JS TOPICS

Here is the revised content without timestamps:

Chapter 1: Data Types

1.1 Adding JavaScript
1.2 Variables
1.3 Data Types
1.4 Strings
1.5 String Methods
1.6 Numbers
1.7 Loose Equality vs. Strict Equality
1.8 Type Conversion
1.9 Arrays
1.10 Boolean Values and Comparison Operators

Chapter 2: Control Flow

2.1 For Loop
2.2 While Loop
2.3 Do While Loop
2.4 if / else if / else
2.5 Nested if Statement
2.6 Break and Continue
2.7 Logical Operators
2.8 Variable and Block Scope
2.9 Ternary Operator
2.10 Switch Statement

Chapter 3: Functions

3.1 Function and Function Expression
3.2 Function Parameters and Arguments
3.3 Return Statement
3.4 Arrow Function
3.5 Higher Order Function - Callbacks
3.6 Higher Order Function - Returning Functions
3.7 IIFE
3.8 setTimeout and setInterval
3.9 Hoisting

Chapter 4: Objects

4.1 Object Introduction
4.2 Function vs. Methods
4.3 "this" Keyword
4.4 for-Each Method
4.5 Objects Inside Array
4.6 Math Object
4.7 Call, Apply, and Bind
4.8 Pass by Value and Pass by Reference
4.9 for-in Loop

Chapter 5: DOM

5.1 DOM Introduction
5.2 Query Selector
5.3 Other Ways to Access Elements

Chapter 6: DOM - Forms

6.1 Submit Event
6.2 Regular Expression

Chapter 7: Array Methods

7.0 Array Methods
7.1 Slice
7.2 Splice
7.3 At
7.4 Map
7.5 Filter

Chapter 8: Date

8.1 Date and Time

Chapter 9: LocalStorage

9.1 Local Storage Introduction

Chapter 10: OOP

10.1 Constructor and `new` Operator
10.2 Prototypes
10.3 Prototypical Inheritance
10.5 Setters and Getters

10.6 Static Methods
10.7 Class Inheritance

Chapter 11: Async JavaScript

11.4 Callback Function
11.12 Async/Await without Return Statement
11.13 Error Handling Using Try-Catch

Chapter 12: ES6

12.1 Array De-structuring
12.7 for-of Loop

Chapter 13: Modern Tooling

13.1 Importing and Exporting ES6 Modules
13.4 Introduction to NPM

Chapter 14: Advanced Concepts

14.1 Closures
14.1 BOOTStrap

Introduction to JavaScript

JavaScript (JS) is a high-level, dynamic, and versatile programming language primarily used for creating interactive and dynamic content on websites. It's a core technology of the World Wide Web, alongside HTML and CSS. JavaScript enables features like animations, form validations, interactive maps, and more.

History of JavaScript

- **Inventor**: Brendan Eich, a programmer at Netscape, created JavaScript in just 10 days in 1995.
- **Original Name**: JavaScript was initially called **Mocha**, then **LiveScript**, before being renamed JavaScript.
- **Relation to Java**: The name "JavaScript" was chosen as a marketing strategy due to Java's popularity at the time, but the two languages are entirely different.
- **Standardization**: In 1997, JavaScript was standardized under the name **ECMAScript (ES)** by ECMA International.

Key milestones:

- **1995**: JavaScript introduced in Netscape Navigator.
- **1997**: ECMAScript 1 released.
- **2015**: ECMAScript 6 (ES6) introduced modern features like `let`, `const`, arrow functions, and more.

Developer of JavaScript

- Brendan Eich, working at **Netscape Communications**, developed JavaScript to enhance web interactivity.
- Today, JavaScript is maintained by various organizations, including **TC39**, a committee under ECMA International.

JavaScript Keywords

Keywords are reserved words in JavaScript that have predefined meanings and cannot be used as variable names.

Examples:

- **Control flow keywords**: `if`, `else`, `switch`, `case`, `default`, `break`, `continue`.
- **Variable declarations**: `var`, `let`, `const`.
- **Functions and loops**: `function`, `return`, `for`, `while`, `do`.
- **Special purpose**: `this`, `null`, `undefined`, `new`, `delete`, `typeof`, `instanceof`.

File Name for JavaScript

- JavaScript files have the extension **`.js`** (e.g., `script.js`).
- These files can be linked to an HTML file using the `<script>` tag:

```
<script src="script.js"></script>
```

Literal in JavaScript

Literals are fixed values directly used in the code.
Examples of different types of literals:

- **Numeric**: `10`, `3.14`.
- **String**: `"Hello"`, `'World'`.
- **Boolean**: `true`, `false`.

- **Object Literal**:

```
const person = { name: "John", age: 30 };
```

Constant in JavaScript

- Declared using the `const` keyword.
- The value of a constant cannot be reassigned after its initial definition.

- Syntax:--→

```
const PI = 3.14;
console.log(PI); // Outputs: 3.14
```

- **Features**:

- Must be initialized during declaration.
- Useful for values that should not change, like configuration settings or fixed mathematical values.

Chapter 1: Data Types

1.1 Adding JavaScript

Theory: JavaScript can be added to HTML documents in three primary ways:

1. **Inline JavaScript**: Directly within HTML elements using the ((*onclick)),* (*onload),* and other event attributes.
2. **Internal JavaScript**: Within the <script> tag inside the HTML document.
3. **External JavaScript**: In separate .js files linked to the HTML document using the <script src="script.js"></script> tag.

Examples:

- **Inline JavaScript**:

```
<button onclick="alert('Hello, World!')">Click
                    Me</button>
```

- **Internal JavaScript:**

Html

```html
<!DOCTYPE html>
<html>
<head>
    <title>Internal JavaScript Example</title>
    <script>
        function showAlert() {
            alert('Hello, World!');
        }
    </script>
</head>
<body>
    <button onclick="showAlert()">Click Me</button>
</body>
</html>
```

- **External JavaScript:**

Html

```html
<!DOCTYPE html>
<html>
<head>
    <title>External JavaScript Example</title>
    <script src="script.js"></script>
</head>
<body>
    <button onclick="showAlert()">Click Me</button>
</body>
</html>
```

script.js:

Js

```js
function showAlert() {
    alert('Hello, World!');
}
```

1.2 Variables

Notes:

- Variables store data and can be declared using `var`, `let`, or `const`.
- `let` and `const` are block-scoped (introduced in ES6), while `var` is function-scoped.

Code and Explanation:

1. **Declaring Variables**:

```
let name = "Alice"; // Modern way
const PI = 3.14;    // Constant value
var age = 25;       // Older way
console.log(name, age, PI);
```

2. **Differences**:
   - `let`: Can be reassigned but not redeclared within the same block.
   - `const`: Cannot be reassigned.
   - `var`: Can be reassigned and redeclared, but it's less preferred due to potential bugs.

1.3 Data Types

Notes:

- JavaScript has **primitive types** and **non-primitive types**:
  - **Primitive**: `string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`.
  - **Non-primitive**: `object` (includes arrays, functions, etc.).

Code and Explanation:

```
let aString = "Hello"; // String
let aNumber = 42;      // Number
let aBoolean = true;   // Boolean
let anUndefined;       // Undefined
let aNull = null;      // Null
let anObject = { name: "Alice" }; // Object

console.log(typeof aString);    // "string"
console.log(typeof aNumber);    // "number"
console.log(typeof anObject);   // "object"
console.log(typeof anUndefined); // "undefined"
```

Chapter 3: Functions

Functions are a fundamental building block in JavaScript. They are reusable blocks of code designed to perform specific tasks.

3.1 Function and Function Expression

Theory:

- **Function Declaration**: A standard way to define a function using the `function` keyword.
- **Function Expression**: Assigns a function (can be anonymous) to a variable.

Code:

```
// Function Declaration
function greet() {
  console.log("Hello, World!");
}
greet(); // Output: Hello, World!

// Function Expression
const sayHello = function () {
  console.log("Hello from Function Expression!");
};
sayHello(); // Output: Hello from Function Expression
```

- **Key Difference**: Function declarations are **hoisted**, meaning they can be called before they are defined. Function expressions are not.

3.2 Function Parameters and Arguments

Theory:

- **Parameters**: Variables listed in the function definition.
- **Arguments**: Values passed to the function when it's invoked.
- Functions can have **default parameters**, which assign a default value if no argument is provided.

Code:

```
// Function with parameters
function multiply(a, b = 1) { // b has a default value of 1
  return a * b;
}

console.log(multiply(5, 10)); // Output: 50
console.log(multiply(5));    // Output: 5 (b defaults to 1)
```

3.3 Return Statement

Theory:

- The `return` statement sends a value back to the function caller and terminates the function execution.

Code:

```
function add(a, b) {
  return a + b; // Returns the sum of a and b
}
const result = add(3, 7);
console.log(result); // Output: 10

function noReturn() {
  console.log("No return here!");
}
console.log(noReturn()); // Output: "No return here!" followed by undefined
```

3.4 Arrow Function

Theory:

- Arrow functions, introduced in ES6, provide a shorter syntax.
- Arrow functions do not have their own `this` context, making them unsuitable for methods.

Code:

```
// Regular Function
const square = function (x) {
  return x * x;
};

// Arrow Function
const squareArrow = (x) => x * x;

console.log(square(4));      // Output: 16
console.log(squareArrow(4)); // Output: 16

// Without parameters
const greet = () => console.log("Hello!");
greet(); // Output: Hello!
```

3.5 Higher-Order Function - Callbacks

Theory:

- A Higher-Order Function takes another function as an argument.
- A Callback Function is a function passed into another function as an argument.

Code:

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}

function executeCallback(callback, arg) {
  callback(arg); // Execute the passed function
}

executeCallback(greet, "Alice"); // Output: Hello, Alice!
```

3.7 IIFE (Immediately Invoked Function Expression)

Theory:

- IIFE executes immediately after it's defined.
- Useful for creating a private scope to avoid polluting the global namespace.

Code:

```
 (function () {
  console.log("This is an IIFE!");
})();        // Output: This is an IIFE!

                      // With parameters
(function (name) {
  console.log(`Hello, ${name}!`);
})("Alice");          // Output: Hello, Alice!
```

3.8 setTimeout and setInterval

setTimeout Theory:

- Executes a function after a specified delay (in milliseconds).

setTimeout Code:

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

setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);

console.log("End");

// Output:
// Start
// End
// Executed after 2 seconds
```

setInterval Theory:

- Repeatedly executes a function at specified intervals.

setInterval Code:

```
let counter = 0;

const intervalId = setInterval(() => {
  console.log(`Counter: ${++counter}`);
  if (counter === 5) clearInterval(intervalId);  // Stops after 5 iterations
}, 1000);


// Output:
// Counter: 1
// Counter: 2
// Counter: 3
// Counter: 4
// Counter: 5
```

3.9 Hoisting

Theory:

- **Hoisting** moves function declarations and variable declarations to the top of their scope during the compilation phase.
- Function expressions and variables using `let` or `const` are not hoisted.

Code:

```
// Function Declaration Hoisting
console.log(hello()); // Output: "Hello!"

function hello() {

  return "Hello!";
```

```
}

// Function Expression (Not Hoisted)
console.log(sayHi);  // Output: undefined

var sayHi = function () {
  return "Hi!";
};

// let and const Hoisting
let myVar = 10;

console.log(myVar); // ReferenceError: Cannot access 'myVar' before
initialization
```

Summary:

- Functions are reusable blocks of code.
- **Function Declarations** are hoisted, while **Function Expressions** are not.
- Arrow functions offer concise syntax and do not have their own `this`.
- Higher-order functions allow for dynamic programming through callbacks and function returns.
- Tools like **setTimeout** and **setInterval** enable asynchronous code execution.
- Hoisting affects how and when variables and functions are initialized.

Chapter 4: Objects

In JavaScript, **objects** are a collection of key-value pairs. They can store multiple values of different types and are essential for working with structured data. Objects are used extensively in JavaScript to represent real-world entities and more.

4.1 Object Introduction

Theory:

- Objects are a collection of properties (key-value pairs).
- Keys are strings (or symbols), and values can be any valid data type: primitive values, functions, or even other objects.

Code:

```
// Creating an object

const person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log("Hello, " + this.name);
  }
};
```

```
console.log(person.name); // Output: John
console.log(person.age);  // Output: 30
person.greet();           // Output: Hello, John
```

4.2 Function vs. Methods

Theory:

- A **function** is a block of code designed to perform a task.
- A **method** is a function that is a property of an object.
- Functions can exist independently, while methods are associated with specific objects.

Code:

```
// Function

function greet() {

  console.log("Hello!");
}

greet(); // Output: Hello!

                    // Object with Method

const person = {
  name: "Alice",
  greet: function () {
    console.log("Hello, " + this.name);
  }
};
person.greet(); // Output: Hello, Alice
```

4.3 "this" Keyword

Theory:

- The `this` keyword refers to the context of the function where it is called.
- In a regular function, `this` refers to the object from which the function was called.
- In an arrow function, `this` is lexically inherited from the surrounding context.

Code:

```
// Regular function
const person = {
```

```
  name: "Alice",
  greet: function () {
    console.log("Hello, " + this.name); // 'this' refers to the person object
  }
};
person.greet(); // Output: Hello, Alice

// Arrow function
const personArrow = {
  name: "Bob",
  greet: () => {
    console.log("Hello, " + this.name); // 'this' refers to the outer scope,
not the personArrow object
  }
};
personArrow.greet();

// Output: Hello, undefined (as 'this' refers to the global context in non-
methods)
```

4.4 forEach Method

Theory:

- `forEach()` is an array method used to execute a provided function once for each element in the array.
- It does not return a value (undefined is returned).

Code:

```
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function (num) {
  console.log(num * 2);
});

// Output:
// 2
// 4
// 6
// 8
// 10
```

You can also use an arrow function:

```
numbers.forEach(num => console.log(num * 2));

// Output: 2, 4, 6, 8, 10
```

4.5 Objects Inside Array

Theory:

- You can store objects within an array.
- Arrays of objects are useful for organizing complex data structures like lists of people, products, etc.

Code:

```
const students = [
  { name: "John", age: 22 },
  { name: "Jane", age: 23 },
  { name: "Mike", age: 21 }
];

students.forEach(student => {

  console.log(`${student.name} is ${student.age} years old`);
});

// Output:
// John is 22 years old
// Jane is 23 years old
// Mike is 21 years old
```

4.6 Math Object

Theory:

- The **Math** object provides properties and methods for mathematical constants and functions.
- It is a built-in object and does not need to be instantiated.

Code:

```
console.log(Math.PI);         // Output: 3.141592653589793
console.log(Math.random());   // Output: A random number between 0 and 1
console.log(Math.pow(2, 3)); // Output: 8 (2 raised to the power of 3)

console.log(Math.sqrt(25));   // Output: 5 (square root of 25)
console.log(Math.round(4.5)); // Output: 5 (rounds to nearest integer)
```

4.7 Call, Apply, and Bind

Theory:

- These methods allow you to invoke functions with a specific `this` context.
  - **call()**: Invokes the function with a specified `this` value and arguments.
  - **apply()**: Similar to `call()`, but arguments are passed as an array.
  - **bind()**: Returns a new function with a specific `this` value, but does not invoke it immediately.

Code:

```
// Using call()
const person = { name: "Alice" };
function greet(greeting) {
  console.log(greeting + ", " + this.name);
}
greet.call(person, "Hello"); // Output: Hello, Alice

// Using apply()
greet.apply(person, ["Hi"]); // Output: Hi, Alice

// Using bind()
const greetPerson = greet.bind(person, "Hey");
greetPerson(); // Output: Hey, Alice
```

4.8 Pass by Value and Pass by Reference

Theory:

- **Pass by Value**: When primitive values (e.g., numbers, strings, booleans) are passed to a function, they are copied, and changes inside the function do not affect the original value.
- **Pass by Reference**: When objects are passed to a function, the reference (memory address) of the object is passed, meaning changes affect the original object.

Code:

```
// Pass by Value
let a = 10;
function changeValue(x) {
  x = 20;
}
changeValue(a);
console.log(a); // Output: 10 (the original value is unchanged)

// Pass by Reference
let person = { name: "John", age: 25 };
function changePerson(obj) {
  obj.age = 26;
}
changePerson(person);
console.log(person.age); // Output: 26 (the original object is modified)
```

4.9 for-in Loop

Theory:

- The `for-in` loop is used to iterate over the properties of an object.

- It loops through all enumerable properties of the object, including those that are inherited through the prototype chain.

Code:

```
const person = {
  name: "Alice",
  age: 25,
  greet: function () {
    console.log("Hello!");
  }
};

for (let key in person) {
  console.log(key + ": " + person[key]);
}
// Output:
// name: Alice
// age: 25
// greet: function() { console.log("Hello!"); }

// To avoid inherited properties, you can use hasOwnProperty()
for (let key in person) {

  if (person.hasOwnProperty(key)) {

    console.log(key + ": " + person[key]);
  }
}
```

Chapter 5: DOM (Document Object Model)

Chapter 5: DOM

5.1 DOM Introduction
5.2 Query Selector
5.3 Other Ways to Access Elements

The DOM (Document Object Model) is a programming interface for web documents. It represents the page so that programs can manipulate the structure, style, and content of web pages. The DOM represents a page as a tree

structure where each node is an object representing a part of the page.

5.1 DOM Introduction

Theory:

- The **DOM** allows JavaScript to access and update the content, structure, and style of HTML documents.
- JavaScript interacts with the DOM to dynamically update web pages without requiring a page reload.
- The DOM represents the HTML structure as a hierarchy of nodes (elements, attributes, text, etc.).

Code Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DOM Example</title>
</head>
<body>
  <div id="myDiv">Hello, DOM!</div>

  <script>

    // Accessing the DOM element

    const div = document.getElementById("myDiv");
    console.log(div.textContent); // Output: Hello, DOM!

    // Modifying the DOM element"

    div.textContent = "Hello, JavaScript!";

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

5.2 Query Selector

Theory:

- `querySelector()` is a method used to select an element from the DOM using CSS selectors.
- It returns the first matching element, or `null` if no matching element is found.

Code Example:

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

```
<body>

  <p class="message">This is a paragraph.</p>
  <p id="intro">Another paragraph.</p>

  <script>

    // Using querySelector to select by class

    const message = document.querySelector(".message");
    console.log(message.textContent); // Output: This is a paragraph.

    // Using querySelector to select by ID

    const intro = document.querySelector("#intro");
    console.log(intro.textContent); // Output: Another paragraph.
  </script>
</body>
</html>
```

Code Example:

```
html
Copy code
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DOM Example</title>
</head>
<body>
  <div id="myDiv">Hello, DOM!</div>

  <script>
    // Accessing the DOM element
    const div = document.getElementById("myDiv");
    console.log(div.textContent); // Output: Hello, DOM!

    // Modifying the DOM element
    div.textContent = "Hello, JavaScript!";
  </script>
</body>
</html>
```

5.2 Query Selector

Theory:

- `querySelector()` is a method used to select an element from the DOM using CSS selectors.
- It returns the first matching element, or `null` if no matching element is found.

Code Example:

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

```
  <title>Query Selector Example</title>
</head>
<body>
  <p class="message">This is a paragraph.</p>
  <p id="intro">Another paragraph.</p>

  <script>
    // Using querySelector to select by class
    const message = document.querySelector(".message");
    console.log(message.textContent); // Output: This is a paragraph.

    // Using querySelector to select by ID
    const intro = document.querySelector("#intro");
    console.log(intro.textContent); // Output: Another paragraph.
  </script>
</body>
</html>
```

5.3 Other Ways to Access Elements

Theory:

Besides `querySelector()`, there are other methods to access DOM elements:

- `getElementById()`: Selects an element by its ID.
- `getElementsByClassName()`: Selects elements by their class name.
- `getElementsByTagName()`: Selects elements by their tag name.

Code Example:

```
html
Copy code
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Access Methods</title>
</head>
<body>
  <div id="container">
    <p class="text">First paragraph.</p>
    <p class="text">Second paragraph.</p>
  </div>

  <script>
    // Using getElementById
    const container = document.getElementById("container");
    console.log(container);

    // Using getElementsByClassName (returns a live HTMLCollection)
    const paragraphs = document.getElementsByClassName("text");
    console.log(paragraphs[0].textContent); // Output: First paragraph.
  </script>
</body>
</html>
```

Chapter 4: Objects

Chapter 4: Objects

4.1 Object Introduction

Definition: An object is a collection of properties, where each property is defined as a key-value pair. Objects can store data and functions, making them essential for organizing complex data structures in JavaScript.

Code Example

```
let person = {
    name: "John",
    age: 30,
    greet: function() {
        console.log("Hello, " + this.name);
    }
};

person.greet(); // Output: Hello, John
```

4.2 Function vs. Methods

Definition: A function is a standalone block of code that can be executed independently, while a method is a function that is associated

with an object. Methods can manipulate the object's properties and provide behavior specific to that object.

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

let obj = {
    sayHello: function() {
        console.log("Hello from method!");
    }
};

sayHello(); // Output: Hello!
obj.sayHello(); // Output: Hello from method!
```

4.3 "this" Keyword

Definition: The `this` keyword refers to the current object in which the code is executing. Its value can change based on the context in which a function is called, making it crucial for object-oriented programming.

```
let car = {
    brand: "Toyota",
    getBrand: function() {
        return this.brand;
    }
};

console.log(car.getBrand()); // Output: Toyota
```

4.4 for-Each Method

Definition: The `forEach` method is an array method that executes a provided function once for each array element. It simplifies the process of iterating over arrays and is often preferred for its readability.

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(num) {
    console.log(num * 2);
});
// Output: 2, 4, 6, 8, 10
```

4.5 Objects Inside Array

Definition: Arrays can contain objects as their elements, allowing for the creation of complex data structures. This is useful for representing collections of related data, such as a list of students with their attributes.

```
let students = [
    { name: "Alice", age: 20 },
    { name: "Bob", age: 22 }
];

students.forEach(function(student) {
    console.log(student.name + " is " + student.age + " years old.");
});
// Output: Alice is 20 years old.
//         Bob is 22 years old.
```

4.6 Math Object

Definition: The Math object provides a set of mathematical functions and constants that are built into JavaScript. It allows developers to perform complex calculations without needing to implement these functions manually.

```
let radius = 5;
let area = Math.PI * Math.pow(radius, 2);
console.log("Area of the circle: " + area); // Output: Area of the ci
```

4.7 Call, Apply, and Bind

Definition: These methods allow you to control the value of `this` in functions. `call` and `apply` invoke a function with a specified `this` value, while `bind` creates a new function with a fixed `this` value, which can be called later.

```
function greet(greeting) {
    console.log(greeting + ", " + this.name);
}

let person = { name: "John" };

greet.call(person, "Hello"); // Output: Hello, John
greet.apply(person, ["Hi"]); // Output: Hi, John

let greetJohn = greet.bind(person);
greetJohn("Hey"); // Output: Hey, John
```

4.8 Pass by Value and Pass by Reference

Definition: In JavaScript, primitive data types (like numbers and strings) are passed by value, meaning a copy is made. In contrast, objects and arrays are passed by reference, allowing functions to modify the original object.

```
let num = 10;
function changeValue(n) {
    n = 20; // This does not change the original num
}
changeValue(num);
console.log(num); // Output: 10

let obj = { value: 10 };
function changeObject(o) {
    o.value = 20; // This changes the original object
}
changeObject(obj);
console.log(obj.value); // Output: 20
```

4.9 **The `for-in` loop** :

iterates over all enumerable properties of an object, including those inherited from its prototype chain.

It is important to use the **`hasOwnProperty`** method to filter out properties that are inherited from the prototype chain if you only want to work with the object's own properties.

```
let car = {
    brand: "Toyota",
    model: "Camry",
    year: 2020
};

// Using for-in loop to iterate over the properties of the car object
for (let key in car) {
    console.log(key + ": " + car[key]);
}
```

```
brand: Toyota
model: Camry
year: 2020
```

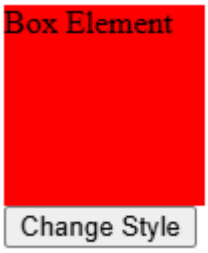
TASK →

Style color :

index.html styles.css script.js

```html
<!DOCTYPE html>
<html lang="en">
<body>
    <div id="box" style="width: 100px; height: 100px; background-color: blue;">
        Box Element
    </div>
    <button id="changeButton">Change Style</button>

    <script>
        const box = document.getElementById('box');

        // Change the background color of the box when the button is clicked
        document.getElementById('changeButton').addEventListener('click', function() {
            box.style.backgroundColor = 'red'; // Changes background color to red
        });
    </script>
</body>
</html>

```

After click :->



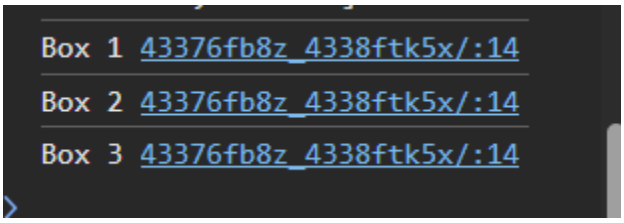
In addition to the common methods like `getElementById()`, there are other ways to access elements in the DOM. These methods are useful depending on what you're trying to achieve.

1. `getElementsByClassName(class)`

- **Explanation**:
  This method returns a **live collection** of elements with the given class name. The collection is "live," meaning it updates automatically when the DOM changes.

```html
<!DOCTYPE html>
<html lang="en">
<body>
    <div class="box">Box 1</div>
    <div class="box">Box 2</div>
    <div class="box">Box 3</div>

    <script>
        const boxes = document.getElementsByClassName('box');
        console.log(boxes);  // Will return a live HTMLCollectio

        // You can loop through the collection and log each box
        for (let i = 0; i < boxes.length; i++) {
            console.log(boxes[i].textContent);
        }
    </script>
</body>
</html>
```

Box 1
Box 2
Box 3



2. `getElementsByTagName(tag)`

- **Explanation**:
  This method returns a **live collection** of all elements with the specified tag name (e.g., `div`, `p`, etc.).

```html
<!DOCTYPE html>
<html lang="en">
<body>
    <p>This is paragraph 1.</p>
    <p>This is paragraph 2.</p>
    <p>This is paragraph 3.</p>

    <script>
        const paragraphs = document.getElementsByTagName('p');
        console.log(paragraphs);  // Returns a live HTMLCollection

        // Loop through paragraphs
        for (let i = 0; i < paragraphs.length; i++) {
            console.log(paragraphs[i].textContent);
        }
    </script>
</body>
</html>
```

This is paragraph 1.

This is paragraph 2.

This is paragraph 3.

```
This  43376ff8e_4338fwbpb/:14
is paragraph 1.
This  43376ff8e_4338fwbpb/:14
is paragraph 2.
This  43376ff8e_4338fwbpb/:14
is paragraph 3.
```

Chapter 6: DOM - Forms

6.1 Submit Event

6.2 Regular Expression

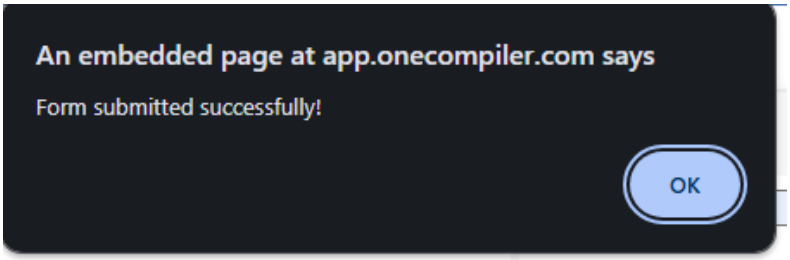
6.3 Basic Form Validation

6.4 Keyboard Event

6.1 Submit Event

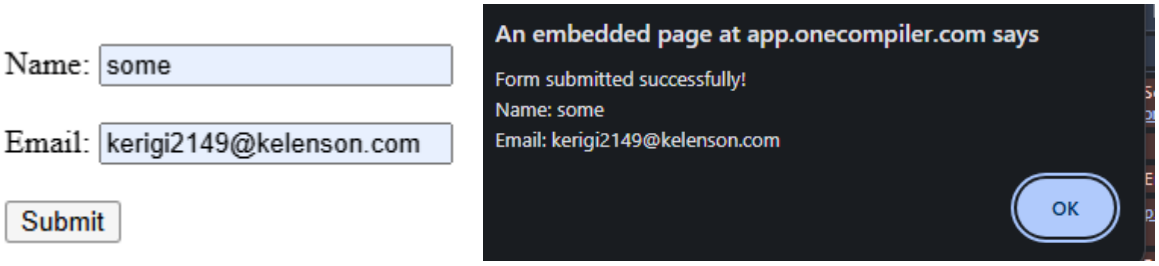
Theory: The `submit` event is triggered when a form is submitted. This event can be used to execute JavaScript when the form is submitted, such as validating form data before sending it to the server.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Submit Event Example</title>
</head>
<body>
    <form id="myForm">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <button type="submit">Submit</button>
    </form>

    <script>
        document.getElementById('myForm').addEventListener('submit', function(event) {
            alert('Form submitted successfully!');
            // Prevent the default form submission
            event.preventDefault();
        });
    </script>
</body>
</html>

```



Example →

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Submit Event Example</title>
</head>
<body>
    <form id="myForm">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <br><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <br><br>
        <button type="submit">Submit</button>
    </form>
    <script>
        document.getElementById('myForm').addEventListener('submit', function(event) {
            // Prevent the default form submission
            event.preventDefault();

            // Get the form data
            const name = document.getElementById('name').value;
            const email = document.getElementById('email').value;

            // Print form data to the console
            console.log('Name:', name);
            console.log('Email:', email);
            // Show an alert with the form data
            alert('Form submitted successfully!\nName: ' + name + '\nEmail: ' + email);
        });
    </script>
</body>
</html>

```




```
Name: some
Email: kerigi2149@kelenson.com
```

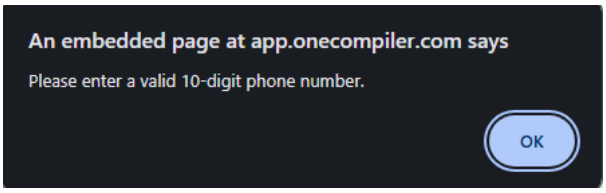
6.2 Regular Expression

Theory: Regular expressions (regex) are patterns used to match character combinations in strings. They are often used for form validation to ensure that user input matches a specific pattern, such as an email address or phone number.

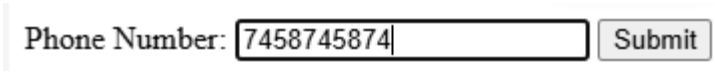
Phone Number →

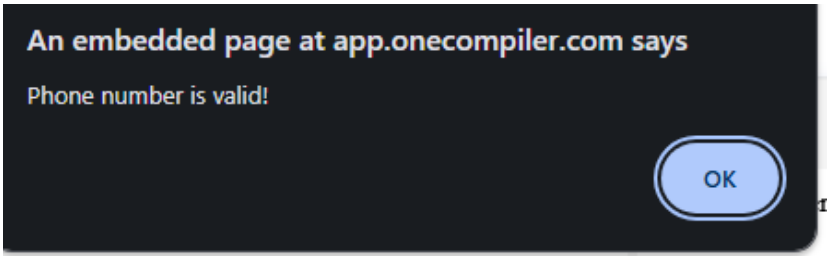
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Phone Number Validation Example</title>
</head>
<body>
    <form id="phoneForm">
        <label for="phone">Phone Number:</label>
        <input type="text" id="phone" name="phone" required>
        <button type="submit">Submit</button>
    </form>

    <script>
        document.getElementById('phoneForm').addEventListener('submit', function(event) {
            const phone = document.getElementById('phone').value;
            // Regular expression for phone number validation
            const phonePattern = /^\d{10}$/;

            if (!phonePattern.test(phone)) {
                alert('Please enter a valid 10-digit phone number.');
                event.preventDefault();
            } else {
                alert('Phone number is valid!');
            }
        });
    </script>
</body>
</html>
```

Incorrect



Correct



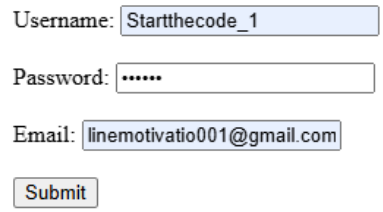


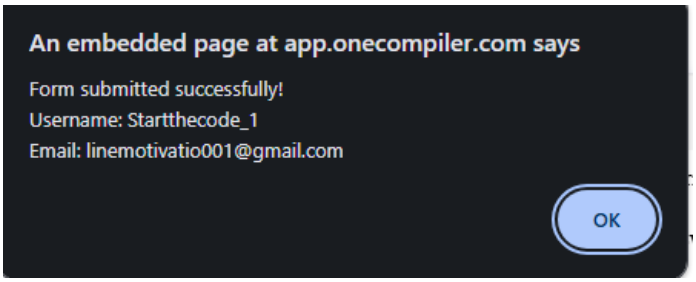
6.3 Basic Form Validation

Theory: Basic form validation is a way to ensure that user input meets certain criteria before the form is submitted. This includes checking if required fields are filled, if input values match specific patterns, and if values fall within a particular range. This helps in preventing incorrect or incomplete data submission.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Basic Form Validation Example</title>
</head>
<body>
    <form id="validationForm">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
        <br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <br><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <br><br>
        <button type="submit">Submit</button>
    </form>
    <script>
        document.getElementById('validationForm').addEventListener('submit', function(event) {
            // Prevent the default form submission
            event.preventDefault();
            // Get the form data
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;
            const email = document.getElementById('email').value;
```

```
    <script>
        document.getElementById('validationForm').addEventListener('submit', function(event) {
            event.preventDefault();
            // Get the form data
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;
            const email = document.getElementById('email').value;

            // Basic validation checks
            if (username === '' || password === '' || email === '') {
                alert('All fields are required!');
                return;
            }
            // Password length check
            if (password.length < 6) {
                alert('Password must be at least 6 characters long.');
                return;
            }
            // Email pattern check
            const emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
            if (!emailPattern.test(email)) {
                alert('Please enter a valid email address.');
                return;
            }
            // If all checks pass
            alert('Form submitted successfully!\nUsername: ' + username + '\nEmail: ' + email);
            console.log('Username:', username);
            console.log('Password:', password);
            console.log('Email:', email);
        });
    </script>
</body>
</html>
```





6.4 Keyboard Event

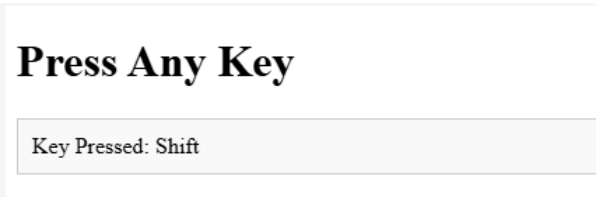
Theory: Keyboard events are events triggered by keyboard actions, such as pressing, releasing, or holding down a key. Common keyboard events include `keydown`, `keyup`, and `keypress`. These events are useful for capturing user input, creating keyboard shortcuts, and enhancing user interactions on a webpage.

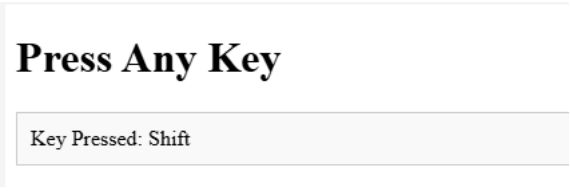
Common Keyboard Events:

1. **keydown**: Triggered when a key is pressed down.
2. **keyup**: Triggered when a key is released.
3. **keypress**: Triggered when a key is pressed down and that key produces a character value (deprecated, `keydown` is preferred).

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Keyboard Event Example</title>
    <style>
        #output {
            margin-top: 20px;
            padding: 10px;
            border: 1px solid #ccc;
            background-color: #f9f9f9;
        }
    </style>
</head>
<body>
    <h1>Press Any Key</h1>
    <div id="output">Key Pressed: </div>

    <script>
        document.addEventListener('keydown', function(event) {
            const output = document.getElementById('output');
            output.textContent = 'Key Pressed: ' + event.key;
        });
    </script>
</body>
</html>
```

Detect All Keyword keys





Chapter 7: Array Methods

7.0 Array Methods

7.0 Array Methods

Array methods in JavaScript are functions that can be called on arrays to perform various operations. They help in manipulating and processing array data efficiently.

7.1 Slice

Theory: The `slice()` method returns a shallow copy of a portion of an array into a new array object. It takes two arguments: start and end (optional). The end index is not included in the new array.

Example:

```
const fruits = ['apple', 'banana', 'cherry', 'date'];
const slicedFruits = fruits.slice(1, 3);

console.log(slicedFruits);

// Output: ['banana', 'cherry']
```

7.2 Splice

Theory: The `splice()` method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. It takes at least two arguments: start and deleteCount.

Example:

Javascript →

```
const fruits = ['apple', 'banana', 'cherry', 'date'];
const removedFruits = fruits.splice(1, 2, 'blueberry', 'citrus');

console.log(fruits);

// Output: ['apple', 'blueberry', 'citrus', 'date']

console.log(removedFruits);

// Output: ['banana', 'cherry']
```

7.3 At

Theory: The `at()` method takes an integer value and returns the item at that index. It supports negative integers to count back from the last item in the array.

Example:

javascript

```
const fruits = ['apple', 'banana', 'cherry', 'date'];
console.log(fruits.at(1));               // Output: 'banana'
console.log(fruits.at(-1));              // Output: 'date'
```

7.4 Map

Theory: The `map()` method creates a new array populated with the results of calling a provided function on every element in the calling array.

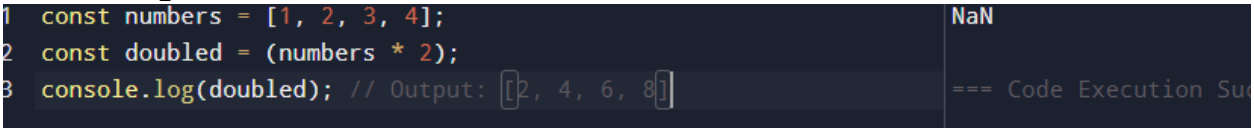
Example:

javascript

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled);                    // Output: [2, 4, 6, 8]
```

for DIRECT USE →

Code o/p in NaN



7.5 Filter

Theory: The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);                // Output: [2, 4]
```

7.6 Reduce

Theory: The `reduce()` method executes a reducer function on each element of the array, resulting in a single output value. It takes a callback function and an initial value.

Example:

Javascript

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) =>
accumulator + currentValue, 0);
console.log(sum);                          // Output: 10
```

*Logic

- **`reduce` ka kaam:**

Yeh array ke elements ko ek single value mein reduce karta hai, yahan hum elements ka sum nikaal rahe hain.

- **Parameters:**

`reduce` method ko ek callback function aur ek initial value pass hoti hai.
Callback Function Parameters:

- **`accumulator`:** Yeh previous iteration ka result ya initial value hota hai.
- **`currentValue`:** Yeh array ka current element hota hai.

- **Initial Value:**

Callback function ke pehle argument (`accumulator`) ki initial value set karne ke liye ek value pass ki jati hai. Yahan wo `0` hai.

Callback Function Workflow:

`reduce` har element par callback function ko run karta hai. Iska flow dekhte hain:

Iteration	accumulator (Initial Result)	currentValue (Current Array Element)	Result (accumulator + currentValue)
1	`0`	`1`	`0 + 1 = 1`
2	`1`	`2`	`1 + 2 = 3`
3	`3`	`3`	`3 + 3 = 6`
4	`6`	`4`	`6 + 4 = 10`

7.7 Find

Theory: The `find()` method returns the value of the first element in the array that satisfies the provided testing function. If no values satisfy the testing function, `undefined` is returned.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const found = numbers.find(num => num > 3);
console.log(found); // Output: 4
```

How `find` Works in Iterations:

`find` ek ek karke array ke elements ko check karega:

Iteration	Current Element (num)	Condition (num > 3)	Result
1	`1`	`1 > 3` (False)	Skip
2	`2`	`2 > 3` (False)	Skip
3	`3`	`3 > 3` (False)	Skip
4	`4`	`4 > 3` (True)	Return `4`

O/p is 4.

7.8 FindIndex

Theory: The `findIndex()` method returns the index of the first element in the array that satisfies the provided testing function. If no elements satisfy the testing function, `-1` is returned.

Example:

javascript

```
const numbers = [1, 2, 3, 4, 5];
const index = numbers.findIndex(num => num > 3);
console.log(index); // Output: 3
```

logic→

How `findIndex` Works in Iterations:

Iteration	Current Element (num)	Condition (num > 3)	Action
1	`1`	`1 > 3` (False)	Continue
2	`2`	`2 > 3` (False)	Continue
3	`3`	`3 > 3` (False)	Continue
4	`4`	`4 > 3` (True)	Return index `3`

- Jaise hi `4` (index `3`) condition ko satisfy karta hai (`4 > 3`), `findIndex` wahan ruk jaata hai aur us element ka **index** (`3`) return karta hai.

Chapter 8: Date

8.1 Date and Time

Chapter 8: Date and Time

Understanding how to work with dates and times is essential for many programming tasks, from displaying the current date to calculating durations or time differences. JavaScript provides a built-in `Date` object that simplifies these tasks.

```
console.log("This is date and time tutorial");
let now = new Date();
console.log(now);

let dt = new Date(1000);
console.log(dt);

// let newDate = new Date("2029-09-30");
// console.log(newDate)

// let newDate = new Date(year, month, date, hours, minutes, seconds, milliseconds);
let newDate = new Date(3020, 4, 6, 9, 3, );
console.log(newDate)
```

DATE →

date_time.html > html > script

```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>
    <script>
        // let specificDate = new Date('2024-12-21');
        // console.log(specificDate);
        let dateComponents = new Date(2024, 12, 21);
        console.log(dateComponents);
        let specificDate = new Date('2024-12-21');
        console.log(specificDate);


    </script>
<body>
    <h1>3456789</h1>
</body>
</html>
```

Month Starting index in 0;

o/p → ?

```
Tue Jan 21 2025 00:00:00 GMT+0530 (India Standard Time)   date_time.html:12
Sat Dec 21 2024 05:30:00 GMT+0530 (India Standard Time)   date_time.html:14
```

```javascript
let dateComponents = new Date(2024, 11, 21); // 11 का मतलब दिसंबर
console.log(dateComponents);
```

Date Methods in JavaScript

Here is a comprehensive list of the various methods available in the `Date` object in JavaScript, along with their usage and why you might use them:

1. `getFullYear()`

- **Usage**: Retrieves the year (4 digits for 4-digit years) from a date.
- **Why**: To get the current year or the year from a specific date.

```
let now = new Date();
console.log(now.getFullYear()); // Outputs: 2023
```

2. `getMonth()`

- **Usage**: Retrieves the month from a date (0-11, where 0 is January and 11 is December).
- **Why**: To get the current month or the month from a specific date.

javascript

```
let now = new Date();
```

```
console.log(now.getMonth()); // Outputs: 3 (April)


explain :-
let now6 = new Date();
console.log(now6.getMonth()); // Outputs: 11 (Novermber)
Because current date is 21-12-2024
So o/p is November .
Because month starting is 0 indexing .
```

3. `getDate()`

- **Usage**: Retrieves the day of the month from a date (1-31).
- **Why**: To get the current day or the day from a specific date.

javascript

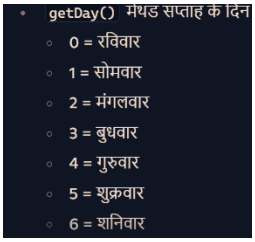
```
let now = new Date();
console.log(now.getDate()); // Outputs: 21
because Current date is 21 .
```

4. `getDay()`

- **Usage**: Retrieves the day of the week from a date (0-6, where 0 is Sunday and 6 is Saturday).
- **Why**: To get the current weekday or the weekday from a specific date.

javascript

```
let now = new Date();
console.log(now.getDay()); // Outputs: 6 (because today is Saturday)
```



5. `setMinutes(minutes, [seconds], [milliseconds])`

- **Usage**: Sets the minutes for a date object, optionally setting seconds and milliseconds.
- **Why**: To change the minutes of a specific date.

javascript

```
let now = new Date();
now.setMinutes(45, 0, 0);
console.log(now); // Date object now represents 45 minutes
```

Chapter 9: LocalStorage

9.1 Local Storage Introduction

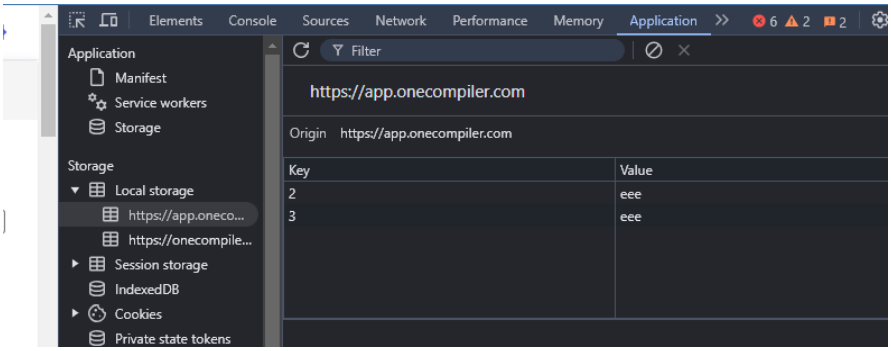
9.1 Local Storage Introduction in JavaScript

Theory: Local Storage is a web storage feature that allows you to store data on the client's browser with no expiration date. This data persists even after the browser is closed and reopened. Local Storage provides a **simple key-value pair storage system**, making it useful for saving user preferences, session data, and other information that needs to be maintained across page loads.

Simple Example :->

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Simple Local Storage Example</title>
</head>
<body>
    <h1>Simple Local Storage Example</h1>
    <input type="text" id="key" placeholder="Enter key">
    <input type="text" id="value" placeholder="Enter value">
    <button onclick="setItem()">Set Item</button>
    <button onclick="getItem()">Get Item</button>
    <button onclick="removeItem()">Remove Item</button>
    <button onclick="clearStorage()">Clear Storage</button>
    <p id="output"></p>
```

```javascript
    <script>
        function setItem() {
            const key = document.getElementById('key').value;
            const value = document.getElementById('value').value;
            localStorage.setItem(key, value);
            document.getElementById('output').textContent = `Set ${key} = ${value}`;
        }

        function getItem() {
            const key = document.getElementById('key').value;
            const value = localStorage.getItem(key);
            document.getElementById('output').textContent = value ? `Got ${key} = ${value}` : `No value found for key "${key}"`;
        }

        function removeItem() {
            const key = document.getElementById('key').value;
            localStorage.removeItem(key);
            document.getElementById('output').textContent = `Removed item with key "${key}"`;
        }

        function clearStorage() {
            localStorage.clear();
            document.getElementById('output').textContent = 'Cleared all items from Local Storage';
        }
    </script>
</body>
</html>
```



Get

Clear Storage :

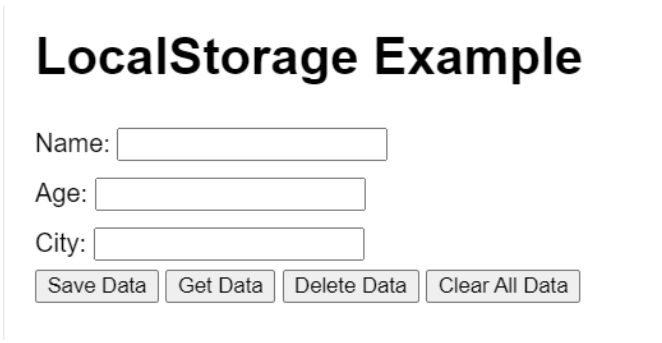
Another :

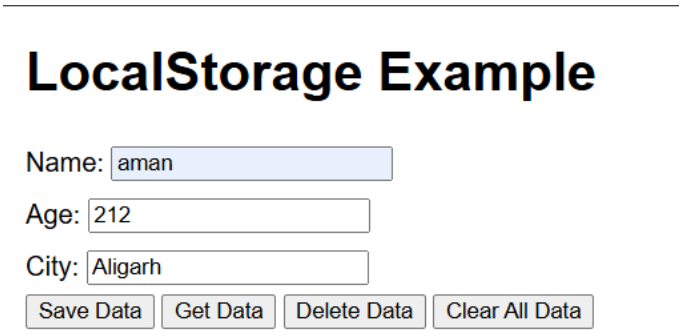
code_1.html

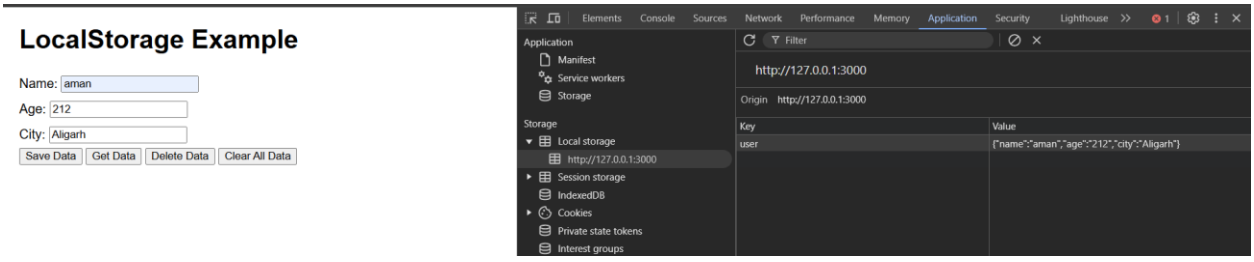
code_1.html > html > body > script > getData

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>LocalStorage Example</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
        }
        input {
            margin: 5px 0;
        }
    </style>
</head>
<body>
    <h1>LocalStorage Example</h1>

    <label for="nameInput">Name:</label>
    <input type="text" id="nameInput"><br>

    <label for="ageInput">Age:</label>
    <input type="number" id="ageInput"><br>

    <label for="cityInput">City:</label>
    <input type="text" id="cityInput"><br>

    <button onclick="saveData()">Save Data</button>
    <button onclick="getData()">Get Data</button>
    <button onclick="deleteData()">Delete Data</button>
    <button onclick="clearData()">Clear All Data</button>

    <script>
        // Function to save data to LocalStorage
        function saveData() {
            const name = document.getElementById('nameInput').value;
            const age = document.getElementById('ageInput').value;
            const city = document.getElementById('cityInput').value;

            // Create a user object
            const user = {
                name: name,
                age: age,
                city: city
            };
            // Convert the user object to a JSON string and store it in LocalStorage
            localStorage.setItem('user', JSON.stringify(user));
            alert('Data saved!');
        }
        // Function to get data from LocalStorage
        function getData() {
            // Retrieve the user data from LocalStorage
            const user = JSON.parse(localStorage.getItem('user'));

            if (user) {
                alert(`Name: ${user.name}\nAge: ${user.age}\nCity: ${user.city}`);
            } else {
                alert('No data found!');
            }
        }

        // Function to delete specific data from LocalStorage
        function deleteData() {
            localStorage.removeItem('user');
            alert('Data deleted!');
```

```
        }
        // Function to get data from LocalStorage
        function getData() {
            // Retrieve the user data from LocalStorage
            const user = JSON.parse(localStorage.getItem('user'));

            if (user) {
                alert(`Name: ${user.name}\nAge: ${user.age}\nCity: ${user.city}`);
            } else {
                alert('No data found!');
            }
        }

        // Function to delete specific data from LocalStorage
        function deleteData() {
            localStorage.removeItem('user');
            alert('Data deleted!');
        }

        // Function to clear all data from LocalStorage
        function clearData() {
            localStorage.clear();
            alert('All data cleared!');
        }
    </script>
</body>
</html>
```







Chapter 10: OOP

10.1 Constructor and `new` Operator

10.2 Prototypes
10.3 Prototypical Inheritance
10.4 ES6 Classes
10.5 Setters and Getters
10.6 Static Methods
10.7 Class Inheritance
10.8 Inheritance by Prototypes
10.9 Chaining of Methods

Chapter 10: Object-Oriented Programming (OOP)

Simple Example of Objects and Classes

Theory:

1. **Object:** An object is an instance of a class. It encapsulates data (attributes) and methods (functions) that operate on the data.
2. **Class:** A class is a blueprint for creating objects. It defines properties and behaviors that the objects created from the class will have.

```
// Define a class
class Person {
    // Constructor function to initialize object properties
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    // Method to greet
    greet() {
        console.log(`Hello, my name is ${this.name} and I am ${this.age} ye
    }
}

// Create an instance (object) of the Person class
const john = new Person('John', 30);
const jane = new Person('Jane', 25);

// Access object properties and methods
john.greet(); // Output: Hello, my name is John and I am 30 years old.
jane.greet(); // Output: Hello, my name is Jane and I am 25 years old.
```

Without object use for this code , using static keyword →

```
// Define a class
class Person {
    // Constructor function to initialize object properties
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    // Static method to greet
    static greet(name, age) {
        console.log(`Hello, my name is ${name} and I am ${age} years old.`)
    }
}

// Call the static method directly on the class
Person.greet('John', 30); // Output: Hello, my name is John and I am 30 yea
Person.greet('Jane', 25); // Output: Hello, my name is Jane and I am 25 yea
```

- **Static Method:** The `greet` method is defined as a static method using the `static` keyword.

- **Calling the Method:** Static methods are called on the class itself rather than on instances of the class. Therefore, you can call `Person.greet('John', 30)` directly without creating an instance.

Pillars

Inheritance

Inheritance allows a class to inherit properties and methods from another class, promoting code reusability.

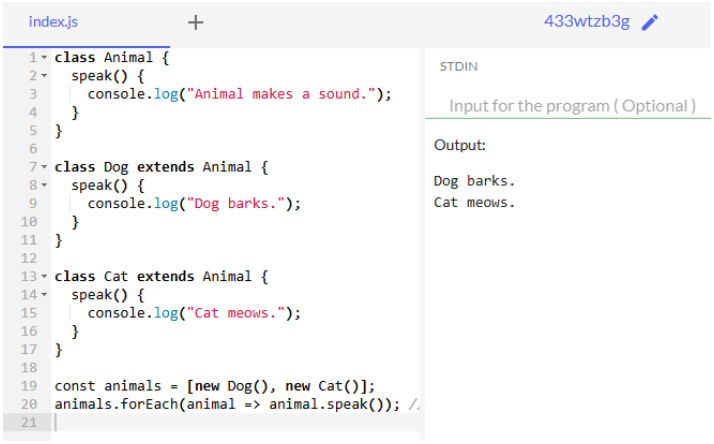
```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog("Rex");
dog.speak(); // Output: Rex barks.
```

Here's a breakdown of what each part does:

- **`class Person { ... }`**: This defines a new class named `Person`.
- **`constructor(name) { ... }`**: The `constructor` is a special function that gets called whenever a new instance of the `Person` class is created. It takes a parameter `name`.
- **`this.name = name;`**: Inside the constructor, `this` refers to the new instance being created. This line sets the `name` property of the instance to the value passed as the `name` parameter.

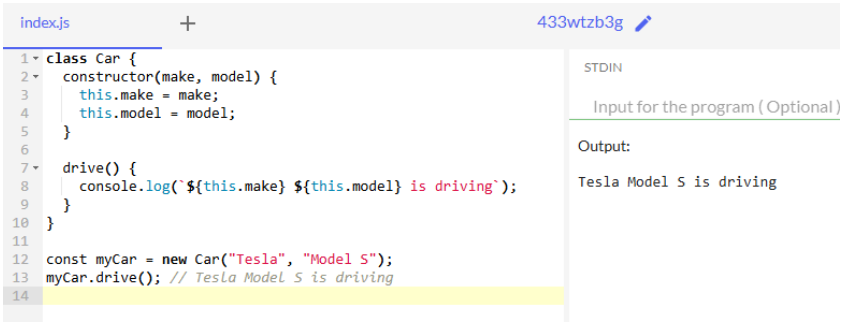
Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms (data types).

Theory: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the use of methods with the same name but potentially different implementations in different classes. In JavaScript, polymorphism is typically implemented using method overriding, where a subclass redefines a method from its parent class.



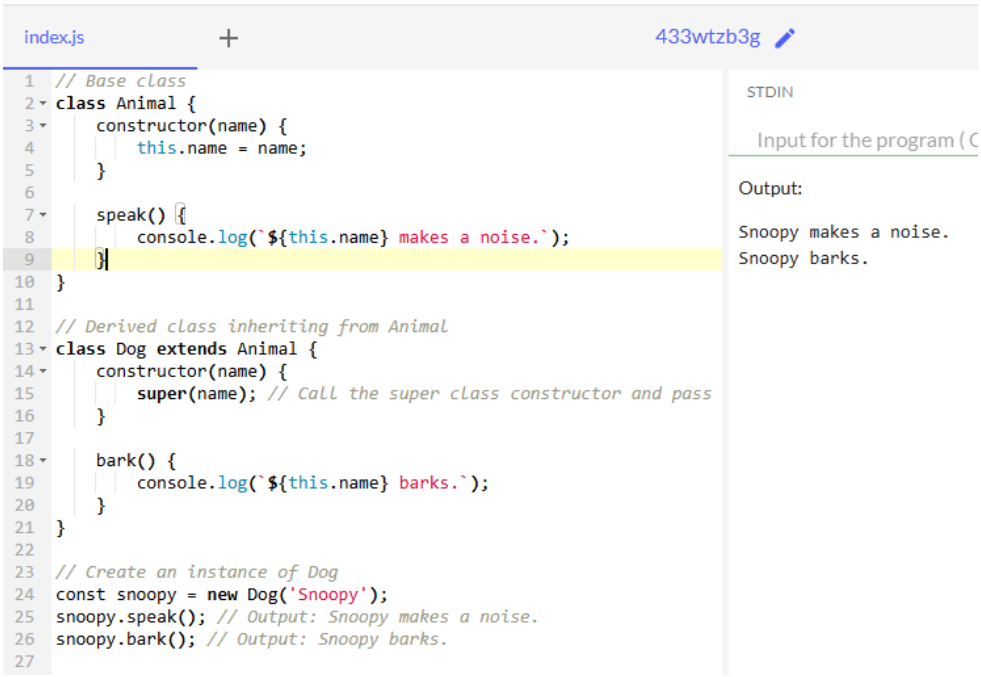
Abstraction:

Theory: Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object. It allows the developer to focus on the higher-level functionality without worrying about the internal workings. In JavaScript, abstraction is often achieved through the use of classes and methods that provide a simplified interface.



What is Inheritance?

Inheritance in object-oriented programming (OOP) allows a class (child class) to **inherit** properties and methods from another class (parent class). This helps avoid repeating code and promotes code reuse.



ENCAPSULATION :

10.1 Constructor and `new` Operator

Constructor and `new` Operator in JavaScript

In JavaScript, constructors and the `new` operator are used to create and initialize objects. Let's break down these concepts clearly:

1. Constructor in JavaScript

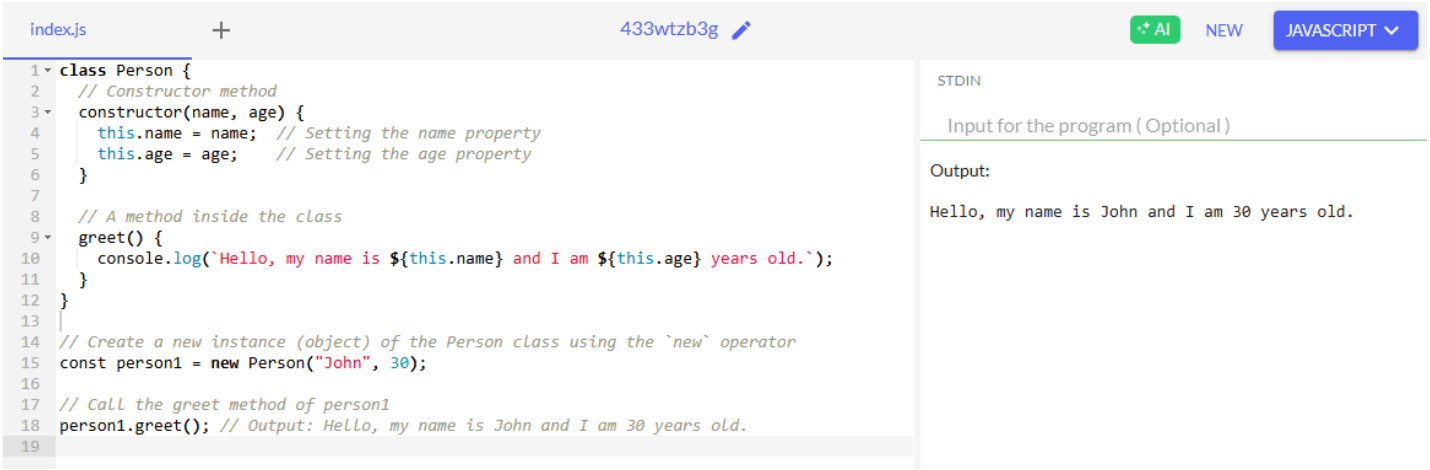
A **constructor** is a special function used to initialize objects created from a class or function. It is automatically called when a new object is created using the `new` keyword.

- **For Classes**: A constructor is a special method in a class that gets called when an object of that class is created.
- **For Functions (Pre-ES6)**: Functions can also act as constructors when used with the `new` keyword.

2. The `new` Operator

The `new` operator is used to create an instance of a class or a function (used as a constructor). When `new` is used:

1. A new empty object is created.
2. The constructor function is called with `this` referring to the newly created object.
3. The newly created object is returned from the constructor (unless the constructor explicitly returns another object).



10.2 Prototypes

Concept: Prototypes are a fundamental feature in JavaScript that allows objects to inherit properties and methods from other objects. Every JavaScript object has a prototype, which is another object that the original object inherits properties from.

Why Use Prototypes? :

- **Inheritance**: Allows objects to share properties and methods, promoting code reuse.
- **Efficient Memory Usage**: Methods can be defined once on the prototype and shared by all instances, rather than defining them on each instance separately.

- **Dynamic Updates**: Adding or modifying methods on the prototype automatically updates all instances that inherit from it.

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

// Adding a method to the prototype
Person.prototype.greet = function() {
    console.log(`Hello, my name is ${this.name}`);
};

const person1 = new Person('John', 30);
person1.greet(); // Outputs: Hello, my name is John
```

Without Prototype

When we don't use prototypes, each object instance has its own copy of methods, which can lead to redundant memory usage.

```
function Car(model, year) {
    this.model = model;
    this.year = year;
    this.drive = function() {
        console.log(`${this.model} is driving`);
    };
}

const car1 = new Car('Toyota', 2020);
const car2 = new Car('Honda', 2019);
const car3 = new Car('Ford', 2018);

car1.drive(); // Outputs: Toyota is driving
car2.drive(); // Outputs: Honda is driving
car3.drive(); // Outputs: Ford is driving
```

With Prototype →

```
function Car(model, year) {
    this.model = model;
    this.year = year;
}

// Adding the drive method to the prototype
Car.prototype.drive = function() {
    console.log(`${this.model} is driving`);
};

const car1 = new Car('Toyota', 2020);
const car2 = new Car('Honda', 2019);
const car3 = new Car('Ford', 2018);

car1.drive(); // Outputs: Toyota is driving
car2.drive(); // Outputs: Honda is driving
car3.drive(); // Outputs: Ford is driving
```

Another Example : →

```
function Car(model, year) {
    this.model = model;
    this.year = year;
}
Car.prototype.drive = function() {
    console.log(`${this.model} is driving`);
};

// Stop method
Car.prototype.stop = function() {
    console.log(`${this.model} has stopped`);
};
// Honk method
Car.prototype.honk = function() {
    console.log(`${this.model} is honking: Beep! Beep!`);
};
const car1 = new Car('Toyota', 2020);
const car2 = new Car('Honda', 2019);
const car3 = new Car('Ford', 2018);
```

o/p →

```

car1.drive(); // Outputs: Toyota is driving
car1.stop();  // Outputs: Toyota has stopped
car1.honk();  // Outputs: Toyota is honking: Beep! Beep!

car2.drive(); // Outputs: Honda is driving
car2.stop();  // Outputs: Honda has stopped
car2.honk();  // Outputs: Honda is honking: Beep! Beep!

car3.drive(); // Outputs: Ford is driving
car3.stop();  // Outputs: Ford has stopped
car3.honk();  // Outputs: Ford is honking: Beep! Beep!

```

```
Toyota is driving
Toyota has stopped
Toyota is honking: Beep! Beep!
Honda is driving
Honda has stopped
Honda is honking: Beep! Beep!
Ford is driving
Ford has stopped
Ford is honking: Beep! Beep!
```

10.3 Prototypical Inheritance

Theory: Prototypical Inheritance is a core concept in JavaScript where objects can inherit properties and methods from other objects. This is achieved through the prototype chain, which allows objects to reference other objects' properties and methods.

Prototypical Inheritance enables:

1. **Inheritance**: Objects inherit properties and methods from their prototypes.
2. **Prototype Chain**: If a property or method is not found on an object, JavaScript looks up the prototype chain to find it.
3. **Dynamic Nature**: Prototypes can be modified at runtime, and changes are reflected in all inheriting objects

4. Prototypical Inheritance (Theory and Code)

5. Prototypical inheritance is a fundamental concept in JavaScript. It allows objects to inherit properties and methods from other objects. This differs from classical inheritance (used in languages like Java or C++) because, in JavaScript, every object has a prototype (another object), and it can inherit properties and methods from that prototype.

Without use →

Example without Using `Object.create()` (Prototype Property):

In this example, we won't use `Object.create()`, but we'll manually set up the prototype inheritance.

```
function Animal(species) {
  this.species = species;
}
Animal.prototype.speak = function() {
  console.log(this.species + ' makes a sound');
};
// Child object constructor
function Dog(name) {
  Animal.call(this, 'Dog'); // Call the parent constructor
  this.name = name;
}
// Set up the inheritance manually
Dog.prototype = Object.create(Animal.prototype); // Inherit methods
    from Animal
Dog.prototype.constructor = Dog; // Correct the constructor reference
// Add additional methods specific to Dog
Dog.prototype.bark = function() {
  console.log(this.name + ' barks');
};
// Create an instance of Dog
const dog = new Dog('Buddy');
dog.speak();  // Output: "Dog makes a sound"
dog.bark();   // Output: "Buddy barks"
```

Use →

```
// Parent object
const animal = {
  species: 'Unknown',
  speak() {
    console.log(this.species + ' makes a sound');
  }
};

// Creating a child object
const dog = Object.create(animal); // dog inherits from animal
dog.species = 'Dog'; // Overriding the species property for dog
dog.speak(); // Output: "Dog makes a sound"

// Creating another child object
const cat = Object.create(animal);
cat.species = 'Cat';
cat.speak(); // Output: "Cat makes a sound"
```

```
Dog makes a sound
Cat makes a sound

=== Code Execution Successful ===
```

The code is very easy and simple

```
function Animal(eats) {
    this.eats = eats;
    this.walk = function() {
        console.log("Animal walks");
    };
}

function Rabbit(jumps) {
    this.jumps = jumps;
    this.eats = true; // Inherited property added manually
    this.walk = function() {
        console.log("Animal walks"); // Inherited method added manually
    };
}

const rabbit1 = new Rabbit(true);

console.log(rabbit1.eats); // true, manually added
console.log(rabbit1.jumps); // true, own property
rabbit1.walk(); // Animal walks, manually added method
```

10.4 ES6 Classes

Theory: ES6 (ECMAScript 2015) introduced a class syntax that provides a clearer and more concise way to create objects and handle inheritance. Although classes in JavaScript are syntactic sugar over the prototypical inheritance model, they make the code more readable and maintainable.

Features of ES6 Classes:

1. **Class Declaration**: Use the `class` keyword to define a class.
2. **Constructor Method**: A special method for creating and initializing objects created within a class.
3. **Inheritance**: Use the `extends` keyword to create a subclass.
4. **Super**: Use the `super` keyword to call the constructor and methods of the parent class.

main.js

Share Run Output

```
// Base class
class Animal {
    constructor(name) {
        this.name = name;
    }
    walk() {
        console.log(`${this.name} walks`);
    }
}
// Subclass
class Rabbit extends Animal {
    constructor(name, color) {
        super(name); // Call the constructor of the parent class
        this.color = color;
    }
    jump() {
        console.log(`${this.name} jumps`);
    }
}
const rabbit1 = new Rabbit("Bunny", "white");
rabbit1.walk(); // Bunny walks
rabbit1.jump(); // Bunny jumps
// Check instance
console.log(rabbit1 instanceof Rabbit); // true
console.log(rabbit1 instanceof Animal); // true
```

```
Bunny walks
Bunny jumps
true
true

=== Code Exec
```

Using Super Keyword (simple Code and simple logic)

main.js

Share Run Output

```
// Base class
class Animal {
    constructor(name) {
        this.name = name;
    }
    makeSound() {
        console.log(`${this.name} makes a sound`);
    }
}
// Subclass
class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Call the constructor of the parent class
        this.breed = breed;
    }
    bark() {
        console.log(`${this.name} barks: Woof! Woof!`);
    }
}
const dog1 = new Dog('Buddy', 'Golden Retriever');
dog1.makeSound(); // Outputs: Buddy makes a sound
dog1.bark();      // Outputs: Buddy barks: Woof! Woof!
```

```
Buddy makes a sound
Buddy barks: Woof! Woof!

=== Code Execution Success
```

10.5 Setters and Getters

Theory: Setters and getters are special methods in JavaScript that allow you to define how properties of an object are accessed and updated. They are useful for adding extra logic when reading or setting property values.

```
class User {
    constructor(name) {
        this._name = name;
    }
    get name() {
        return this._name;
    }

    set name(value) {
        if (value.length > 0) {
            this._name = value;
        } else {
            console.log('Name cannot be empty');
        }
    }
}
const user = new User('Alice');
console.log(user.name); // Outputs: Alice
user.name = 'Bob';
console.log(user.name); // Outputs: Bob
user.name = ''; // Outputs: Name cannot be empty
```

```
Alice
Bob
Name cannot be empty

=== Code Execution Successfu
```

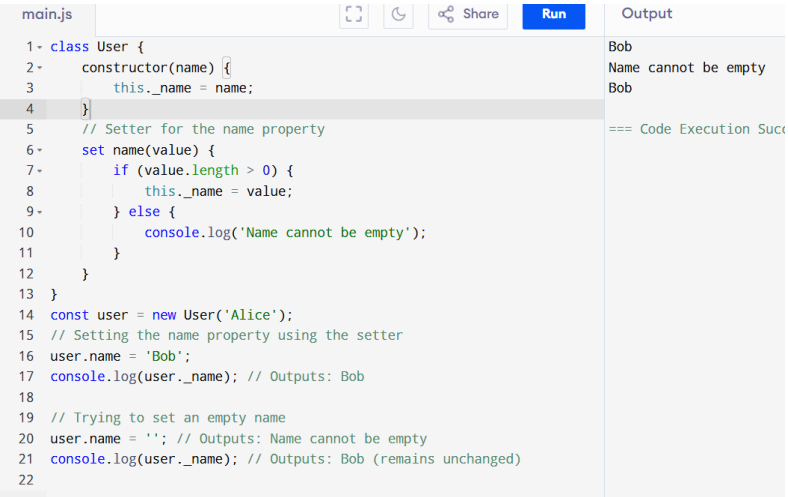
Difference →

Aspect	Getter	Setter
Purpose	Retrieve the value of a property	Set the value of a property
Syntax	Defined using the `get` keyword	Defined using the `set` keyword
Usage	Accessed like a regular property (e.g., `object.property`)	Assigned like a regular property (e.g., `object.property = value`)
Logic	Can include logic for processing or transforming the value	Can include logic for validation or transformation before setting the value
Read-Only	Can make properties read-only by not defining a corresponding setter	Cannot make properties read-only
Controlled Access	Allows controlled access to	Allows controlled modification of

Only get use →

```
class User {
    constructor(name) {
        this._name = name;
    }

    // Getter for the name property
    get name() {
        return this._name;
    }
}

const user = new User('Alice');
console.log(user.name); // Outputs: Alice
```

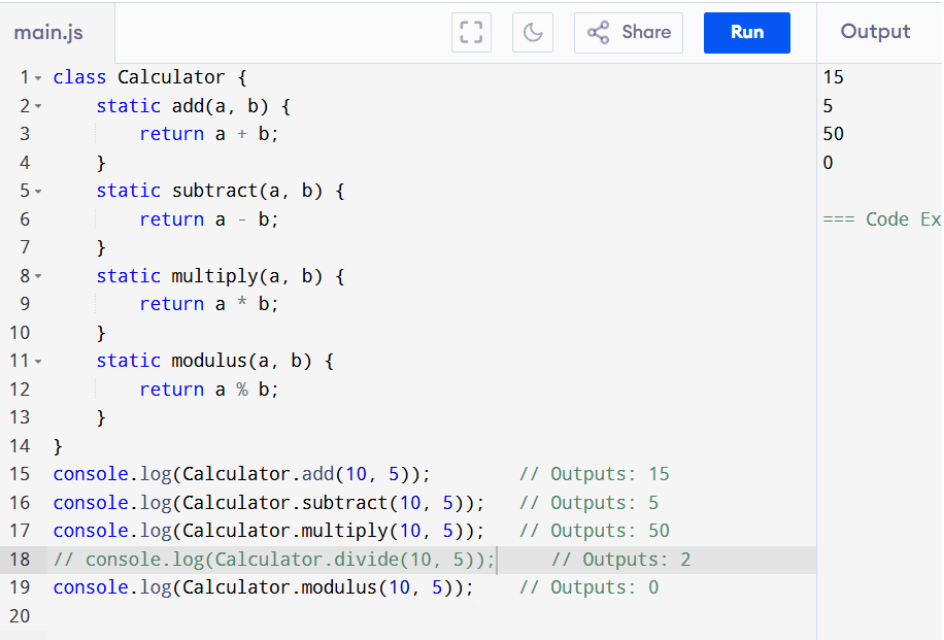
Only value Set :→



10.6 Static Methods
10.7 Class Inheritance

10.6 Static Methods

Theory: Static methods are functions that are defined on the class itself rather than on instances of the class. They are called directly on the class and do not require an instance to be invoked. Static methods are often used for utility functions, such as creating or configuring objects.



10.7 Class Inheritance

Theory: Class inheritance allows one class to inherit properties and methods from another class. This is achieved using the `extends` keyword. The subclass can also have its own additional properties and methods or override the parent class's methods.

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    makeSound() {
        console.log(`${this.name} makes a sound`);
    }
}
class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Call the parent class constructor
        this.breed = breed;
    }
    bark() {
        console.log(`${this.name} barks: Woof! Woof!`);
    }
}
const dog = new Dog('Buddy', 'Golden Retriever');
dog.makeSound(); // Outputs: Buddy makes a sound
dog.bark();      // Outputs: Buddy barks: Woof! Woof!

```

```
Buddy makes a sound
Buddy barks: Woof! Woof!

=== Code Execution Successful =
```

10.9 Chaining of Methods

Theory: Method chaining is a technique that allows you to call multiple methods on the same object consecutively in a single statement. Each method returns the object itself, allowing further methods to be called on it.

main.js Share Run Output

```
class Calculator {
    constructor(value = 0) {
        this.value = value;}
    add(number) {
        this.value += number;
        return this; // Return the object itself
    }
    subtract(number) {
        this.value -= number;
        return this; // Return the object itself
    }
    multiply(number) {
        this.value *= number;
        return this; }
    divide(number) {
        if (number !== 0) {
            this.value /= number;}
        return this; // Return this for chaining
    }
    result() {
        console.log(this.value);
        return this.value;}}
const calc = new Calculator();
calc.add(10).subtract(2).multiply(3).divide(2).result(); // Outputs:
    12

```

```
12

=== Code Exe
```

```
const calc = new Calculator();
calc.add(10).subtract(2).multiply(3).divide(2).result(); // Outputs: 12
```

- `const calc = new Calculator();` : यह `Calculator` का एक नई इंस्टेंस बनाता है जिसका प्रारंभिक `value` 0 होता है।
- `calc.add(10).subtract(2).multiply(3).divide(2).result();` : यह विधियाँ चेनिंग तरीके से कॉल की जाती हैं।
  - `add(10)` : प्रारंभिक `value` 0 में 10 जोड़ता है → वर्तमान `value` = 10।
  - `subtract(2)` : 10 से 2 घटाता है → वर्तमान `value` = 8।
  - `multiply(3)` : 8 को 3 से गुणा करता है → वर्तमान `value` = 24।
  - `divide(2)` : 24 को 2 से भाग करता है → वर्तमान `value` = 12।
  - `result()` : वर्तमान `value` (12) को कंसोल में प्रदर्शित करता है और इसे वापस लौटाता है।

PROJECT →

TODO List:

index.html styles.css script.js

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-Do List</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div class="todo-container">
        <h1>My To-Do List</h1>
        <input type="text" id="new-task" placeholder="Add a new task...">
        <button id="add-task">Add Task</button>
        <ul id="task-list"></ul>
    </div>
    <script src="script.js"></script>
</body>
</html>
```

index.html styles.css script.js

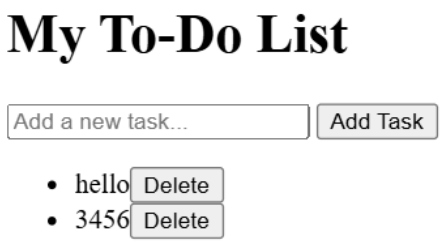
```css
body {
    font-family: Arial, sans-serif;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background-color: #f4f4f4;
    margin: 0;
}

.todo-container {
    background: white;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    text-align: center;
    width: 300px;
}

h1 {
    margin-bottom: 20px;
}

input[type="text"] {
    width: calc(100% - 40px);
    padding: 10px;
    margin-bottom: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
}

button {
    padding: 10px;
    border: none;
    background-color: #007BFF;
    color: white;
    border-radius: 5px;
    cursor: pointer;
}

button:hover {
    background-color: #0056b3;
}

ul {
    list-style: none;
    padding: 0;
}
```

```css
li {
    padding: 10px;
    border-bottom: 1px solid #ddd;
    display: flex;
    justify-content: space-between;
}
```

index.html styles.css script.js

```
document.getElementById('add-task').addEventListener('click', function() {
    const taskInput = document.getElementById('new-task');
    const taskValue = taskInput.value;

    if (taskValue) {
        const li = document.createElement('li');
        li.textContent = taskValue;
        const deleteButton = document.createElement('button');
        deleteButton.textContent = 'Delete';
        deleteButton.addEventListener('click', function() {
            li.remove();
        });

        li.appendChild(deleteButton);
        document.getElementById('task-list').appendChild(li);
        taskInput.value = '';
    }
});
```

o/p ->



➔

✔ What is a Constructor?

A **constructor** is a special function used to **create and initialize an object** in object-oriented programming.

In **JavaScript**, the constructor is:

- Defined using the `constructor()` method inside a class.
-

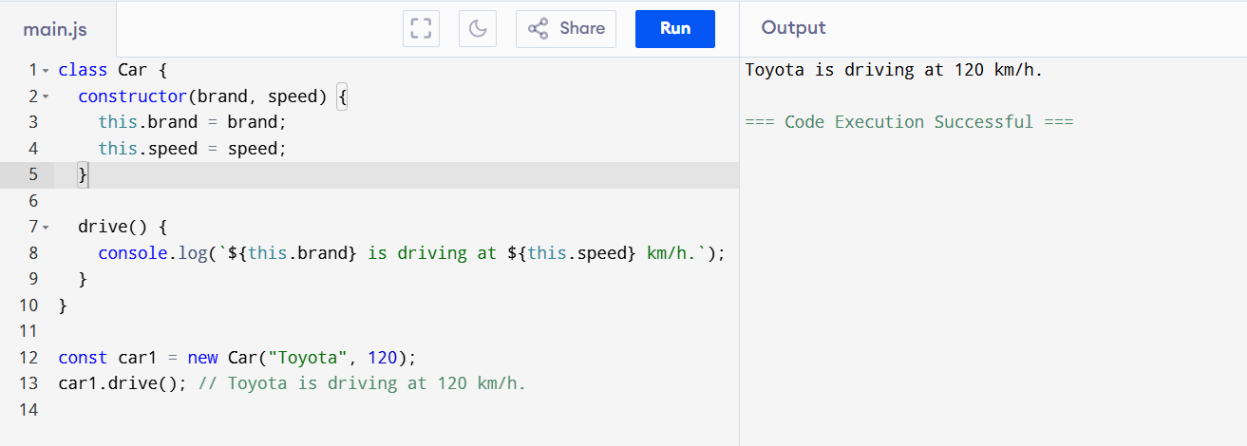
🤔 Think of it like this:

A constructor is like the **blueprint's builder** — it fills in the object's details when you create it.

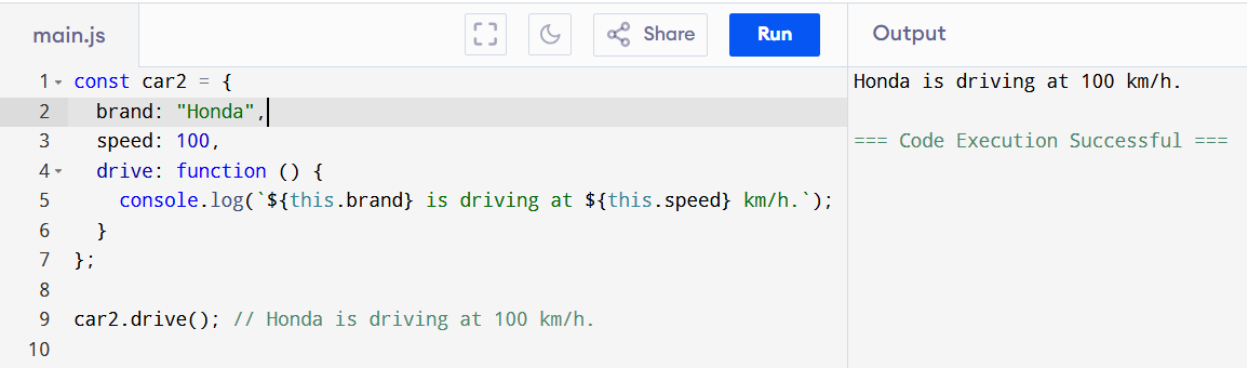
◆ Constructor Definition (in simple Hindi + English):

Constructor ek special function hota hai jo tab automatically chalta hai **jab object banate ho** kisi class ka.
Iska kaam hota hai **object ke andar values set karna** (initialization).

With Constructor ➔



w/o ➔



➔

✓ What is Inheritance?

Inheritance means one class (child) can use properties and methods of another class (parent).

Think of it like: **"Bachcha apne maa-baap ke features le sakta hai."**

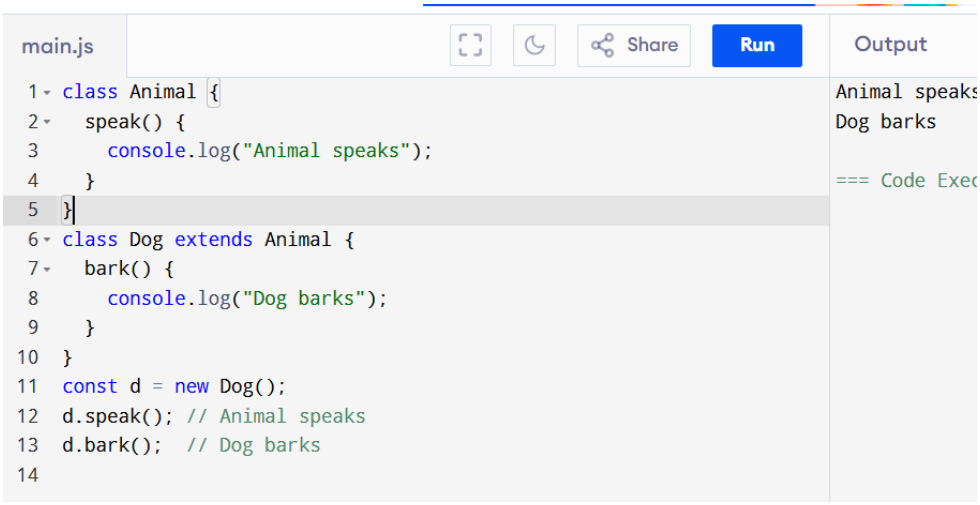
? Types of Inheritance in JavaScript

JavaScript supports inheritance mainly through **prototypes** and **classes (ES6)**.

Here are the main types:

1. Single Inheritance

One child class inherits from one parent.



2. Multilevel Inheritance

Child → Parent → Grandparent

main.js Share Run Output

```
class LivingBeing {
  live() {
    console.log("I am alive");
  }
}

class Animal extends LivingBeing {
  speak() {
    console.log("Animal speaks");
  }
}

class Dog extends Animal {
  bark() {
    console.log("Dog barks");
  }
}

const d = new Dog();
d.live();   // I am alive
d.speak();  // Animal speaks
d.bark();   // Dog barks

```

```
I am alive
Animal speaks
Dog barks

=== Code Executi
```

3. Hierarchical Inheritance

One parent → multiple children

main.js

Share Run

Output

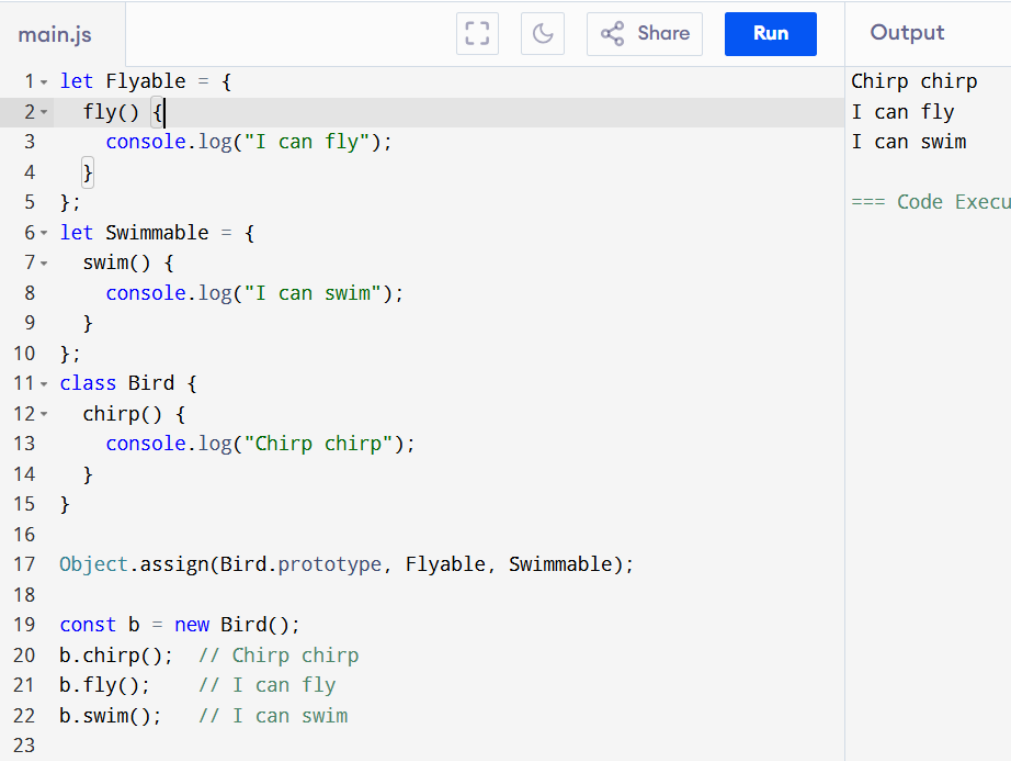
```
class Vehicle {
  start() {
    console.log("Engine starts");
  }
}
class Car extends Vehicle {
  drive() {
    console.log("Car is driving");
  }
}
class Bike extends Vehicle {
  ride() {
    console.log("Bike is riding");
  }
}

const c = new Car();
c.start(); // Engine starts
c.drive(); // Car is driving

const b = new Bike();
b.start(); // Engine starts
b.ride();  // Bike is riding

```

```
Engine starts
Car is driving
Engine starts
Bike is riding

=== Code Execution
```

⚠ 4. Multiple Inheritance (Not directly supported)

JavaScript **does not support** true multiple inheritance (i.e., one class inheriting from multiple parents) but you can simulate it using **mixins**.

✓ Mixin Example (Workaround for Multiple Inheritance):



Inheritance Type	Supported?	Example Class Usage
Single	✓ Yes	`class B extends A`
Multilevel	✓ Yes	`A → B → C`
Hierarchical	✓ Yes	One parent → many children
Multiple	✗ (via mixins)	`Object.assign()` workaround

✓ What is Encapsulation?

Encapsulation means **hiding internal details** of an object and only exposing **what's necessary**.

Think of it like a **TV remote** — you don't need to know what's inside, just how to use it.

➔
Encapsulation is the process of **binding data and methods** together and **restricting direct access** to some of the object's internal details.

main.js

Share Run Output

```
class TVRemote {
  #volume = 10; // 🔒 private variable
  // ✅ Getter (to access safely)
  getVolume() {
    return this.#volume;
  }
  // ✅ Setter (to modify with control)
  setVolume(v) {
    if (v >= 0 && v <= 100) {
      this.#volume = v;
    } else {
      console.log("❌ Volume must be between 0 and 100");
    }
  }
}
const remote = new TVRemote();

console.log(remote.getVolume());  // ✅ 10
remote.setVolume(50);             // ✅ set to 50
console.log(remote.getVolume());  // ✅ 50

remote.setVolume(200);            // ❌ Invalid, will not change
console.log(remote.getVolume());  // ✅ still 50

```

```
10
50
❌ Volume must be between 0 and 100
50

=== Code Execution Successful ===
```

Symbol	Type	Compares	Converts Type?	Language
==	Loose	Value	✓ Yes	JavaScript
===	Strict	Value + Type	✗ No	JavaScript
`.equals()`	Method	Value	✗ No	**Java only**

```
5 === "5"   // false ❌ (number ≠ string)
true === 1  // false ❌
5 === 5     // true ✅
```

➔constructor

main.js Share Run Output

```
class Car {
  constructor(name) {
    this.name = name; // ✅ initialization
  }

  showName() {
    console.log(`🚗 Car name is ${this.name}`);
  }
}

const c = new Car("BMW"); // 👆 constructor call
c.showName(); // 🚗 Car name is BMW

```

```
🚗 Car name is BMW

=== Code Execution Successful
```

➔

Abstraction

main.js Share Run Output

```
class Car {
  #engineStarted = false; // 🔒 private detail

  start() {
    this.#engineStarted = true;
    console.log("🚀 Car started");
  }

  stop() {
    this.#engineStarted = false;
    console.log("🛑 Car stopped");
  }
}

const c = new Car();
c.start(); // 🚀 Car started
c.stop();  // 🛑 Car stopped

```

```
🚀 Car started
🛑 Car stopped

=== Code Execution Su
```

Feature	Constructor	Abstraction
Purpose	Object banate hi data set karna	Use karna asaan banana (andar ka logic hide)
Kab use hota hai?	`new ClassName(...)` ke time	Object banne ke baad
Focus	Initialization (setup)	Hiding complexity
Visible to User	Yes (`constructor(...)`)	No (andar ka logic nahi dikhata)

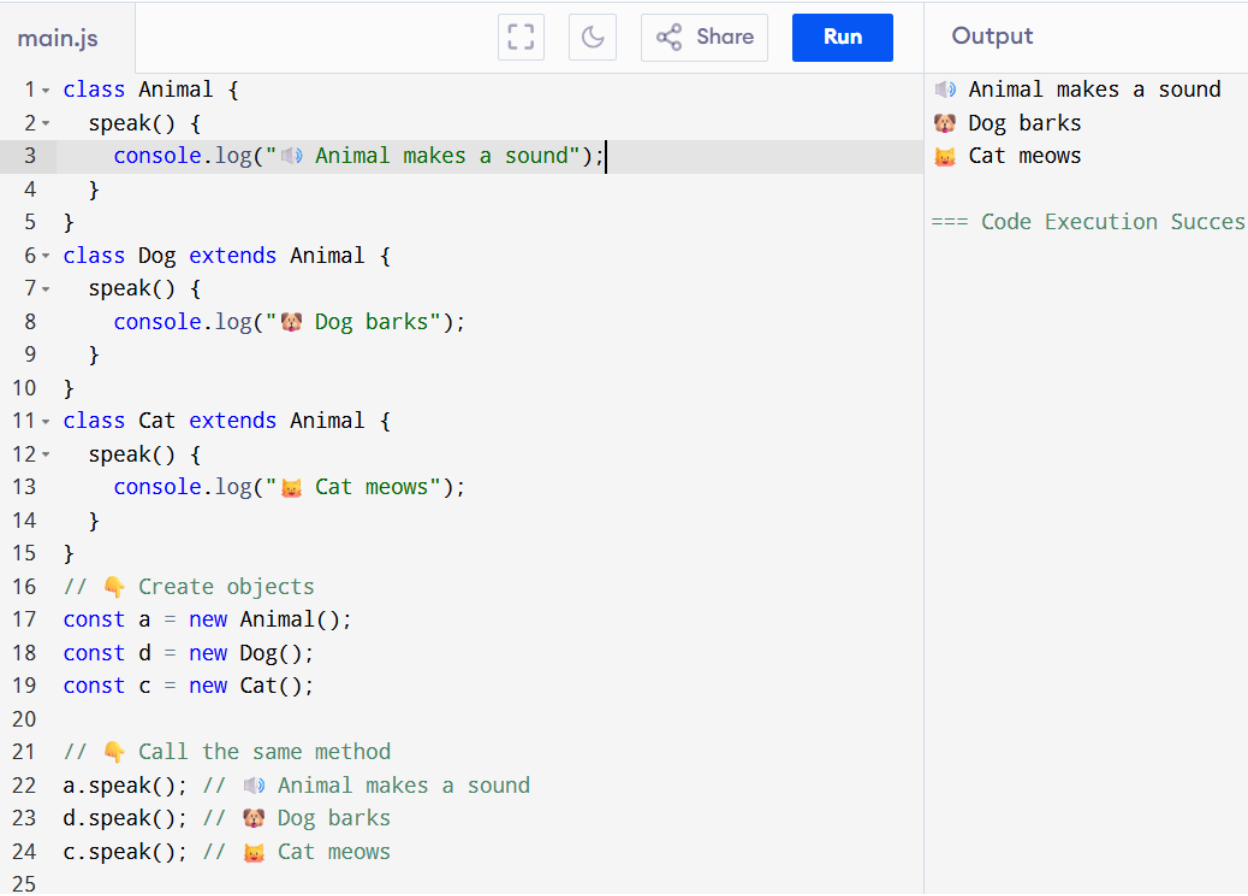
✅ What is Polymorphism?

Poly = many
Morph = form
So, **Polymorphism** means:
"Ek hi cheez ka alag-alag form ya behavior."

◆ Real-life Example:

- **"draw()" function**:
  - Circle ke liye **draw a circle**
  - Square ke liye **draw a square**
  - Rectangle ke liye **draw a rectangle**

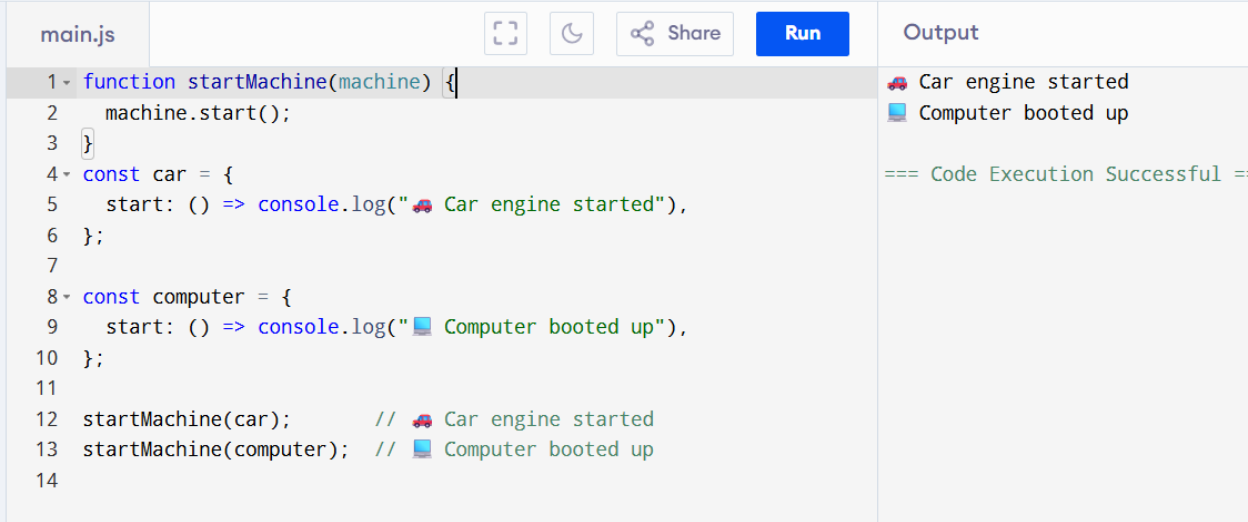
Type	Description
1. Method Overriding	Same method name, different classes
2. Duck Typing	JS ka unique style — function call based on behavior



Duck Typing Simple Definition:

"Agar koi object kisi method ko support karta hai, to hum use wo method call kar lete hain — bina ye jaane ki wo kis class ka hai."

Iska naam hai:
"Agar wo duck ki tarah chalta hai, aur duck ki tarah awaaz karta hai, to wo duck hi hoga."



Concept	Status	Tera Progress
Constructor	✅	`constructor()` se object init kiya
Encapsulation	✅	private variable + getter/setter (like TV Remote)
Abstraction	✅	sirf zaroori method dikhaye, logic chhupaya
Inheritance	✅	`extends` keyword, base → derived
Polymorphism	✅	same method name, different behavior
Duck Typing	✅	JS special — method hona chahiye, class check nahi