JavaScript

changing htlm content

document.querySelector("h1").textContent="hello world";

toggle text

```
const listedItem= document.querySelectorAll("li");
    function toggleDone(e){
      if(!e.target.className){
        e.target.className="done";
      }
      else{
        e.target.className="";
      }

    }
    listedItem.forEach((item) => {
      item.addEventListener("click",toggleDone);
    });
```

toggle image

```
const myImage= document.querySelector("img");
    myImage.addEventListener("click", () => {
      const mySrc= myImage.getAttribute("src");

      if (mySrc === "P:/calibraint/pic1.jpg"){
        myImage.setAttribute("src","P:/calibraint/pic2.jpg");
      }
      else{
        myImage.setAttribute("src","P:/calibraint/pic1.jpg");
    }})
```

ES6 features

1. let

- Introduced in **ES6** (2015).
- **Block-scoped** (only works inside the {} where it's declared).
- Can be **updated**, but **not redeclared** in the same scope.

```
let name = "Deepthi";
name = "Sai";       // ✅ Allowed
let name = "New";    // ❌ Error (redeclaration in same block)
```

2. const

- Also **block-scoped**.
- Must be **initialized** at the time of declaration.
- **Cannot be updated or redeclared**.
- But if it holds an **object or array**, the **contents can be changed**.

```
const age = 25;
age = 30;           // ❌ Error

const person = { name: "Sai" };
person.name = "Deepthi"; // ✅ Allowed (modifying property)
```

3. var (Old way)

- **Function-scoped**, not block-scoped.
- Can be **updated and redeclared**.
- Gets **hoisted** (moved to the top of its scope), but initialized as u**ndefined**.

```
var city = "Hyderabad";
var city = "Delhi";   // ✅ Allowed

console.log(x);      // undefined
var x = 10;
```

2.Arrow function

Why Arrow Functions Are Recommended Over Regular Functions:

- You can write short functions in a more straightforward manner
- For single-line functions, the return statement can be implicit
- The this keyword is not bound to the function.

Why Use Arrow Functions?

1. Shorter & Cleaner Syntax

Arrow functions are concise — great for short, inline functions.

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

// Arrow function
const add = (a, b) => a + b;
```

2. No this Binding (Lexical this)

In arrow functions, this refers to the **surrounding (outer) scope** — it doesn't change.

```
const person = {
  name: "Deepthi",
  greet: function () {
    setTimeout(() => {
      console.log("Hello, I am " + this.name); // ✅ Arrow function keeps 'this'
    }, 1000);
  }
};

person.greet(); // Hello, I am Deepthi
```

If you used a normal function inside setTimeout, this.name would be undefined unless manually bound.

but if we use normal function ,

```
class Person {
name = "Deepthi";
greet() {
setTimeout(function () {
console.log("Hello, I am " + this.name);
}, 1000);
}
}
const person = new Person();
person.greet();
```

3. Perfect for Callbacks

Arrow functions are ideal for short functions inside .map(), .filter(), .foreach()

```
const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2); // ✅ Arrow = cleaner
```

>call back

A **callback** is a function passed **as an argument** to another function, and then **called inside** that function.

> Think of it like: "I'll call you back when I'm done."

Real-life Analogy:

> You order food at a restaurant and give them your number.When your food is ready, they call you back.In JavaScript, that “call you back” is a callback function.

```
function orderPizza(pizzaType, callback) {
  console.log(`Ordering a ${pizzaType} pizza...`);

  setTimeout(() => {
    console.log(`${pizzaType} pizza is ready!`);
    callback();
  }, 2000);
}

function deliverPizza() {
  console.log("Pizza delivered to your home!");
}
orderPizza("Margherita", deliverPizza);
```

another example....

```
function sai(a,callback){
console.log("obtaining your age")
setTimeout(()=>{
console.log("analyzing...")
callback(a)
},2000)
}
function vote(age){
if(age>=18){
console.log("hurrayy...! you are eligible..")
}
else{
console.log("poo..padiii...")
}
}
sai(16,vote)
//obtaining your age
```

Examples:

extracting data from a text field using enter key

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>form</title>
</head>
<body>
    <form action="_blank">
        <textarea name="name" id="a"></textarea>

    </form>
    <script>

        document.getElementById("a").addEventListener("keydown", function(event){
            if(event.key === "Enter")
           {
            const val=document.getElementById("a").value
            console.log(val)
           }
        })

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

extract data from a textfield using submit button

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Form</title>
</head>
<body>
    <form action="_blank">
        <textarea name="name" id="a"></textarea><br><br>
        <button type="button" onclick="getText()">Submit</button>
    </form>
<script>
        function getText() {
            const b = document.getElementById("a").value;
            console.log( b);
        }
    </script>
</body>
</html>
```

extracting data and printing it in a card

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

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

</head>
 <style>
  .cards {
   display: flex;
   flex-wrap: wrap;
   gap: 10px;
   margin-top: 20px;
  }

  .card {
   padding: 15px;
   background-color: #eda706;
   width: fit-content;
   min-width: 100px;
  }
 </style>
<body>
  <form action="_blank">
    <textarea name="name" id="a"></textarea>

  </form>

    <div class="cards" id="container"></div>


  <script>

    document.getElementById("a").addEventListener("keydown", function(event){
      if(event.key === "Enter")
     {
      event.preventDefault();
      const val=document.getElementById("a").value;
      if(val!==""){
        const card=document.createElement("div")
        card.className="card"
        card.textContent= val;
        document.getElementById("container").appendChild(card)
        document.getElementById("a").value="";
      }

     }
    })

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

extracting data and printing it in a line

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>form</title>
</head>
<body>
    <form action="_blank">
        <textarea name="name" id="a"></textarea>

    </form>
    <div id="output"></div>
    <script>

        document.getElementById("a").addEventListener("keydown", function(event){
            if(event.key === "Enter")
           {
            event.preventDefault();
            const val=document.getElementById("a").value;
            if(val!==""){
                const line=document.createElement("div")
                line.textContent= val;
                output.appendChild(line);
                document.getElementById("a").value="";
            }

           }
        })

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

3. Classes

blueprint for creating objects, offering a more structured and object-oriented way to define and create reusable components.

class expression

class expressions may be anonymous, or have a name that's different from the variable that it's assigned to.

```
const Bike = class {
  constructor(brand) {
    this.brand = brand;
  }

  ride() {
    console.log(`${this.brand} bike is riding.`);
```

class declaration

A **class declaration** uses the class keyword followed by the class name. It's **hoisted**, meaning you can use it *after* it's defined in the code

```
class Car {
  constructor(brand) {
    this.brand = brand;
  }

  drive() {
  }
};

const myBike = new Bike("Yamaha");
myBike.ride();
```

real-world Scenario:

while building a **chat app** (like WhatsApp or Slack). Users can send messages in different formats (text, image, code). You dynamically return different message handler classes using **class expressions**.

```
function getMessageHandler(type) {
  if (type === "text") {
    return class {
      send(message) {
        console.log(`Sending text: ${message}`);
      }
    };
  } else if (type === "image") {
    return class {
      send(imageUrl) {
        console.log(`Sending image: ${imageUrl}`);
      }
    };
  }
}

const TextMessage = getMessageHandler("text");
const imgMessage = getMessageHandler("image");

const t = new TextMessage();
t.send("Hello!");
const img = new imgMessage();
img.send("http://image.jpg");
```

```
    console.log(`${this.brand} is driving.`);
  }
}

const myCar = new Car("Tesla");
myCar.drive();
```

real-world Scenario

while building a **ride-booking web app** (like Uber). You define base vehicle classes using **class declarations** because they are needed across the entire app in different modules (drivers, admin panel, maps).

```

class Vehicle {
  constructor(type, brand) {
    this.type = type;
    this.brand = brand;
  }

  start() {
    console.log(`${this.brand} ${this.type} is starting...`);
  }
}

export default Vehicle;

import Vehicle from './vehicle.js';

const taxi = new Vehicle("car", "Toyota");
taxi.start();
```

examples

```
class try {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
```

```
const person1 = new try("Deepthi", 22);
person1.greet();
```

creating a class

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

static method

associated with the class, but not with any particular object of the class.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }

  static generateName() {
    const names = ['sai', 'me', 'Bob'];
    const index = Math.floor(Math.random() * names.length);
    return names[index];
  }
}
const name = Person.generateName();
console.log(name);
```

inheritance class

ay to extend the functionality of a class by creating a new class that inherits from the original class.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

class Student extends Person {
  constructor(name, age, grade) {
    super(name, age);
    this.grade = grade;
  }
}

const student1 = new Student('me', 30, 'A');
console.log(student1);
```