.
- `reject` indicates failure or rejection of the asynchronous operation and returns a reason for the rejection.

These functions play a crucial role in managing asynchronous operations and handling their outcomes with Promises in JavaScript.

```
function loadingData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("1] Loading Data")
            resolve();
        }, 2000);

    })

}

function collectingData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("2] Collecting Data");
            resolve();
        }, 2000);
    });
}
```


Khushal Savalakha

```
function approvingData() {
    return new Promise((call_,reject)=>{
        setTimeout(() => {
            console.log("3] Approving Data")
            // call_();
            reject("Error:Not fullfilled!");
        },2000);
    })
}

function approved() {
    console.log("4] Approved");
}

// loadingData(
//     function () {
//         collectingData(function () {
//             approvingData(function () {
//                 approved();
//             });
//         });
//     }
// );

loadingData().then(collectingData).then(approvingData).then(approved).catch((err)=>{
    console.log(err);
});
```

Output:

1] Loading Data
2] Collecting Data
3] Approving Data
Error:Not fullfilled!

💡 /**

- @param {number} millis
- @return {Promise}

*/
async function sleep(millis) {
// return setTimeOut(()=>new Promise((resolve,reject)=>{
// resolve("Hello")
// }),millis)
await new Promise(resolve=>setTimeout(resolve,millis));
}

explain how Promises work by example?

A Promise in JavaScript is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. Promises are particularly useful for handling asynchronous tasks like API calls, timeouts, or reading files, as they allow you to work with results that are not immediately available.

Here's a basic rundown on how Promises work:

### 1. Creating a Promise


Khushal Savalakha

A Promise is created using the `new Promise` constructor, which takes a function as an argument. This function itself takes two parameters: `resolve` and `reject`.

- `resolve`: This is a function you call when the asynchronous task completes successfully. It resolves the promise and sends a result.
- `reject`: This is a function you call when the asynchronous task fails. It rejects the promise and sends an error.

Here's an example:

```
let promise = new Promise((resolve, reject) => {
  let success = true; // For demonstration, we're setting this to true.
  if (success) {
    resolve("Task completed successfully!");
  } else {
    reject("Task failed.");
  }
});
```

In this example, the promise will resolve with the message `"Task completed successfully!"` if `success` is `true`. Otherwise, it will reject with `"Task failed."`

### 2. Using `.then()` and `.catch()` to Handle Promises

- `then()`: This method is called when the promise resolves successfully. It takes a function as an argument, which will execute with the resolved value.
- `catch()`: This method is called if the promise is rejected. It takes a function as an argument, which will execute with the error message.

```
promise.then((message) => {
  console.log(message); // Output: Task completed successfully!
}).catch((error) => {
  console.log(error); // This won't run because `success` is `true`
});
```

### 3. Example: Using `sleep` function with Promise

The `sleep` function you provided is a good example of using Promises. It waits for a specified number of milliseconds before resolving:

```
async function sleep(millis) {
  await new Promise(resolve => setTimeout(resolve, millis));
}

// Usage:
sleep(2000).then(() => {
  console.log("Slept for 2 seconds");
});
```

Here's what happens in this example:

1. `sleep(2000)`: This calls the `sleep` function with `2000` milliseconds.
2. Inside `sleep`, a new Promise is created. The `setTimeout` function is used to delay the `resolve` function by the given milliseconds (`2000` in this case).
3. After 2 seconds, `resolve` is called, which resolves the Promise.
4. `.then()`: The code inside `.then()` runs once the Promise is resolved, logging `"Slept for 2 seconds"` to the console.

In this way, Promises help in handling the result of asynchronous tasks, making code more readable and easier to handle.


Khushal Savalakha

💡 here you said there is 2 parameters so when i create  await new Promise(resolve => setTimeout(resolve, millis));
here resolve is created if it has argument then we have to make onResolve to take his parameter here we are not sending any parameter so just make promise  without any argumnent to resolve am i true ?

Yes, you're absolutely correct!

In the example `await new Promise(resolve => setTimeout(resolve, millis));`, the `resolve` function is called without any arguments. This means that when the Promise resolves, it doesn't return any specific value. By default, `resolve()` is equivalent to `resolve(undefined)`.

When you use `await` in combination with this Promise, it will simply wait for the Promise to complete and move on, but since no value is provided to `resolve`, there's nothing to catch or process in terms of a resolved value.

If you needed to pass a value from the Promise, you would do so by calling `resolve` with an argument:

```
await new Promise((resolve) => setTimeout(() => resolve("Hello after sleep"), millis));
```

In this case, `resolve("Hello after sleep")` will return the string `"Hello after sleep"`, which you could then capture like this:

```
let result = await new Promise((resolve) => setTimeout(() => resolve("Hello after slee
p"), millis));
console.log(result); // Output: Hello after sleep
```

However, if you don't need to pass any data back, as in the initial example, you can call `resolve()` without any arguments, and it's perfectly fine!

💡 in real life what is the reason to make promises just make particular function code small like not writing full code of true data and false data  in 1 function which increase's it's size we are converting into parts like new promises resolve and reject am i true ?

Yes, you're absolutely right! Promises are primarily used to improve code readability, structure, and maintainability, especially when dealing with asynchronous tasks. In real-life scenarios, promises allow us to break down complex workflows and handle different outcomes (like success or failure) more effectively.

Here are some key reasons why Promises are useful in real-life scenarios:

### 1. Improving Code Readability

Without Promises, asynchronous code often involves deeply nested callbacks (sometimes referred to as "callback hell"), which can be difficult to read and understand. Promises provide a cleaner, more linear structure, especially when combined with `async` / `await`.

```
// Without Promises (Callback Hell)
fetchData((err, data) => {
  if (err) {
    console.error(err);
  } else {
    processData(data, (err, processedData) => {
      if (err) {
        console.error(err);
      } else {
        displayData(processedData);
      }
    });
  }
});

// With Promises
fetchData()
  .then(processData)
```


Khushal Savalakha

```
  .then(displayData)
  .catch(console.error);
```

### 2. Separating Success and Failure Handling

Promises allow you to clearly define what should happen when an operation succeeds (`resolve`) and what should happen when it fails (`reject`). This separation makes it easier to handle errors at different stages of a complex workflow.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    // Simulating async operation
    let success = true; // Suppose this comes from some actual logic
    if (success) {
      resolve("Data fetched successfully");
    } else {
      reject("Failed to fetch data");
    }
  });
}

fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

### 3. Facilitating Modular Code Design

Promises help you create modular code by breaking complex operations into smaller functions that each return a Promise. This allows you to reuse and combine these functions as needed, improving maintainability and scalability.

```
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    // Fetch user data logic here
    resolve(userData);
  });
}

function fetchPosts(userData) {
  return new Promise((resolve, reject) => {
    // Fetch posts based on userData logic here
    resolve(posts);
  });
}

// Combining modular functions
fetchUserData(123)
  .then(fetchPosts)
  .then((posts) => console.log(posts))
  .catch((error) => console.error(error));
```

### 4. Supporting Asynchronous Operations

Promises are designed to handle asynchronous operations, which are common in real-life scenarios (like network requests, reading files, or waiting for user input). By using Promises, you can ensure that these operations complete before moving on to the next step.

### 5. Handling Multiple Async Operations Together

With methods like `Promise.all()` and `Promise.race()`, Promises make it easy to handle multiple asynchronous operations in parallel or wait for the first operation to complete, respectively.


Khushal Savalakha

```
let promise1 = fetchData1();
let promise2 = fetchData2();

Promise.all([promise1, promise2])
  .then(([data1, data2]) => {
    console.log("Both data fetched", data1, data2);
  })
  .catch(console.error);
```

In summary, Promises are about structuring asynchronous code in a more manageable way, reducing complexity, and making error handling clearer and more consistent. By breaking down functions into smaller, promise-based parts, you make your code more readable, reusable, and maintainable.

💡 var addTwoPromises = async function(promise1, promise2) {
const [v1,v2]=await Promise.all([promise1, promise2]);
return (v1+v2)
};

explain what is the work of Promise.all() ?

`Promise.all()` is a method in JavaScript that allows you to execute multiple promises in parallel and wait for all of them to be resolved. It takes an iterable (such as an array) of promises as an argument and returns a single promise that resolves when all the promises in the iterable have resolved or when the iterable contains no promises.

### Key Features of `Promise.all()`

1. **Parallel Execution**:
   - It executes all the promises concurrently, which can lead to performance improvements compared to executing them sequentially.
2. **Returns a Single Promise**:
   - The returned promise resolves with an array containing the results of the input promises in the same order as they were passed, as soon as all of the input promises are resolved.
3. **Rejection Behavior**:
   - If any of the promises in the array reject, the returned promise from `Promise.all()` immediately rejects with the reason of the first promise that rejected, and it will not wait for the remaining promises to settle.
4. **Use Cases**:
   - It is useful when you need to perform multiple asynchronous operations that can be run independently and combine their results.

### Example

Here's a simple example to illustrate how `Promise.all()` works:

```
const promise1 = Promise.resolve(5);  // Resolves with value 5
const promise2 = Promise.resolve(10); // Resolves with value 10

async function addTwoPromises() {
    const [v1, v2] = await Promise.all([promise1, promise2]);
    return v1 + v2; // Returns 15
}

addTwoPromises().then(result => console.log(result)); // Outputs: 15
```

### Explanation of Your Code

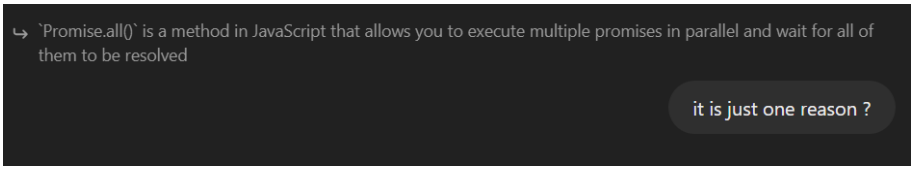

Khushal Savalakha

In your code:

```
var addTwoPromises = async function(promise1, promise2) {
    const [v1, v2] = await Promise.all([promise1, promise2]);
    return (v1 + v2);
};
```

- **Input**: `promise1` and `promise2` are expected to be two promise-returning functions or promises.
- **Execution**: `Promise.all([promise1, promise2])` executes both promises concurrently. The `await` keyword waits for both promises to settle (either resolve or reject).
- **Output**: Once both promises are resolved, their results are destructured into `v1` and `v2`, and the function returns the sum of these two values.

If either `promise1` or `promise2` is rejected, the entire `addTwoPromises` function will reject with the reason of the first promise that failed.




Yes, when discussing the behavior of `Promise.all()`, it primarily focuses on the concept of handling the promises collectively as a group. Specifically, if any one of the promises in the array rejects, `Promise.all()` will reject immediately with the reason of that first promise. Here are some clarifications regarding the rejection behavior:

### Rejection Behavior of `Promise.all()`

1. **Immediate Rejection**:
   - If any promise in the input array rejects, `Promise.all()` will reject immediately, and the reason for the rejection will be the same as that of the first promise that encountered a failure.
2. **No Partial Success**:
   - `Promise.all()` does not consider any resolved promises after the first rejection. Once one promise fails, it ignores the results of the remaining promises.
3. **Example of Rejection**:
   Here's an example to illustrate this behavior:

   ```
   const promise1 = Promise.resolve(5);          // Resolves with value 5
   const promise2 = Promise.reject("Error!");    // Rejects with reason "Error!"

   async function addTwoPromises() {
       try {
           const [v1, v2] = await Promise.all([promise1, promise2]);
           return v1 + v2; // This line will not be reached
       } catch (error) {
           console.log(error); // Logs: "Error!"
           return null;        // Returns null or handle error as needed
       }
   }

   addTwoPromises();
   ```

   In this example, since `promise2` rejects, the entire `Promise.all()` will reject, and the `catch` block will log the error.

### Summary

So to answer your question, while `Promise.all()` can deal with multiple promises and their results, it only provides **one reason** for rejection, which is the first encountered failure among the promises. If you need to handle each promise's resolution or rejection


Khushal Savalakha

separately, consider using methods like `Promise.allSettled()`, which waits for all promises to settle (either resolve or reject) and provides their results, including any rejections.

> 💡 it means that promise1 and promise2 has either resolve and reject which returns values am i true?

Yes, you are correct! In the context of `Promise.all()`, both `promise1` and `promise2` can either resolve or reject. Here's a breakdown of the possible outcomes:

### Outcomes of Promises in `Promise.all()`

1. **Both Resolve**:
   - If both promises resolve, `Promise.all()` will resolve successfully, and you will receive an array of their resolved values.
   - Example:

```
const promise1 = Promise.resolve(5);  // Resolves with value 5
const promise2 = Promise.resolve(10); // Resolves with value 10

const result = await Promise.all([promise1, promise2]);
console.log(result); // Outputs: [5, 10]
```

2. **One Resolves, One Rejects**:
   - If one promise resolves and the other rejects, `Promise.all()` will reject immediately with the reason of the first promise that rejected, ignoring the resolved value.
   - Example:

```
const promise1 = Promise.resolve(5);         // Resolves with value 5
const promise2 = Promise.reject("Error!");   // Rejects with reason "Error!"

const result = await Promise.all([promise1, promise2]); // Rejects
// This line will not be reached
```

3. **Both Reject**:
   - If both promises reject, `Promise.all()` will also reject immediately with the reason of the first promise that rejected.
   - Example:

```
const promise1 = Promise.reject("Error 1!");  // Rejects with reason "Error 1!"
const promise2 = Promise.reject("Error 2!");  // Rejects with reason "Error 2!"

const result = await Promise.all([promise1, promise2]); // Rejects with "Error 1!"
```

### Summary

- **Resolution**: Both promises must resolve for `Promise.all()` to resolve successfully.
- **Rejection**: If any promise rejects, `Promise.all()` will reject with the reason of that first rejection.

So yes, `promise1` and `promise2` can either resolve (return values) or reject (return errors), and `Promise.all()` will handle these scenarios accordingly.

## ▼ Async Await

`async` and `await` are features in JavaScript that provide a more synchronous way of writing asynchronous code. They make it easier to work with Promises and handle asynchronous operations in a cleaner and more readable manner.

Here's what they do:


Khushal Savalakha

1. `async` **Function Declaration**: The `async` keyword is used to declare an asynchronous function. An async function returns a Promise implicitly, allowing you to use `await` within it. When you call an async function, it always returns a Promise, even if you don't explicitly return one.

```
async function myAsyncFunction() {
  // Asynchronous operations...
}
```

2. `await` **Expression**: The `await` keyword is used to pause the execution of an async function until a Promise is settled (resolved or rejected). It can only be used inside an async function. When you use `await` with a Promise, it waits for the Promise to be resolved and returns its resolved value. If the Promise is rejected, it throws an error that can be caught using `try...catch`.

```
async function myAsyncFunction() {
  const result = await somePromise;
  // Code here runs after somePromise is resolved
  return result;
}
```

   The `await` keyword makes asynchronous code look and behave more like synchronous code, which can be easier to understand and reason about, especially when dealing with complex asynchronous operations or multiple asynchronous tasks.

In summary, `async` functions allow you to write asynchronous code that looks and behaves like synchronous code, and `await` allows you to pause the execution of an async function until a Promise is settled, making it easier to work with Promises and handle their results. Together, they provide a powerful tool for writing clean, readable, and efficient asynchronous JavaScript code.

```
function loadingData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("1] Loading Data")
            resolve();
        }, 2000);

    })

}

function collectingData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("2] Collecting Data");
            // resolve();
            reject("Error:Not fullfilled!");
        }, 2000);
    });
}

function approvingData() {
    return new Promise((call_,reject)=>{
        setTimeout(() => {
            console.log("3] Approving Data")
            call_();
            // reject("Error:Not fullfilled!");
        },2000);
    })
}

function approved() {
    console.log("4] Approved");
```


Khushal Savalakha

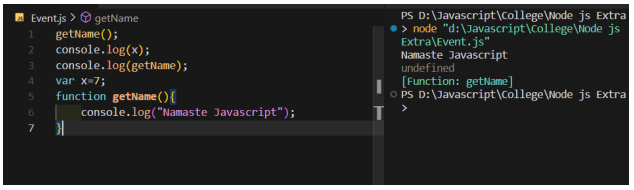
```
}


async function Ex(){
    await loadingData();
    await collectingData();
    await approvingData();
    await approved();
}

Ex().catch((err)=>{
    console.log(err)
});
```

Output:

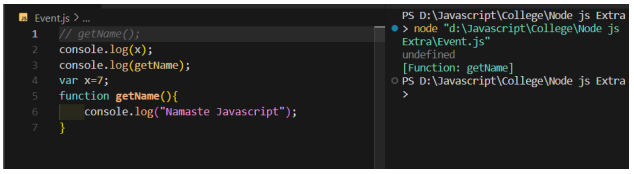
1] Loading Data
2] Collecting Data
Error:Not fullfilled!

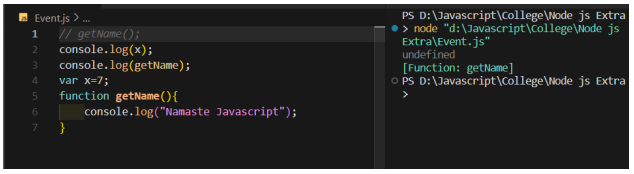
## ▼ Hoisting (Variables & Functions)

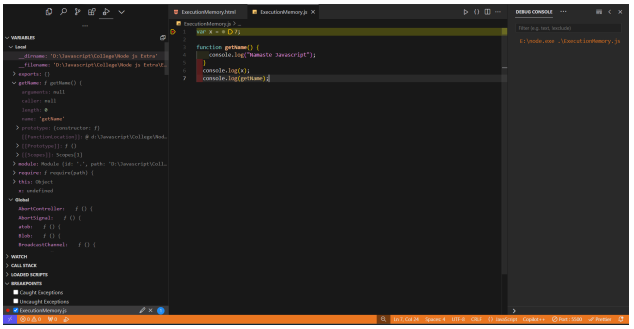
```
getName();
console.log(x);
console.log(getName);
var x=7;
function getName(){
    console.log("Namaste Javascript");
}
```




```
// getName();
console.log(x);
console.log(getName);
var x=7;
function getName(){
    console.log("Namaste Javascript");
}
```

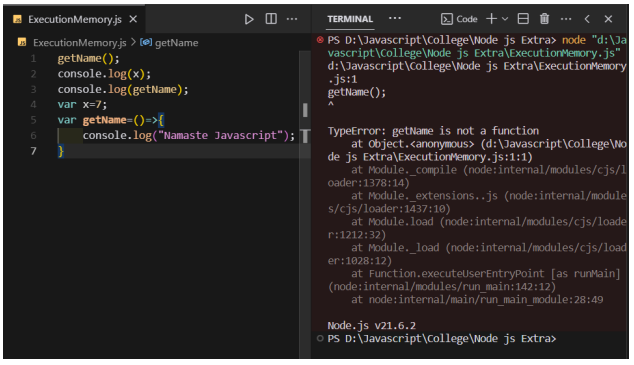


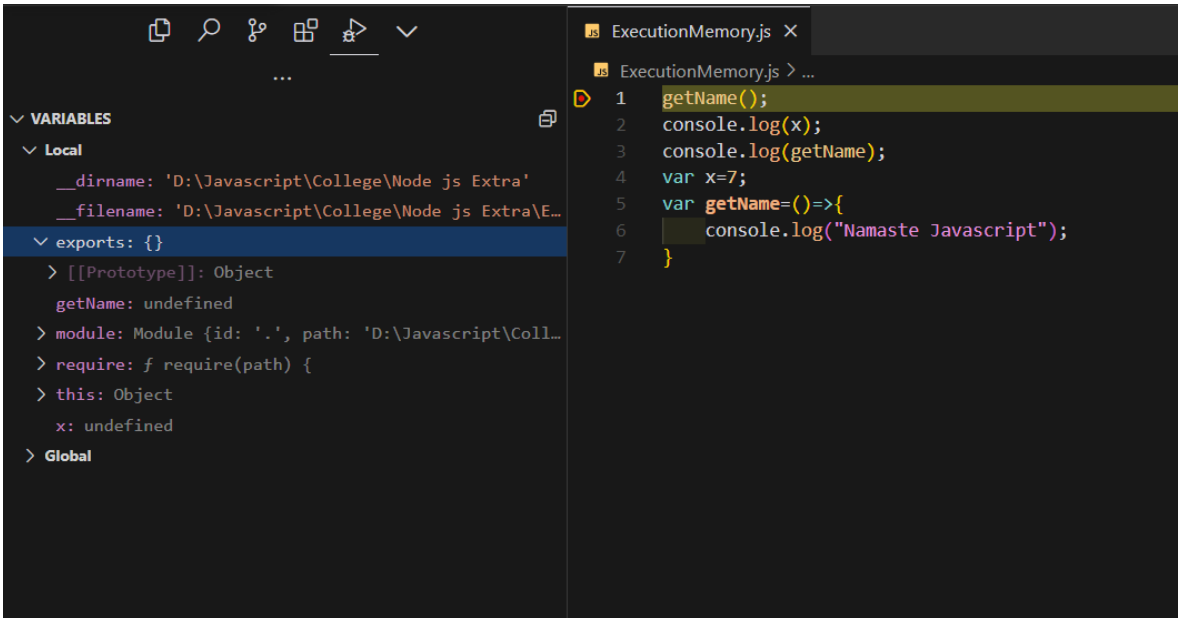



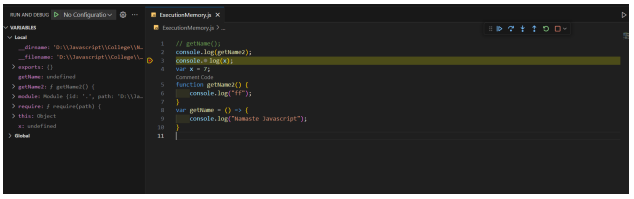


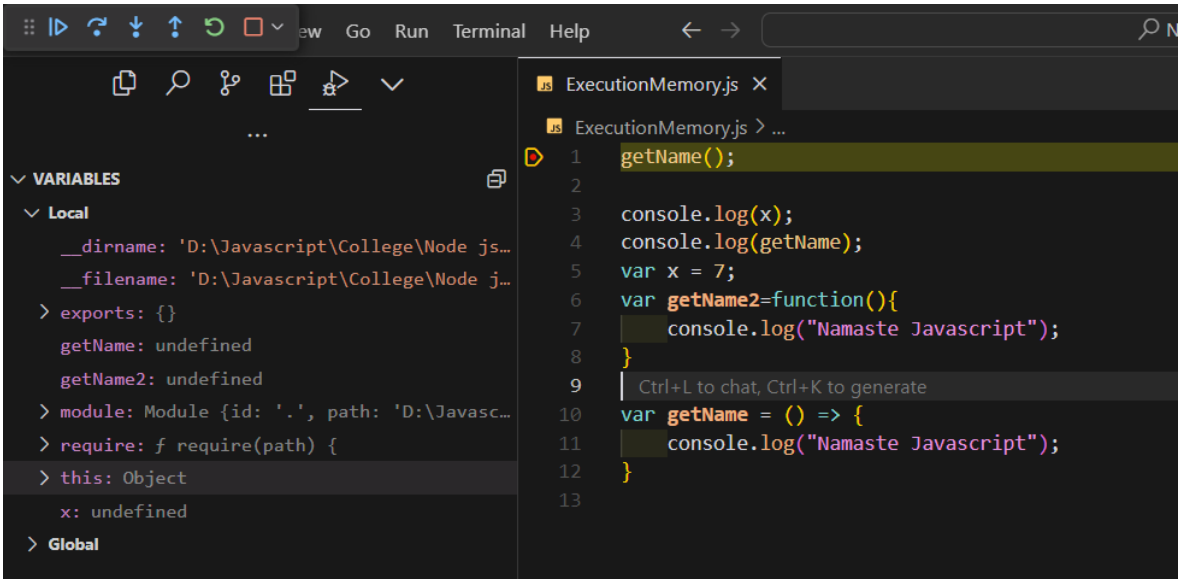
```javascript
getName();
console.log(x);
console.log(getName);
var x=7;
var getName=()=>{
    console.log("Namaste Javascript");
}
```

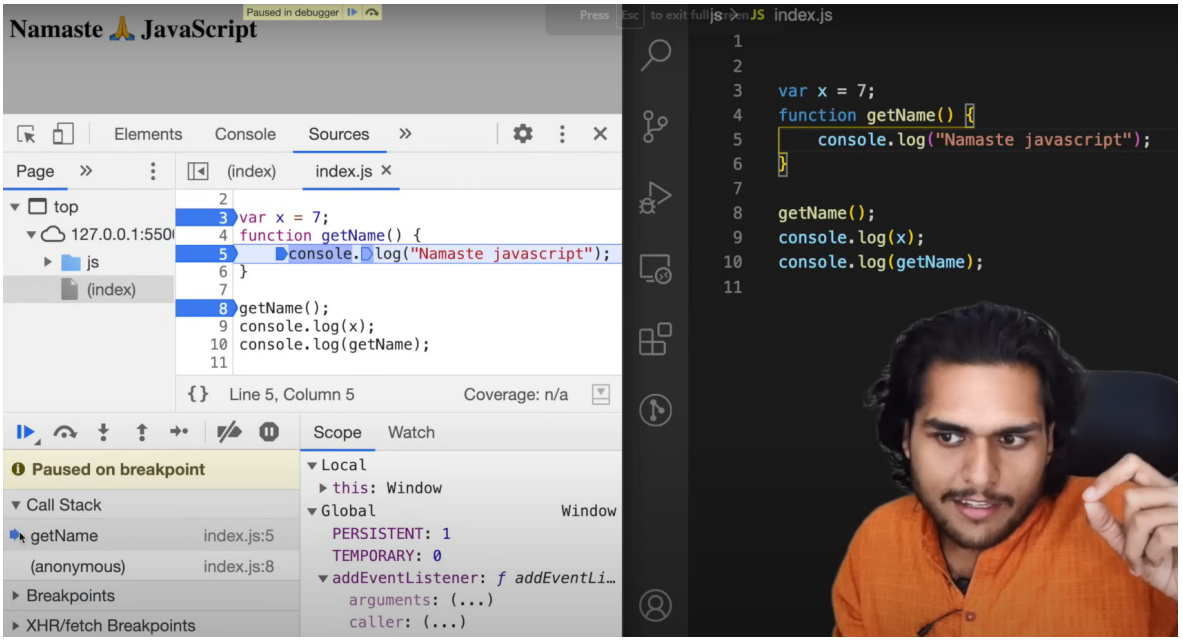
```javascript
getName();

console.log(x);
console.log(getName);
var x = 7;
var getName2=function(){
    console.log("Namaste Javascript");
}

var getName = () => {
    console.log("Namaste Javascript");
}
```

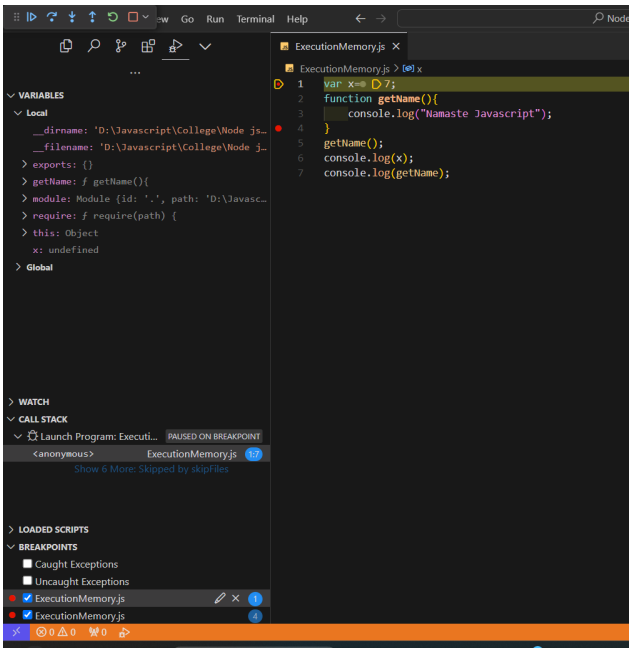


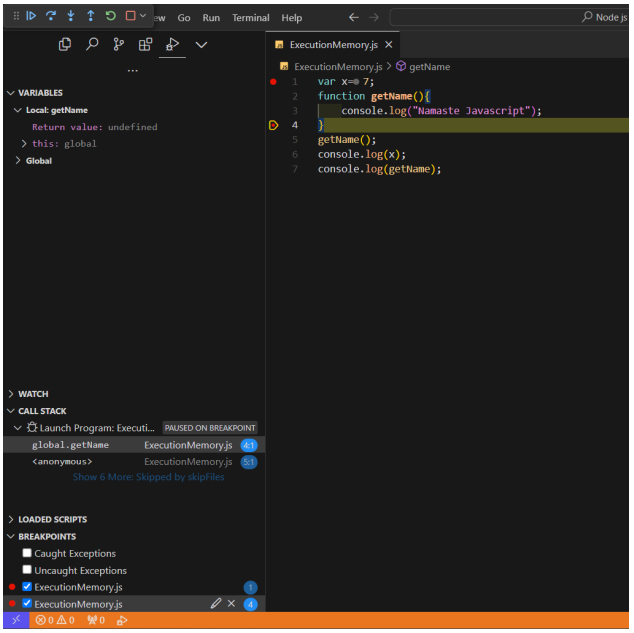

VARIABLES
Local
__dirname: 'D:\Javascript\College\Node js Extra'
__filename: 'D:\Javascript\College\Node js Extra\E…
exports: {}
[[Prototype]]: Object
getName: undefined
module: Module {id: '.', path: 'D:\Javascript\Coll…
require: ƒ require(path) {
this: Object
x: undefined
Global
ExecutionMemory.js
ExecutionMemory.js > ...
getName();
console.log(x);
console.log(getName);
var x=7;
var getName=()=>{
console.log("Namaste Javascript");
}



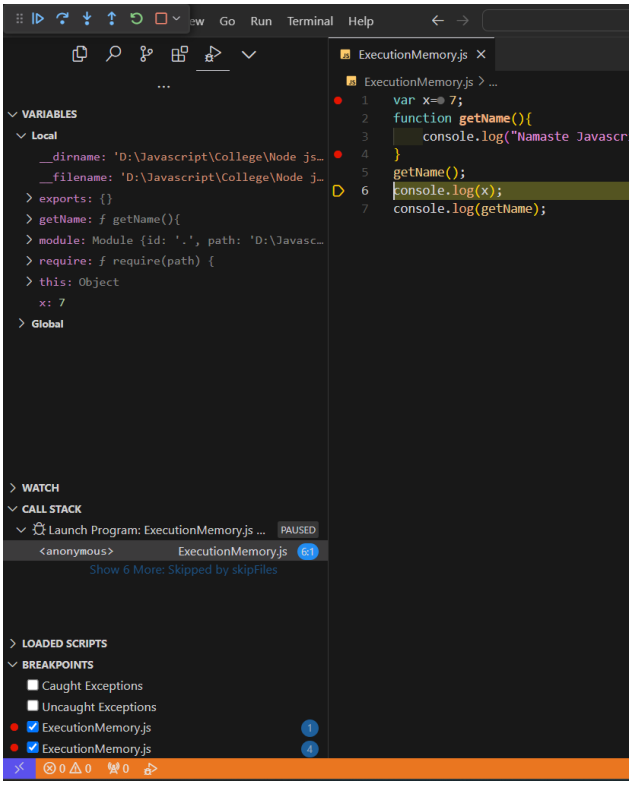

Go
Run
Terminal
Help
VARIABLES
Local
__dirname: 'D:\Javascript\College\Node js…
__filename: 'D:\Javascript\College\Node j…
exports: {}
getName: undefined
getName2: undefined
module: Module {id: '.', path: 'D:\Javasc…
require: ƒ require(path) {
this: Object
x: undefined
Global
ExecutionMemory.js
ExecutionMemory.js > ...
getName();
console.log(x);
console.log(getName);
var x = 7;
var getName2=function(){
console.log("Namaste Javascript");
}
Ctrl+L to chat, Ctrl+K to generate
var getName = () => {
console.log("Namaste Javascript");
}


Namaste 🙏 JavaScript
Paused in debugger
Elements
Console
Sources
Page
(index)
index.js
top
127.0.0.1:550
js
(index)
var x = 7;
function getName() {
console.log("Namaste javascript");
}
getName();
console.log(x);
console.log(getName);
Line 5, Column 5
Coverage: n/a
Scope
Watch
Paused on breakpoint
Call Stack
getName
index.js:5
(anonymous)
index.js:8
Breakpoints
XHR/fetch Breakpoints
Local
this: Window
Global
Window
PERSISTENT: 1
TEMPORARY: 0
addEventListener: ƒ addEventLi…
arguments: (...)
caller: (...)
index.js
var x = 7;
function getName() {
console.log("Namaste javascript");
}
getName();
console.log(x);
console.log(getName);

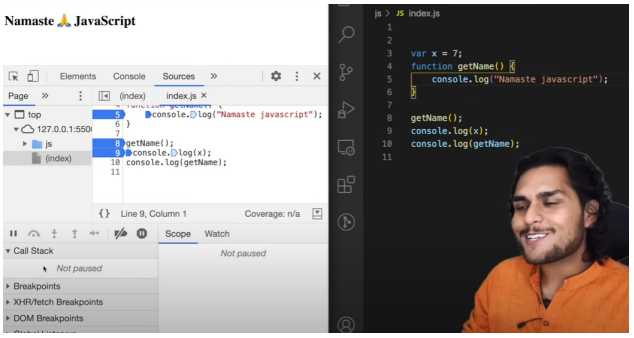

Go
Run
Terminal
Help
ExecutionMemory.js
VARIABLES
Local
__dirname: 'D:\Javascript\College\Node js…
__filename: 'D:\Javascript\College\Node j…
exports: {}
getName: ƒ getName(){
module: Module {id: '.', path: 'D:\Javasc…
require: ƒ require(path) {
this: Object
x: undefined
Global
WATCH
CALL STACK
Launch Program: Executi…
PAUSED ON BREAKPOINT
<anonymous>
ExecutionMemory.js
Show 6 More: Skipped by skipFiles
LOADED SCRIPTS
BREAKPOINTS
Caught Exceptions
Uncaught Exceptions
ExecutionMemory.js
var x = 7;
function getName(){
console.log("Namaste Javascript");
}
getName();
console.log(x);
console.log(getName);




After Cokmpletion getName function Execution Context is vanished.




After Full code Execution Call Stack is empty now.




## ▼ Sorting

Absolutely, let's delve into the `sort()` method in JavaScript to clarify its behavior and the importance of using a compare function for numeric sorting.

### `Array.prototype.sort()`

The `sort()` method in JavaScript is used to sort the elements of an array in place and returns the sorted array. By default, `sort()` sorts elements as strings and arranges them based on their Unicode code points. This default behavior can lead to unexpected results when sorting numbers, especially when numeric values are mixed with strings.

### Default Behavior

When you apply `sort()` directly to an array of numbers or a mixed array (containing both numbers and strings), JavaScript sorts the elements as if they were strings. Here's an example with the array `[0, 11, -1, 4, 1, 31, 45, -1, 2, 5, 8, 9, 10]`:

```
let a = [0, 11, -1, 4, 1, 31, 45, -1, 2, 5, 8, 9, 10];
console.log(a.sort());
```

### Output Explanation

The output of `console.log(a.sort());` can be:

```
[-1, -1, 0, 1, 10, 11, 2, 31, 4, 45, 5, 8, 9]
```

- **Lexicographical Sorting**: Numbers are sorted as strings based on their Unicode code points. For instance, `1` comes before `0` because `-` (hyphen) has a lower Unicode code point than digits.
- **Inconsistent Numeric Order**: `10` appears before `2` because `1` (from `10`) is lexicographically less than `2`.

### Correct Approach for Numeric Sorting

To achieve numerical sorting, it's crucial to provide a compare function to `sort()`:

```
let a = [0, 11, -1, 4, 1, 31, 45, -1, 2, 5, 8, 9, 10];
a.sort((x, y) => x - y);
console.log(a);
```

### Output with Numeric Sorting

With the compare function `(x, y) => x - y`, the output will correctly sort the array numerically:

```
[-1, -1, 0, 1, 2, 4, 5, 8, 9, 10, 11, 31, 45]
```

- **Compare Function Explanation**: The expression `x - y` inside the compare function subtracts `y` from `x`. If the result is negative, `x` is sorted before `y`, ensuring ascending numerical order.
- **Expected Order**: Now, `1` appears before `0`, and numbers are sorted correctly without inconsistencies like `10` appearing before `2`.


Khushal Savalakha

## Conclusion

Always use a compare function `(x, y) => x - y` when sorting arrays of numbers to ensure they are sorted numerically and in the desired order. This practice avoids unexpected sorting results caused by the default lexicographical sorting behavior of `sort()`. Understanding how `sort()` works with strings versus numbers is fundamental for effectively manipulating arrays in JavaScript.

> 💡 here how x,y takes values ?

In the context of the compare function `(x, y) => x - y` used with `array.sort()`, `x` and `y` represent two elements of the array being compared during the sorting process. Here's a breakdown of how `x` and `y` take values:

## Sorting Mechanism

1. **Compare Function Execution**:
   - When you call `array.sort(compareFunction)`, JavaScript internally executes the `compareFunction` for pairs of elements in the array to determine their order.
2. **Parameters `x` and `y`**:
   - During each comparison, the `sort()` method passes two elements from the array to the `compareFunction`.
   - `x` represents the first element in the comparison.
   - `y` represents the second element in the comparison.
3. **Sorting Logic**:
   - The `compareFunction` determines the sorting order based on the return value:
     - If `compareFunction(x, y)` returns a value less than `0`, `x` is placed before `y`.
     - If `compareFunction(x, y)` returns `0`, `x` and `y` are considered equal in terms of sorting order (though their original order might still influence the final sorted order).
     - If `compareFunction(x, y)` returns a value greater than `0`, `y` is placed before `x`.
4. **Numeric Sorting Example**:
   - For numeric sorting (`(x, y) => x - y`):
     - When comparing `x` and `y`, `x - y` calculates the difference between `x` and `y`.
     - If the result is negative, `x` is sorted before `y`.
     - If the result is positive, `y` is sorted before `x`.
     - If the result is `0`, `x` and `y` are considered equal in numerical value.
5. **Usage in Context**:
   - In the context of `array.sort((x, y) => x - y)`, each pair of elements (`x` and `y`) is compared based on their numeric values.
   - This ensures that the array is sorted in ascending numerical order.

## Example Walkthrough

Given the array:

```
let numbers = [5, 2, 8, 1, 4];
```

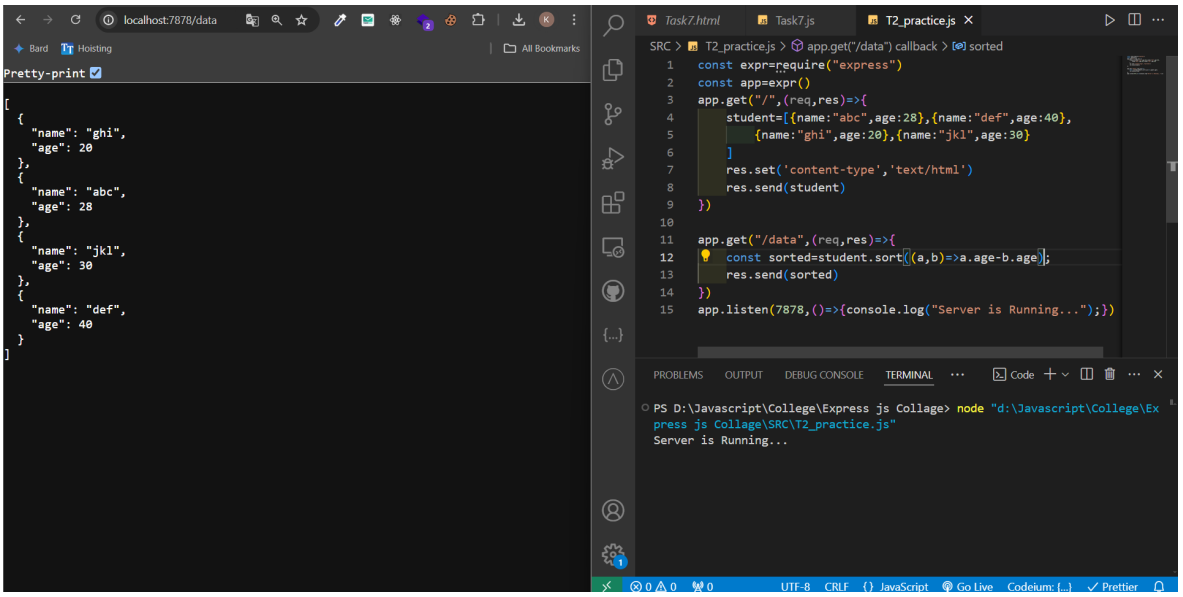
- During the sorting process, the `sort()` method might compare pairs like `(5, 2)`, `(2, 8)`, `(8, 1)`, and so on.
- For each comparison, the `compareFunction` `(x, y) => x - y` computes the difference between `x` and `y`.
- Based on the computed difference, `sort()` rearranges the elements to achieve the desired order.

## Conclusion

Understanding how `x` and `y` take values in the context of the `compareFunction` `(x, y) => x - y` is essential for effectively using `array.sort()` for numeric sorting in JavaScript. This approach ensures clarity and precision in sorting arrays based on specific criteria, such as numerical order in this case.


Khushal Savalakha

```javascript
const expr=require("express")
const app=expr()
app.get("/",(req,res)=>{
    student=[{name:"abc",age:28},{name:"def",age:40},
        {name:"ghi",age:20},{name:"jkl",age:30}
    ]
    res.set('content-type','text/html')
    res.send(student)
})

app.get("/data",(req,res)=>{
    const sorted=student.sort((a,b)=>a.age-b.age);
    res.send(sorted)
})
app.listen(7878,()=>{console.log("Server is Running...");})
```





Khushal Savalakha