DAY-1 (Introduction)

JS stands for JavaScript, which is a programming language commonly used to create interactive effects within web browsers. It allows web developers to add dynamic content to websites, such as animations, form validation, and interactivity without needing to reload the page. JavaScript is essential for modern web development and works alongside HTML and CSS to create rich, interactive web applications. It can also be used on the server-side (with environments like Node.js).

1995 - Creation of JavaScript: JavaScript was created by Brendan Eich at Netscape Communications. It was initially developed under the name Mocha, then changed to LiveScript, and finally renamed JavaScript. The first version was developed in just 10 days.

1996 - JavaScript and JScript: In 1996, Microsoft created its own version of JavaScript called JScript, which was introduced in Internet Explorer. This led to some compatibility issues between browsers.

1997 - Standardization: JavaScript was standardized by ECMA International under the name ECMAScript. The first official standard, ECMAScript 1, was published in 1997.

1999 - ECMAScript 3: This version became the most widely used version for many years, adding important features like regular expressions, try/catch handling for errors, and better string handling.

2005 - AJAX and Web 2.0: JavaScript became even more critical with the rise of AJAX (Asynchronous JavaScript and XML), which allowed web pages to update dynamically without needing to reload. This period marked the beginning of more interactive websites and the Web 2.0 era.

2009 - Node.js and ECMAScript 5: In 2009, Node.js was introduced, enabling JavaScript to be used on the server-side, breaking the browser-only limitation. Meanwhile, ECMAScript 5 (ES5) introduced important features like strict mode and JSON handling.

2015 - ECMAScript 6 (ES6): Also known as ECMAScript 2015, this was a major update that introduced features like arrow functions, classes, modules, promises, and let/const for variable declarations. This helped modernize the language and make it more concise and powerful. Post-2015 - Continuous Evolution: Since ECMAScript 2015, the standard has been updated annually with incremental improvements. Features like async/await, spread operator, destructuring, and optional chaining have been added in later versions, improving the developer experience. JavaScript has evolved from a simple scripting language into one of the most important and versatile programming languages, used not only for client-side scripting but also for server-side programming, mobile apps, desktop applications, and more

DAY-2 (Basic Syntax and Variables ,Data Types in JavaScript, Type Conversion,Basic Operators)

Basic Syntax and Variables

1. Declaring Variables with `let`, `const`, and `var`

In JavaScript, we use three keywords to declare variables: `let`, `const`, and `var`.

- **`let`: Allows you to declare variables that can be reassigned later. It has block-level scope (meaning it's only available within the block it's defined in, such as a loop or function).**

```
let name = "Alice"; // Declaring a variable using let
name = "Bob"; // Reassigning the value of name
```

- **`const`: Used for declaring variables that cannot be reassigned after they are initialized. `const` also has block-level scope.**

```
const birthYear = 1990; // Declaring a constant
// birthYear = 1991; // Error! Cannot reassign to a constant
```

- **`var`: An older way of declaring variables. It has function-level scope (or global scope if declared outside a function), meaning it is accessible throughout the entire function where it's declared (or globally, if declared outside a function). Avoid using `var` for new code because it can lead to unexpected behavior.**

```
var city = "Paris"; // Declaring a variable with var
```

2. Data Types in JavaScript

JavaScript has several fundamental data types:

- **String: Represents a sequence of characters.**

```
let name = "Alice";
```

- **Number: Represents both integer and floating-point numbers.**

```
let age = 25; // Integer
let price = 19.99; // Floating-point number
```

- **Boolean: Represents true or false values.**

```
let isStudent = true;
let isActive = false;
```

- **Null: Represents the intentional absence of any value.**

```
let emptyValue = null;
```

- **Undefined: Indicates that a variable has been declared but has not been assigned a value.**

```
let notAssigned;
console.log(notAssigned); // undefined
```

3. Type Conversion

In JavaScript, type conversion can be implicit (automatically done by JavaScript) or explicit (done manually by the programmer).

- **Implicit Type Conversion: JavaScript automatically converts data types when necessary.**

```
let x = "5" + 2; // Implicit conversion, x becomes "52" (string)
let y = "5" - 2; // Implicit conversion, y becomes 3 (number)
```

- **Explicit Type Conversion: You can manually convert data types using functions like `String()`, `Number()`, `Boolean()`, etc.**

```
let str = String(123); // Explicitly convert number to string
let num = Number("123"); // Explicitly convert string to number
let bool = Boolean(1); // Explicitly convert to boolean (1 is truthy)
```

4. Basic Operators

- **Arithmetic Operators: Used for mathematical calculations.**

```
let sum = 5 + 3; // Addition
let diff = 5 - 3; // Subtraction
let product = 5 * 3; // Multiplication
let quotient = 5 / 3; // Division
let remainder = 5 % 3; // Modulo (remainder of division)
```

- **Assignment Operators: Used to assign values to variables.**

```
let x = 10; // Basic assignment
x += 5; // Equivalent to x = x + 5
x -= 3; // Equivalent to x = x - 3
x *= 2; // Equivalent to x = x * 2
x /= 4; // Equivalent to x = x / 4
```

- **Comparison Operators: Used to compare two values.**

```
let a = 5;
let b = 10;

console.log(a == b); // Equal to (value)
console.log(a === b); // Strict equal to (value and type)
console.log(a != b); // Not equal to
console.log(a !== b); // Strict not equal to
console.log(a > b); // Greater than
console.log(a < b); // Less than
```

- **Logical Operators: Used for logical operations (often in conditions).**

```
let isAdult = true;
let hasTicket = false;

console.log(isAdult && hasTicket); // AND: true only if both are true
console.log(isAdult || hasTicket); // OR: true if either is true
console.log(!isAdult); // NOT: inverts the boolean value
```

DAY-3 (Control Flow and Conditional Statements)

Control flow is a key concept in programming that determines the order in which different sections of code are executed. In most languages, control flow is handled using conditional statements (like `if`, `else`, `switch`, and ternary operators) that allow the program to execute different code depending on certain conditions.

1. `if`, `else if`, and `else` Statements

These are the most common way of making decisions in programming.

- **`if` statement: The `if` statement allows you to execute a block of code only if a specified condition evaluates to true.**
- **`else if` statement: You can use `else if` to test additional conditions if the previous `if` condition was false.**
- **`else` statement: If none of the `if` or `else if` conditions are true, the `else` block will execute.**

Syntax Example:

```
let number = 10;


if (number > 10) {
      console.log("Number is greater than 10");
} else if (number === 10) {
      console.log("Number is exactly 10");
} else {
      console.log("Number is less than 10");
}
```

- **In this example, the program checks if `number` is greater than, equal to, or less than 10 and prints the corresponding message.**

2. `switch` Statement

The `switch` statement allows you to compare a variable with multiple possible values and execute different code blocks based on which value the variable matches.

Syntax Example:

```
let day = "Monday";


switch (day) {
	case "Monday":
	console.log("Start of the week");
	break;
	case "Tuesday":
	console.log("Second day of the week");
	break;
	case "Wednesday":
	console.log("Midweek");
	break;
	default:
	console.log("Another day");
	break;
}
```

- **The `switch` statement compares the value of `day` to each `case`. If a match is found, it executes the corresponding block of code.**
- **The `break` keyword prevents the code from "falling through" to the next case.**
- **The `default` block runs if no cases match.**

3. Ternary Operator

The ternary operator is a shorthand for an `if-else` statement. It evaluates a condition and returns one of two values based on whether the condition is true or false.

Syntax Example:

```
let age = 18;

let isAdult = age >= 18 ? "Yes" : "No";

console.log(isAdult);  // Output: "Yes"
```

- **The expression `age >= 18 ? "Yes" : "No"` means if `age` is greater than or equal to 18, it returns `"Yes"`, otherwise, it returns `"No"`.**
- **It is a concise way to handle simple conditionals.**

4. Truthy and Falsy Values

In JavaScript (and many other languages), certain values are considered "truthy" or "falsy" when evaluated in a boolean context (such as in `if` or `while` conditions).

- **Falsy values: These values are considered `false` in boolean context.**
  - **`false`**
  - **`0`**
  - **`""` (empty string)**
  - **`null`**
  - **`undefined`**
  - **`NaN`**
- **Truthy values: Everything that is not falsy is considered truthy, including:**
  - **Non-zero numbers (e.g., `1`, `-1`)**
  - **Non-empty strings (e.g., `"hello"`)**
  - **Objects (e.g., `{}`, `[]`)**
  - **`true`**

Example of Truthy and Falsy:

```
let value = 0;


if (value) {
```

```
        console.log("This is truthy");
} else {
        console.log("This is falsy");  // Output: "This is falsy"
}
```

- **In this case, `value` is `0`, which is a falsy value, so the `else` block executes.**

Summary of Concepts:

- **`if`, `else if`, `else`: Used to make decisions based on conditions.**
- **`switch`: Useful for checking a variable against multiple possible values.**
- **Ternary Operator: A shorthand for `if-else` that evaluates a condition and returns one of two values.**
- **Truthy and Falsy: Values that evaluate as `true` or `false` in boolean contexts.**

These control structures help you manage the flow of your program and ensure the right logic is executed under the right conditions.

DAY-4 (Loops and Iteration)

Loops are used to repeatedly execute a block of code as long as a given condition is met. In JavaScript, there are several types of loops, and they allow us to iterate over arrays, objects, or even specific ranges of values.

1. for, while, and do...while loops

for loop

The `for` loop is commonly used when the number of iterations is known ahead of time.

```
for (let i = 0; i < 5; i++) {
  console.log(i);  // Will log 0, 1, 2, 3, 4
}
```

- **The loop starts by initializing `i` to 0.**
- **It continues as long as `i < 5`.**
- **After each iteration, `i` is incremented by 1.**

while loop

The `while` loop is used when you don't know how many times you need to iterate, but you know the condition that must be met to stop looping.

```
let i = 0;
while (i < 5) {
  console.log(i);  // Will log 0, 1, 2, 3, 4
  i++;
}
```

- **The condition `i < 5` is checked before every iteration.**
- **The loop will exit once `i` is no longer less than 5.**

do...while loop

The `do...while` loop ensures that the code block is executed at least once before the condition is checked.

```
let i = 0;

do {

  console.log(i);  // Will log 0, 1, 2, 3, 4

  i++;

} while (i < 5);
```

- **The loop will run the block of code once, and then check the condition after each iteration.**

---

2. for...of and for...in loops

for...of loop

The `for...of` loop is used to iterate over iterable objects like arrays or strings.

```
const fruits = ["apple", "banana", "cherry"];

for (let fruit of fruits) {

  console.log(fruit);  // Logs each fruit: apple, banana, cherry

}
```

- **This loop directly gives you the values from the iterable, not the indexes.**

for...in loop

The `for...in` loop is used to iterate over object properties or indexes of arrays.

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

for (let key in person) {

  console.log(key, person[key]);  // Logs "name John", "age 30", "city New
York"
```

```
}
```

- **This loop gives you the keys (or property names) of the object.**

For arrays, `for...in` gives you the indices:

```
const fruits = ["apple", "banana", "cherry"];

for (let index in fruits) {

  console.log(index, fruits[index]);  // Logs "0 apple", "1 banana", "2 cherry"

}
```

Note: Use `for...in` for objects and `for...of` for arrays.

---

3. Using break and continue

- **break: Stops the loop completely when a certain condition is met.**
- **continue: Skips the current iteration and moves to the next one.**

Example using `break`:

```
for (let i = 0; i < 5; i++) {

  if (i === 3) {

      break;  // Exit the loop when i is 3

  }

  console.log(i);  // Logs 0, 1, 2

}
```

Example using `continue`:

```
for (let i = 0; i < 5; i++) {

  if (i === 3) {

      continue;  // Skip this iteration when i is 3
```

```
  }

  console.log(i);  // Logs 0, 1, 2, 4

}
```

---

4. Iterating through arrays and objects

- **Arrays can be iterated using a `for` loop, `for...of`, or `forEach` method.**

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

// Using for...of
for (let number of numbers) {
  console.log(number);  // Logs 1, 2, 3, 4, 5
}

// Using forEach
numbers.forEach(num => console.log(num));  // Logs 1, 2, 3, 4, 5
```

- **Objects can be iterated using a `for...in` loop or `Object.keys()` combined with `forEach`.**

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

// Using for...in
for (let key in person) {
  console.log(key, person[key]);  // Logs "name John", "age 30", "city
New York"
}

// Using Object.keys()
Object.keys(person).forEach(key => {
  console.log(key, person[key]);  // Logs "name John", "age 30", "city
New York"
});
```

DAY-5 (Functions and Scope)

---

1. Defining Functions with the `function` Keyword

A function is a block of code designed to perform a particular task. The syntax for defining a function in JavaScript using the `function` keyword looks like this:

```
function myFunction() {

      console.log("Hello, World!");

}


myFunction();  // Outputs: Hello, World!
```

- **`myFunction` is the function name.**
- **The code inside the curly braces `{}` is the function body.**
- **Functions can take parameters and return values.**

---

2. Function Expressions and Arrow Functions

Function Expression: A function expression is when you define a function and assign it to a variable. It is different from a function declaration because it is not hoisted.

```
const greet = function() {

      console.log("Hello!");

};


greet();  // Outputs: Hello!
```

Arrow Functions: Arrow functions provide a more concise way of writing functions. They are often used for shorter functions and maintain lexical scoping of `this`.

```
const greet = () => {
      console.log("Hello!");
};


greet();  // Outputs: Hello!
```

Arrow functions are particularly useful for functions that have a small body. Here's an example of a more compact one:

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


console.log(add(3, 4));  // Outputs: 7
```

---

3. Parameters, Arguments, and Return Values

- **Parameters are the variables listed in the function definition.**
- **Arguments are the values passed to the function when it is called.**
- **Return Values are the values that a function sends back after execution.**

Example:

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


console.log(add(5, 3));  // Outputs: 8
```

Here, `a` and `b` are parameters, and `5` and `3` are arguments. The function returns `8`.

---

4. Default Parameters and Rest Parameters

- **Default Parameters: You can assign default values to function parameters in case the caller does not provide a value.**

```
function greet(name = "Guest") {
      console.log(`Hello, ${name}`);
}

greet();          // Outputs: Hello, Guest
greet("Alice");   // Outputs: Hello, Alice
```

- **Rest Parameters: You can use the `...` syntax to gather all remaining arguments into an array.**

```
function sum(...numbers) {
      return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sum(1, 2, 3, 4));  // Outputs: 10
```

Here, `numbers` will be an array of all the arguments passed to the `sum` function.

---

5. Scopes: Block Scope, Function Scope, Global Scope

- **Global Scope: Variables declared outside any function or block are in the global scope. They are accessible throughout the entire program.**

```
let globalVar = "I am global";

function showGlobal() {
      console.log(globalVar);  // Outputs: I am global
```

```
}


showGlobal();
```

- **Function Scope: Variables declared inside a function are only accessible within that function.**

```
function myFunction() {

      let localVar = "I am local";

      console.log(localVar);  // Outputs: I am local

}


myFunction();

// console.log(localVar);  // Error: localVar is not defined
```

- **Block Scope: Variables declared with `let` or `const` inside a block (like an `if` statement or loop) are limited to that block.**

```
if (true) {

      let blockScoped = "I am block scoped";

    console.log(blockScoped);  // Outputs: I am block scoped

}


// console.log(blockScoped);  // Error: blockScoped is not defined
```

---

6. Closures and Lexical Scope

- **Closure: A closure is a function that retains access to its lexical scope (the environment in which it was created) even after the function has returned. This allows functions to "remember" variables from their outer scope.**

```
function outer() {
      let outerVar = "I am from outer";

      function inner() {
      console.log(outerVar);  // Accesses the outer scope
      }

      return inner;
}

const closureFunc = outer();
closureFunc();  // Outputs: I am from outer
```

- **Lexical Scope: In JavaScript, the scope of a variable is determined by where it is declared, not by where it is called. This is known as lexical scoping.**

```
let x = 10;

function outer() {
      let x = 20;

      function inner() {
      console.log(x);  // Lexically scoped to the outer function
      }

      inner();
```

```
}


outer();  // Outputs: 20 (The inner function looks up to the nearest scope,
which is the outer function's scope)
```

Summary

- **Function Definitions: Use the `function` keyword for function declarations or assign anonymous functions to variables for expressions.**
- **Function Expressions vs. Arrow Functions: Arrow functions are concise and do not bind their own `this` value.**
- **Parameters/Arguments: Functions can accept parameters and return values. Default parameters and rest parameters offer flexibility.**
- **Scopes: JavaScript has global, function, and block scopes to determine where variables are accessible.**
- **Closures and Lexical Scope: Functions can "close over" their scope, maintaining access to outer variables even after they are no longer in scope.**

DAY-6 (Arrays)

Creating and Using Arrays

Arrays are used to store multiple values in a single variable. They can store elements of any type.

```
let fruits = ['apple', 'banana', 'orange'];
```

Looping Through Arrays

You can loop through an array using several methods:

- `for` **loop: Standard looping.**

```
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

- `for...of` **loop: A cleaner way to loop through arrays.**

```
for (let fruit of fruits) {
  console.log(fruit);
}
```

- `forEach()`**: Iterates over each element in an array.**

```
fruits.forEach(fruit => console.log(fruit));
```

Destructuring Arrays

Destructuring allows you to extract values from arrays or objects and assign them to variables.

- **Array Destructuring:**

```
let [first, second] = fruits;

console.log(first); // "apple"

console.log(second); // "banana"
```

```
Spread and Rest Operators (...)

The spread operator (...) is used to unpack or spread elements of an array or
object into a new array or object.
```

- Spread in Arrays:

```
let newFruits = [...fruits, 'grape', 'mango'];

console.log(newFruits); // ["apple", "banana", "orange", "grape", "mango"]
```

- **Rest in Arrays:**

```
let [first, ...rest] = fruits;

console.log(first); // "apple"

console.log(rest); // ["banana", "orange"]
```

Array Creation Methods

- `Array()`
  Creates a new array.

  ```
  let arr = new Array(3);  // Creates an array with 3 empty slots
  console.log(arr);  // [ <3 empty items> ]
  ```

- `Array.from()`

  Creates a new array from an array-like or iterable object.

```
let str = "Hello";
let arr = Array.from(str);
console.log(arr);  // ['H', 'e', 'l', 'l', 'o']
```

- `Array.isArray()`

  Checks if a value is an array.

```
console.log(Array.isArray([1, 2, 3]));  // true
console.log(Array.isArray('hello'));// false
```

2. Array Access Methods

- `concat()`

  Merges two or more arrays.

```
let arr1 = [1, 2];
let arr2 = [3, 4];
let combined = arr1.concat(arr2);
console.log(combined);  // [1, 2, 3, 4]
```

- `copyWithin()`

  Copies a section of the array to another location within the same array.

```
let arr = [1, 2, 3, 4, 5];
arr.copyWithin(0, 3);
console.log(arr);  // [4, 5, 3, 4, 5]
```

- `every()`

  Tests whether all elements in the array pass the test implemented by the provided function.

```
let arr = [2, 4, 6];
let result = arr.every(num => num % 2 === 0);
console.log(result);  // true
```

- `fill()`

  Fills all elements in an array with a static value.

```
let arr = [1, 2, 3, 4];
arr.fill(0, 1, 3);
console.log(arr);  // [1, 0, 0, 4]
```

- `find()`

  Returns the first element that satisfies the provided testing function.

```
let arr = [5, 12, 8, 130, 44];
let found = arr.find(num => num > 10);
console.log(found);  // 12
```

- `findIndex()`

  Returns the index of the first element that satisfies the provided testing function.

```
let arr = [5, 12, 8, 130, 44];
let index = arr.findIndex(num => num > 10);
console.log(index);  // 1
```

- `flat()`

  Flattens a nested array into a single-level array.

```
let arr = [1, [2, 3], [4, [5, 6]]];
let flattened = arr.flat(2);
console.log(flattened);  // [1, 2, 3, 4, 5, 6]
```

- `forEach()`

  Executes a provided function once for each array element.

```
let arr = [1, 2, 3];
arr.forEach(num => console.log(num));  // 1, 2, 3
```

- `includes()`

  Determines whether an array contains a certain element.

```
let arr = [1, 2, 3];
console.log(arr.includes(2));  // true
console.log(arr.includes(4));  // false
```

- `indexOf()`

  Returns the first index at which a given element is found.

```
let arr = [1, 2, 3, 2];
console.log(arr.indexOf(2));  // 1
```

- `join()`

  Joins all the elements of an array into a string.

```
let arr = ['a', 'b', 'c'];
```

```
console.log(arr.join('-'));  // 'a-b-c'
```

- `lastIndexOf()`

  Returns the last index at which a given element is found.

```
let arr = [1, 2, 3, 2];
console.log(arr.lastIndexOf(2));  // 3
```

- `map()`

  Creates a new array with the results of calling a provided function on every element.

```
let arr = [1, 2, 3];
let doubled = arr.map(num => num * 2);
console.log(doubled);  // [2, 4, 6]
```

- `pop()`

  Removes the last element from an array and returns that element.

```
let arr = [1, 2, 3];
let lastElement = arr.pop();
console.log(lastElement);  // 3
console.log(arr);  // [1, 2]
```

- `push()`

  Adds one or more elements to the end of an array and returns the new length.

```
let arr = [1, 2];
arr.push(3);
console.log(arr);  // [1, 2, 3]
```

- `reduce()`

  Applies a function to reduce the array to a single value.

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

- `reverse()`

  Reverses the order of the elements in an array.

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

```
arr.reverse();
console.log(arr);  // [3, 2, 1]
```

- `shift()`

  Removes the first element from an array and returns that element.

```
let arr = [1, 2, 3];
let firstElement = arr.shift();
console.log(firstElement);  // 1
console.log(arr);  // [2, 3]
```

- `slice()`

  Returns a shallow copy of a portion of an array into a new array.

```
let arr = [1, 2, 3, 4];
let sliced = arr.slice(1, 3);
console.log(sliced);  // [2, 3]
```

- `some()`

  Tests whether at least one element in the array passes the test implemented by the provided function.

```
let arr = [1, 2, 3];
let result = arr.some(num => num > 2);
console.log(result);  // true
```

- `sort()`

  Sorts the elements of an array in place.

```
let arr = [3, 1, 2];
arr.sort();
console.log(arr);  // [1, 2, 3]
```

- `splice()`

  Adds/removes elements from any position in an array.

```
let arr = [1, 2, 3, 4];
arr.splice(1, 2, 'a', 'b');
console.log(arr);  // [1, 'a', 'b', 4]
```

- `toLocaleString()`

  Returns a localized string representation of the array.

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

```
console.log(arr.toLocaleString());  // '1,2,3'
```

- `toString()`
  Returns a string representing the array and its elements.

```
let arr = [1, 2, 3];
console.log(arr.toString());  // '1,2,3'
```

- `unshift()`
  Adds one or more elements to the beginning of an array and returns the new length.

```
let arr = [2, 3];
arr.unshift(1);
console.log(arr);  // [1, 2, 3]
```

- `values()`
  Returns a new array iterator object that contains the values of each array element.

```
let arr = ['a', 'b', 'c'];
let iterator = arr.values();
console.log(iterator.next().value);  // 'a'
```

---

3. Array Iteration Methods

- `forEach()`
  Executes a provided function once for each array element.

```
let arr = [1, 2, 3];
arr.forEach((element) => console.log(element));  // 1, 2, 3
```

- `map()`
  Creates a new array with the results of calling a provided function on every element.

```
let arr = [1, 2, 3];
let doubled = arr.map(num => num * 2);
console.log(doubled);  // [2, 4, 6]
```

- `filter()`
  Creates a new array with all elements that pass the provided test function.

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

```
let evenNumbers = arr.filter(num => num % 2 === 0);
console.log(evenNumbers);  // [2, 4]
```

- `reduce()`

  Reduces the array to a single value by applying a function on each element.

```
let arr = [1, 2, 3];
let sum = arr.reduce((acc, num) => acc + num, 0);
console.log(sum);  // 6
```

- `some()`

  Tests whether at least one element in the array satisfies the provided function.

```
let arr = [1, 2, 3];
let result = arr.some(num => num > 2);
console.log(result);  // true
```

DAY-7 (Objects)

Creating and Accessing Objects

Objects are collections of key-value pairs.

```
let person = {
  name: 'John',
  age: 30,
  greet() {
      console.log('Hello!');
  }
};

// Accessing object properties
console.log(person.name); // "John"
console.log(person['age']); // 30
```

Object Methods

- `Object.keys()`: **Returns an array of the object's property names.**

```
let keys = Object.keys(person);
console.log(keys); // ["name", "age", "greet"]
```

- `Object.values()`: **Returns an array of the object's values.**

```
let values = Object.values(person);
console.log(values); // ["John", 30, ƒ]
```

- `Object.entries()`: **Returns an array of key-value pairs as arrays.**

```
let entries = Object.entries(person);

console.log(entries); // [["name", "John"], ["age", 30], ["greet", ƒ]]
```

Destructuring Objects

Destructuring allows you to extract values from arrays or objects and assign them to variables.

- **Object Destructuring:**

```
let { name, age } = person;

console.log(name); // "John"

console.log(age); // 30
```

Spread and Rest Operators (. . .)

The spread operator (. . .) is used to unpack or spread elements of an array or object into object.

- **Spread in Objects:**

```
let newPerson = { ...person, gender: 'male' };

console.log(newPerson); // { name: 'John', age: 30, greet: ƒ, gender: 'male' }
```

The rest operator (. . .) is used to collect multiple elements and put them into an object.

- **Rest in Objects:**

```
let { name, ...otherDetails } = person;

console.log(name); // "John"

console.log(otherDetails); // { age: 30, greet: ƒ }
```

Using `Object.assign()` to copy methods from another object

The `Object.assign()` static method copies all from one or more *source objects* to a *target object*. It returns the modified target object.

```
const target = { a: 1, b: 2 };

const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);

// Expected output: Object { a: 1, b: 4, c: 5 }
```

```
let obj = {a: 1, b: 2};
 let copy = Object.assign({}, obj);
 obj.a = 5;

 console.log(copy.a);
```

The JavaScript call() Method

The call() method is a predefined JavaScript method.

It can be used to invoke (call) a method with an object as an argument (parameter).

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
 const person1 = {
```

```
  firstName:"John",
  lastName: "Doe"
}
const person2 = {
  firstName:"Mary",
  lastName: "Doe"
}

// This will return "John Doe":
person.fullName.call(person1);
```

DAY-8 (Strings and String Manipulation ,Template Literals with Backticks)

Basic String Methods

Here are some commonly used string methods in JavaScript:

1. `length`

   The `length` property returns the number of characters in a string.

```
let str = "Hello, World!";
console.log(str.length);  // 13
```

2. `charAt(index)`

   The `charAt()` method returns the character at a specified index (position) in a string.

```
let str = "Hello";
console.log(str.charAt(1));  // "e"
```

3. `indexOf(searchValue)`

   The `indexOf()` method returns the index of the first occurrence of the specified value. If not found, it returns `-1`.

```
let str = "Hello, World!";
console.log(str.indexOf('o'));  // 4
console.log(str.indexOf('z'));  // -1
```

4. `slice(beginIndex, endIndex)`

   The `slice()` method extracts a section of a string and returns it as a new string, without modifying the original string.

```
let str = "Hello, World!";
console.log(str.slice(0, 5));  // "Hello"
console.log(str.slice(7));    // "World!"
```

5. `split(separator)`

   The `split()` method divides a string into an array of substrings, based on a specified separator.

```
let str = "apple,banana,cherry";
```

```
let fruits = str.split(",");
console.log(fruits);  // ["apple", "banana", "cherry"]
```

6. **toUpperCase()**

Converts all characters in a string to uppercase.

let str = "hello";

console.log(str.toUpperCase()); // "HELLO"

7. **toLowerCase()**

Converts all characters in a string to lowercase.

let str = "HELLO";

console.log(str.toLowerCase()); // "hello"

8. **trim()**

Removes whitespace from both ends of a string.

let str = "   Hello, World!   ";

console.log(str.trim()); // "Hello, World!"

9. **replace(searchValue, newValue)**

Replaces the first occurrence of a specified value with another value.

let str = "Hello, World!";

```
let result = str.replace("World", "Universe");

console.log(result);  // "Hello, Universe!"
```

10. `replaceAll(searchValue, newValue)`

Replaces all occurrences of a specified value with another value.

```
let str = "Hello, World! Hello, Universe!";

let result = str.replaceAll("Hello", "Hi");

console.log(result);  // "Hi, World! Hi, Universe!"
```

11. `includes(searchValue)`

Checks if a string contains a specified value. Returns `true` if the value is found, otherwise `false`.

```
let str = "Hello, World!";

console.log(str.includes("World"));  // true

console.log(str.includes("JavaScript"));  // false
```

12. `startsWith(searchValue)`

Checks if a string starts with the specified value.

```
let str = "Hello, World!";

console.log(str.startsWith("Hello"));  // true

console.log(str.startsWith("World"));  // false
```

13. `endsWith(searchValue)`

Checks if a string ends with the specified value.

```
let str = "Hello, World!";
console.log(str.endsWith("World!"));  // true
console.log(str.endsWith("Hello"));  // false
```

14. `concat(string2, string3, ...)`

Joins two or more strings and returns a new string.

```
let str1 = "Hello";
let str2 = "World";
let result = str1.concat(", ", str2, "!");
console.log(result);  // "Hello, World!"
```

15. `repeat(count)`

Returns a new string that is the result of repeating the original string a specified number of times.

```
let str = "Hello";
console.log(str.repeat(3));  // "HelloHelloHello"
```

Template Literals with Backticks

Template literals (introduced in ES6) allow for embedding expressions inside strings using backticks (``) and `${}`.

```
let name = "Alice";

let age = 25;

let greeting = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(greeting);  // "Hello, my name is Alice and I am 25 years old."
```

You can also perform expressions inside `${}`:

```
let x = 5;

let y = 10;

let result = `The sum of ${x} and ${y} is ${x + y}.`;

console.log(result);  // "The sum of 5 and 10 is 15."
```

String Concatenation

String concatenation is the process of joining two or more strings together.

1. **Using the + Operator:**

   ```
   let str1 = "Hello";
   let str2 = "World";
   let result = str1 + ", " + str2 + "!";
   console.log(result);  // "Hello, World!"
   ```

2. **Using `concat()` Method: The `concat()` method joins two or more strings.**

   ```
   let str1 = "Hello";
   let str2 = "World";
   let result = str1.concat(", ", str2, "!");
   console.log(result);  // "Hello, World!"
   ```

3. **Using Template Literals (ES6+):**

```
let str1 = "Hello";
let str2 = "World";
let result = `${str1}, ${str2}!`;
console.log(result);  // "Hello, World!"
```

DAY-9 (Numbers and Math ,try catch finally block)

JavaScript provides various ways to work with numbers, perform basic arithmetic, and handle precision. Here's an overview of working with numbers, the `Math` object, and number properties and methods.

Working with Numbers and Basic Arithmetic

1. **Basic Arithmetic Operations: You can perform basic arithmetic operations directly on numbers:**

```
let a = 10;
let b = 5;

console.log(a + b); // Addition: 15
console.log(a - b); // Subtraction: 5
console.log(a * b); // Multiplication: 50
console.log(a / b); // Division: 2
console.log(a % b); // Modulo (remainder): 0
console.log(a ** b); // Exponentiation: 100000
```

2. **Increment and Decrement: You can increment or decrement a number using `++` and `--`:**

```
let num = 5;
num++; // num becomes 6
num--; // num becomes 5
```

3. **Unary Plus (+): The unary `+` operator can be used to convert a value to a number:**

```
let str = "10";
let num = +str; // Converts "10" to the number 10
```

`Math` Object and Methods

The `Math` object provides mathematical constants and methods for performing mathematical tasks.

1. **Basic Methods:**
    - `Math.abs(x)`: Returns the absolute value of `x`:

      ```
      console.log(Math.abs(-10)); // 10
      ```

    - `Math.round(x)`: Rounds `x` to the nearest integer:

      ```
      console.log(Math.round(5.4)); // 5
      console.log(Math.round(5.6)); // 6
      ```

    - `Math.ceil(x)`: Rounds `x` up to the nearest integer:

      ```
      console.log(Math.ceil(4.1)); // 5
      ```

    - `Math.floor(x)`: Rounds `x` down to the nearest integer:

      ```
      console.log(Math.floor(4.9)); // 4
      ```

    - `Math.max(x, y, z, ...)`: Returns the largest of the given numbers:

      ```
      console.log(Math.max(1, 2, 3, 4)); // 4
      ```

    - `Math.min(x, y, z, ...)`: Returns the smallest of the given numbers:

      ```
      console.log(Math.min(1, 2, 3, 4)); // 1
      ```

    - `Math.random()`: Returns a random floating-point number between 0 and 1:

      ```
      console.log(Math.random()); // A random number like 0.234
      ```

    - `Math.pow(x, y)`: Returns `x` raised to the power of `y`:

      ```
      console.log(Math.pow(2, 3)); // 8
      ```

    - `Math.sqrt(x)`: Returns the square root of `x`:

      ```
      console.log(Math.sqrt(16)); // 4
      ```

    - `Math.PI`: A constant that represents the value of π (approximately 3.14159):

      ```
      console.log(Math.PI); // 3.141592653589793
      ```

Number Properties and Methods

1. `Number.isInteger(value)`: Checks if the value is an integer:

```
console.log(Number.isInteger(4)); // true
console.log(Number.isInteger(4.5)); // false
```

2. `Number.parseInt(value)`: Converts a string to an integer:

```
console.log(Number.parseInt("10")); // 10
console.log(Number.parseInt("10.5")); // 10
```

3. `Number.parseFloat(value)`: Converts a string to a floating-point number:

```
console.log(Number.parseFloat("10.5")); // 10.5
console.log(Number.parseFloat("10")); // 10
```

4. `Number.toFixed(digits)`: Formats a number to a fixed number of decimal places:

```
let num = 12.34567;
console.log(num.toFixed(2)); // "12.35" (rounded to two decimal places)
```

5. `Number.isNaN(value)`: Checks if a value is NaN (Not-a-Number):

```
console.log(Number.isNaN(100)); // false
console.log(Number.isNaN(NaN)); // true
```

6. `Number.isFinite(value)`: Checks if the value is a finite number:

```
console.log(Number.isFinite(100)); // true
console.log(Number.isFinite(Infinity)); // false
```

7. `Number.toString([radix])`: Converts a number to a string representation. You can also specify the base (radix) for the conversion:

```
let num = 255;
console.log(num.toString(16)); // "ff" (hexadecimal)
console.log(num.toString(2));  // "11111111" (binary)
```

Error Handling

In JavaScript, managing errors and debugging is crucial for writing robust and reliable code. Below is an overview of error handling and debugging techniques.

Types of Errors

1. **Syntax Errors: These errors occur when the code structure is incorrect, such as missing parentheses, brackets, or a semicolon where it's expected.**
    - **Example:**

```
var x = 5
console.log(x)
// SyntaxError: missing semicolon
```

2. **Runtime Errors: These errors occur while the program is running, typically due to operations that can fail, such as undefined variables or incorrect function calls.**
    - **Example:**

```
var x;
console.log(x.toUpperCase()); // TypeError: Cannot read property
'toUpperCase' of undefined
```

3. **Logical Errors: These errors occur when the code runs without crashing but produces incorrect results due to faulty logic.**
    - **Example:**

```
function sum(a, b) {
  return a - b; // Logical error: should be a + b
}
console.log(sum(2, 3)); // Outputs: -1
```

`try...catch...finally` for Handling Exceptions

The `try...catch...finally` statement allows you to handle exceptions in a more controlled manner.

- **`try` block: The code that may potentially throw an error is written here.**
- **`catch` block: The error handling logic, where you can access the error object.**
- **`finally` block: The code inside this block always executes, regardless of whether an error occurred or not (useful for cleanup actions).**

DAY-10 (JavaScript and the DOM (Document Object Model))

The DOM (Document Object Model) is a programming interface for web documents. It represents the structure of a document as a tree of nodes, where each node represents part of the page (such as elements, attributes, and text). JavaScript allows you to interact with and manipulate the DOM to change the content and structure of a webpage dynamically.

1. DOM Structure and Hierarchy

The DOM represents an HTML document as a tree structure, with each element, attribute, and piece of text as a node. The hierarchy usually looks like this:

- **Document Node: The root node representing the entire document.**
  - **Element Nodes: Represent HTML tags like `<html>`, `<body>`, `<div>`, etc.**
    - **Attribute Nodes: Represent HTML attributes like `id`, `class`, etc.**
    - **Text Nodes: Represent the text inside the HTML elements.**

2. Selecting Elements

JavaScript provides several methods to select and interact with HTML elements in the DOM.

- **`getElementById()`: Selects an element by its unique `id` attribute.**

  ```
  const element = document.getElementById("myElement");
  ```

  This will return the element with `id="myElement"`.

- **`querySelector()`: Selects the first element that matches a specified CSS selector.**

  ```
  const element = document.querySelector(".myClass");
  ```

  This will return the first element with the class `.myClass`.

- **`querySelectorAll()`: Selects all elements that match a specified CSS selector, returning a NodeList.**

  ```
  const elements = document.querySelectorAll("div.myClass")[i];
  ```

  This will return all `<div>` elements with the class `.myClass`.

3. Manipulating Elements

- `innerHTML`: Used to get or set the HTML content inside an element.

```
element.innerHTML = "<p>New content here!</p>";
```

- `textContent`: Used to get or set the text content of an element (ignores HTML tags).

```
element.textContent = "Updated text content";
```

- `style`: Allows direct manipulation of the element's CSS styles.

```
element.style.color = "blue";
element.style.fontSize = "20px";
```

4. Creating and Appending Elements

- Creating Elements: Use `createElement()` to create new HTML elements.

```
const newDiv = document.createElement("div");
newDiv.textContent = "This is a new div!";
```

- Appending Elements:Use `appendChild()`, `append()` to add new elements to the dom

```
document.body.appendChild(newDiv);
```

- Inserting Elements: Use `insertBefore()` to insert an element before a specific child.

```
const parent = document.getElementById("parentElement");
const newElement = document.createElement("p");
parent.insertBefore(newElement, parent.firstChild);
```

5. Working with Attributes

- `getAttribute()`: Retrieves the value of an attribute.

```
const classValue = element.getAttribute("class");
```

- `setAttribute()`: Sets or changes the value of an attribute.

```
element.setAttribute("class", "newClassName");
```

- `removeAttribute()`:Removes an attribute from element

```
element.removeAttribute("class");
```

DAY-11 (Event in js)

MOUSE EVENTS

In JavaScript, mouse events are used to detect when the user interacts with elements on a webpage using a mouse. There are various types of mouse events, and each one corresponds to a different type of action or interaction with the mouse. Here's a list of common mouse events with explanations:

1. `click`

- Triggered when: The user presses and releases the mouse button on an element (like a button or a link).
- Example:

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

2. `dblclick`

- Triggered when: The user double-clicks the mouse on an element (pressing and releasing the mouse button twice in quick succession).
- Example:

```
document.getElementById('myElement').addEventListener('dblclick',
function() {
  alert('Element double-clicked!');
});
```

3. `mousedown`

- Triggered when: The user presses a mouse button down on an element.

- **Example:**

```
document.getElementById('myElement').addEventListener('mousedown',
function() {
  alert('Mouse button pressed!');
});
```

4. `mouseup`

- **Triggered when: The user releases the mouse button after pressing it down.**
- **Example:**

```
document.getElementById('myElement').addEventListener('mouseup',
function() {
  alert('Mouse button released!');
});
```

5. `mousemove`

- **Triggered when: The user moves the mouse over an element.**
- **Example:**

```
document.getElementById('myElement').addEventListener('mousemove',
function(event) {
  console.log('Mouse position: ' + event.clientX + ', ' +
event.clientY);
});
```

6. `mouseenter`

- **Triggered when: The mouse pointer enters the element's bounding box (does not bubble).**
- **Example:**

```
document.getElementById('myElement').addEventListener('mouseenter',
function() {
  console.log('Mouse entered element!');
});
```

7. `mouseleave`

- **Triggered when: The mouse pointer leaves the element's bounding box (does not bubble).**
- **Example:**

```
document.getElementById('myElement').addEventListener('mouseleave',
function() {
  console.log('Mouse left element!');
});
```

8. `mouseover`

- **Triggered when: The mouse pointer enters the element or any of its child elements (bubbles).**
- **Example:**

```
document.getElementById('myElement').addEventListener('mouseover',
function() {
  console.log('Mouse is over element or its child!');
});
```

9. `mouseout`

- **Triggered when: The mouse pointer leaves the element or any of its child elements (bubbles).**
- **Example:**

```
document.getElementById('myElement').addEventListener('mouseout',
function() {
  console.log('Mouse is out of element or its child!');
});
```

10. `contextmenu`

- **Triggered when: The user right-clicks on an element, typically bringing up the context menu.**
- **Example:**

```
document.getElementById('myElement').addEventListener('contextmenu',
function(event) {
  event.preventDefault();  // Prevents the default context menu
  console.log('Right-click detected!');
```

```
});
```

KEYBOARD EVENTS

In JavaScript, keyboard events are used to detect when a user interacts with the keyboard. These events can be used to capture specific key presses, keydowns, and keyup actions on the webpage. Here's a list of common keyboard events along with explanations:

1. `keydown`

- **Triggered when: The user presses a key on the keyboard. This event is fired when the key is initially pressed and before it repeats if the key is held down.**
- **Example:**

```
document.addEventListener('keydown', function(event) {
  console.log('Key pressed:', event.key);
});
```

2. `keypress` (Deprecated, use `keydown` or `input` instead)

- **Triggered when: The user presses a key that produces a character (like a letter or number) on the screen. It does not trigger for non-character keys like "Shift", "Arrow", or "Esc".**
- **Note: This event is considered deprecated and is no longer recommended for use in modern web development. It's better to use `keydown` for key press detection.**
- **Example (deprecated):**

```
document.addEventListener('keypress', function(event) {
  console.log('Key pressed:', event.key);
});
```

3. `keyup`

- **Triggered when: The user releases a key on the keyboard. This event is fired after the key has been released.**

- **Example:**

```
document.addEventListener('keyup', function(event) {
  console.log('Key released:', event.key);
});
```

Differences Between `keydown`, `keypress`, and `keyup`

- `keydown`: Fired as soon as the key is pressed down, it repeats if the key is held down.
- `keypress`: Was fired when a printable character key was pressed. Deprecated now in favor of `keydown`.
- `keyup`: Fired when the key is released.

* `mouseenter` VS `mouseover`

- `mouseenter` is triggered when the mouse enters the element itself, whereas `mouseover` is triggered when the mouse enters the element or any of its child elements.

* `mouseleave` VS `mouseout`

- `mouseleave` is triggered when the mouse leaves the element itself, while `mouseout` is triggered when the mouse leaves the element or any of its child elements.

WINDOW EVENTS

In JavaScript, window events are events that are triggered by actions occurring on the browser window. These events can be used to handle tasks such as page loading, resizing, scrolling, or interacting with the browser's history. Here's a list of common window events, along with explanations of what they do:

1. `load`

- **Triggered when: The entire page and all of its resources (like images, scripts, and stylesheets) have finished loading.**
- **Example:**

```
window.addEventListener('load', function() {
  console.log('Page has fully loaded!');
});
```

- **Use case: This is typically used for any actions that should happen after the page is completely ready.**

2. `resize`

- **Triggered when: The window is resized (when the user changes the size of the browser window).**
- **Example:**

```
window.addEventListener('resize', function() {
  console.log('Window has been resized!');
});
```

- **Use case: You can use this event to adjust layout or elements dynamically as the window size changes.**

3. `scroll`

- **Triggered when: The user scrolls the window (either horizontally or vertically).**
- **Example:**

```
window.addEventListener('scroll', function() {
  console.log('Window has been scrolled!');
});
```

Use case: Often used for infinite scrolling, detecting when the user has reached the bottom of the page, or animating elements based on scroll position.

4. `offline`

- **Triggered when: The browser detects that the device has lost internet connection.**
- **Example:**

```
window.addEventListener('offline', function() {
  console.log('The device is offline!');
});
```

- **Use case: This event is helpful for notifying the user when they have lost their internet connection.**

5. `online`

- **Triggered when: The browser detects that the device has reconnected to the internet.**
- **Example:**

```
window.addEventListener('online', function() {
  console.log('The device is now online!');
});
```

- **Use case: This event is helpful for detecting when the user has restored an internet connection after being offline.**

EVENT PROPAGATION:- CAPTURING AND BUBBLING AND STOPING PROPAGATION

1. Event Propagation

Event propagation refers to the way events move through the DOM tree when an event occurs on an element. There are two phases in event propagation:

- **Capturing Phase (Capture Phase)**: The event starts from the root of the DOM tree and travels down to the target element.
- **Bubbling Phase**: The event starts from the target element and propagates upwards to the root of the DOM tree.

By default, most events in JavaScript use the bubbling phase. The event starts at the target element and "bubbles up" to the parent elements (if they have listeners attached), reaching the root element.

2. Event Bubbling

Event Bubbling is the default behavior where the event starts from the innermost element (where it was triggered) and bubbles up through its ancestors in the DOM hierarchy.

Example of Event Bubbling:

```
<!DOCTYPE html>

 <html>

 <head>

 <title>Event Bubbling</title>

</head>

 <body>

 <div id="parent">

 <button id="child">Click Me!</button>

 </div>

<script> // Event listener on parent
document.getElementById('parent').addEventListener('click', function() {
alert('Parent clicked!'); }); // Event listener on child
document.getElementById('child').addEventListener('click', function() {
alert('Child clicked!'); });

 </script>

 </body>

 </html>
```

When you click the "Click Me!" button, you will see two alerts:

1. "Child clicked!" (the event on the button itself)
2. "Parent clicked!" (the event on the parent div because of bubbling)

3. Event Capturing

Event capturing (also called trickling) is the opposite of bubbling. The event starts at the root of the DOM and travels down to the target element. By

default, event listeners are added in the bubbling phase, but you can opt for capturing by passing **true** as the third argument to **addEventListener**.

Example of Event Capturing:

```
<!DOCTYPE html>

<html>

<head>

    <title>Event Capturing</title>

</head>

<body>

    <div id="parent">

        <button id="child">Click Me!</button>

    </div>


    <script>

        // Capturing event listener on parent (note the `true` for capture)

        document.getElementById('parent').addEventListener('click', function()
{

            alert('Parent clicked during capturing phase!');

        }, true);


        // Event listener on child

        document.getElementById('child').addEventListener('click', function()
{

            alert('Child clicked!');

        });
```

```
    </script>

</body>

</html>
```

In this example, if you click the button, the alert from the parent will appear first because the event is being captured from the root.

EVENT DELAGATION:

Event Delegation in JavaScript

Event delegation is a technique used in JavaScript where you attach a single event listener to a parent element, rather than attaching event listeners to each individual child element. This is particularly useful when you're working with dynamically added or removed elements. It leverages the event propagation (bubbling) model to capture events on child elements through their parent.

Why Use Event Delegation?

1. **Efficiency**: If you have many child elements, instead of attaching an event listener to each one, you can attach one listener to a parent element. This reduces the number of event listeners in your code, which can lead to better performance.
2. **Dynamic Content**: If child elements are dynamically added or removed from the DOM, event delegation allows you to handle events on those elements without needing to reattach listeners every time the DOM changes.

Example of Event Delegation:

Consider the following example where you have a list of items and want to handle a click event for each list item:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Event Delegation Example</title>
```

```
</head>

<body>

    <ul id="item-list">

        <li>Item 1</li>

        <li>Item 2</li>

        <li>Item 3</li>

        <li>Item 4</li>

    </ul>


    <script>

        // Attach event listener to the parent (ul)

        document.getElementById('item-list').addEventListener('click',
function(event) {

            // Check if the clicked target is an li

            if (event.target && event.target.nodeName === "LI") {

                alert('You clicked on ' + event.target.textContent);

            }

        });

    </script>

</body>

</html>
```

In this example:

- Instead of attaching a click event listener to each <li>, we attach a single listener to the parent <ul>.

- The event bubbles up from the clicked <li> to the <ul>, and we check if the event.target is an <li> element to determine which item was clicked.

DAY-12 (callbacks , callback hell , promise , higher order function)

Callbacks in JavaScript

A callback is a function that is passed as an argument to another function and is executed at a later time, usually after some asynchronous operation is complete. This is a common pattern in JavaScript for handling asynchronous tasks such as reading files, fetching data from a server, or waiting for a user action.

Example of a Callback:

```
// Simple function that accepts a callback

function fetchData(callback) {

  setTimeout(() => {

    const data = "Here is some data";

    callback(data); // Passing data to the callback function

  }, 2000);

}


// Callback function

function processData(data) {

  console.log("Processed data:", data);

}
```

```
// Calling the function and passing the callback

fetchData(processData);
```

How it Works:

- fetchData() is an asynchronous function that takes a callback (processData).
- After a delay (simulated with setTimeout), the fetchData function calls the callback and passes the result ("Here is some data") to it.
- processData() is the callback function that processes and logs the result.

Callback Hell (Pyramid of Doom)

Callback hell refers to the situation where you have many nested callbacks, making the code difficult to read, maintain, and debug. This often happens in asynchronous operations when you need to perform multiple tasks sequentially, and each task relies on the completion of the previous one.

The problem occurs when callbacks are nested inside one another, creating a "pyramid" structure that can quickly become hard to manage. Here's an example of callback hell:

Example of Callback Hell:

```
// Simulating a series of asynchronous operations

function fetchData(callback) {

  setTimeout(() => {

    console.log("Data fetched");

    callback();

  }, 1000);

}


function processData(callback) {
```

```
  setTimeout(() => {
    console.log("Data processed");
    callback();
  }, 1000);
}

function saveData(callback) {
  setTimeout(() => {
    console.log("Data saved");
    callback();
  }, 1000);
}

fetchData(() => {
  processData(() => {
    saveData(() => {
      console.log("All tasks completed!");
    });
  });
});
```

Problems with Callback Hell:

1. Nested Structure: As shown in the example, each asynchronous function depends on the completion of the previous one, leading to deep nesting. The more tasks you have, the deeper the nesting gets, creating hard-to-read code.

2. Difficulty in Error Handling: Handling errors becomes more complicated because each callback has its own error-handling logic. If something goes wrong, it might not be clear where the error occurred.

3. Poor Readability and Maintenance: As the nesting grows, the code becomes harder to follow and modify. Maintaining such code can be error-prone and challenging.

Solutions to Callback Hell

Over time, several solutions and patterns have been developed to avoid callback hell and make asynchronous code more readable and manageable.

1. Using Promises

Promises were introduced in JavaScript to handle asynchronous operations in a more readable and manageable way. Promises represent a value that may be available now, or in the future, or never.

Promises allow chaining of asynchronous tasks, avoiding deep nesting.

Example with Promises:

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Data fetched");
      resolve();
    }, 1000);
  });
}


function processData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      console.log("Data processed");
      resolve();
    }, 1000);
  });
}

function saveData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Data saved");
      resolve();
    }, 1000);
  });
}

fetchData()
  .then(processData)
  .then(saveData)
  .then(() => {
    console.log("All tasks completed!");
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Advantages of Promises:

- Better Readability: Promises allow chaining, reducing the deep nesting and making the code linear.
- Error Handling: Promises allow .catch() to handle errors centrally, rather than handling errors in each callback.
- Return Values: Promises can return values, making it easier to pass data between steps.

2. Using async/await (ES8)

The async/await syntax, introduced in ES8, makes asynchronous code look and behave more like synchronous code, making it more readable and easier to manage. It is built on top of Promises.

Example with async/await:

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Data fetched");
      resolve();
    }, 1000);
  });
}


function processData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Data processed");
      resolve();
```

```
    }, 1000);
  });
}

function saveData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Data saved");
      resolve();
    }, 1000);
  });
}

async function executeTasks() {
  await fetchData();
  await processData();
  await saveData();
  console.log("All tasks completed!");
}

executeTasks();
```

Advantages of async/await:

- Synchronous Style: The code appears synchronous, making it easier to follow and understand.

- Error Handling: You can use try/catch to handle errors in an elegant way.

- Better Debugging: It's easier to trace and debug asynchronous code written with async/await.

Conclusion

- Callbacks are useful for handling asynchronous operations, but when used excessively or nested deeply, they lead to callback hell, making the code difficult to manage.

- To avoid callback hell, you can use:

o Promises for better chaining and error handling.

o async/await to write asynchronous code in a more synchronous-like manner, improving readability and ease of maintenance

HIGHER ORDER FUNCTION

A higher-order function in JavaScript is a function that either:

1. Takes one or more functions as arguments, or

2. Returns a function as its result.

Higher-order functions are powerful tools that allow you to write more abstract and reusable code. JavaScript has many built-in higher-order functions like map(), filter(), reduce(), but you can also create your own.

Example 1: Higher-Order Function that Takes a Function as an Argument

```
// Higher-order function

function greet(name, callback) {

  console.log("Hello, " + name + "!");

  callback();
```

```
}


// A callback function

function farewell() {

  console.log("Goodbye!");

}

// Calling the higher-order function

greet("Alice", farewell);
```

Explanation:

- greet is a higher-order function because it accepts a function (farewell) as an argument.
- The callback parameter inside greet represents the farewell function passed when the greet function is called.
- The greet function prints a greeting message and then invokes the farewell function.

Output:

```
Hello, Alice!

Goodbye!
```

Key Points about Higher-Order Functions:

1. Function as Arguments: A higher-order function can accept other functions as arguments, allowing you to pass in behavior and customize the function's execution.
2. Function as Return Value: A higher-order function can return a function, enabling powerful patterns like closures and currying.
3. Flexibility and Reusability: Higher-order functions enable abstraction and can make your code more modular, flexible, and reusable.

Real-World Example: Using map() (Built-in Higher-Order Function)

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


// map is a higher-order function that takes a function as an argument
const doubledNumbers = numbers.map(function(num) {
  return num * 2;
});


console.log(doubledNumbers);  // [2, 4, 6, 8, 10]
```

In this case, the map() function takes a function (function(num) { return num * 2; }) and applies it to each element in the array, returning a new array with the results.

DAY-13 (Hoisting and Preventdefault)

HOISTING IN JAVASCRIPT

Hoisting in JavaScript

Hoisting is a JavaScript behavior where variable and function declarations are moved (or hoisted) to the top of their containing scope during the compilation phase, before the code has been executed. This means that you can use variables and functions before they are declared in the code.

However, it's important to note that only the declarations are hoisted, not the assignments. The values assigned to variables are not hoisted.

---

1. Hoisting with Variables

In JavaScript, variables can be declared using var, let, or const, and hoisting behaves differently for each of them.

Using var (Function-scoped or Globally-scoped)

Variables declared with var are hoisted, but only the declaration (not the initialization). If you try to access the variable before its declaration, it will have a value of undefined.

```
console.log(myVar); // undefined

var myVar = 5;

console.log(myVar); // 5
```

Explanation:

- **The declaration (var myVar;) is hoisted to the top of the scope.**
- **However, the assignment (myVar = 5;) is not hoisted, so the first console.log prints undefined.**

Using let and const (Block-scoped)

Variables declared with let and const are also hoisted, but they are placed in a "temporal dead zone" from the start of the block until the declaration is encountered. This means you cannot access them before the declaration without getting a ReferenceError.

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

let myLet = 10;

console.log(myLet); // 10
```

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

const myConst = 20;

console.log(myConst); // 20
```

Explanation:

- **For both let and const, the variable is hoisted, but accessing them before the actual declaration causes a ReferenceError because they are in the temporal dead zone.**
- **Once the declaration is encountered, they can be accessed.**

2. Hoisting with Functions

Function Declarations

In the case of function declarations, the entire function is hoisted, meaning you can call a function before its declaration in the code.

```
greet(); // "Hello, World!"


function greet() {
  console.log("Hello, World!");
}
```

Explanation:

- **The entire function declaration (function greet() { ... }) is hoisted to the top, so calling the function before the declaration works as expected.**

Function Expressions (with var, let, or const)

For function expressions, the function is treated as a variable, and only the variable declaration is hoisted, not the function assignment. This results in undefined when the function is called before its initialization.

```
sayHello(); // TypeError: sayHello is not a function
var sayHello = function() {
  console.log("Hello!");
};
```

Explanation:

- **The var sayHello; declaration is hoisted to the top, but the assignment (sayHello = function() { ... };) is not. As a result, sayHello is undefined when called before the initialization.**

For let or const:

```
sayHello(); // ReferenceError: Cannot access 'sayHello' before initialization
let sayHello = function() {
  console.log("Hello!");
};
```

Explanation:

- **let and const function expressions behave similarly to other let/const variables, causing a ReferenceError if accessed before the declaration.**

Summary of Hoisting Behavior

Declaration Type	Hoisting Behavior	Can Access Before Declaration?
var	Declaration is hoisted, initialization is not.	Yes, but value is undefined before initialization.
let and const	Hoisted to the top but placed in the temporal dead zone.	No, accessing before declaration throws a ReferenceError.
Function Declarations	Entire function (declaration + body) is hoisted.	Yes, you can call it before its declaration.
Function Expressions	Only the variable declaration is hoisted. Initialization is not hoisted.	No, accessing before initialization throws an error.

Important Notes

- **Hoisting is not magic: It's just how JavaScript handles the order of execution during the compilation phase. Variables and functions are processed before the code is executed.**

- let and const are block-scoped, while var is function-scoped (or globally-scoped if declared outside a function).
- var can lead to unexpected behavior due to hoisting, so it's recommended to use let and const for block-scoping and more predictable behavior.

event.preventDefault() in JavaScript

The event.preventDefault() method is used in JavaScript to prevent the default behavior of an event from occurring. It's particularly useful when dealing with user interactions that would normally trigger some action (like submitting a form, navigating to a link, or triggering a browser's default behavior), but you want to intercept or override that action and perform your own logic instead.

When to Use event.preventDefault():

- Form submission: Prevent a form from being submitted when certain conditions aren't met (e.g., validating form fields).
- Anchor (<a>) tags: Prevent the browser from navigating to a link's destination when an anchor tag is clicked.
- Mouse events: Prevent browser's default actions (like selecting text or opening the context menu on right-click).
- Keyboard events: Prevent default key actions (like stopping the form submission when the "Enter" key is pressed).

Syntax:

event.preventDefault();

Where event is the event object passed to the event handler (either implicitly or explicitly).

Examples of Using event.preventDefault()

1. Preventing Form Submission

You may want to prevent the default form submission if certain conditions aren't met. For example, if a user submits a form without filling out required fields, you can stop the form from being submitted and show a custom validation message.

```
<form id="myForm">
  <input type="text" id="username" placeholder="Enter username" required>
  <button type="submit">Submit</button>
</form>


<script>
  const form = document.getElementById("myForm");

  form.addEventListener("submit", function (event) {
    const username = document.getElementById("username").value;

    if (!username) {
      // Prevent form submission if the username is empty
      alert("Username is required!");
      event.preventDefault();
    }
  });
</script>
```

In this example:

- If the user submits the form without filling in the "username" field, the form submission is prevented and a validation message is shown instead.

2. Preventing Link Navigation

You can use event.preventDefault() to prevent the browser from navigating to the URL specified in an anchor (<a>) tag. This is useful when you need to handle the link click with JavaScript, such as in single-page applications (SPAs).

```
<a href="https://www.example.com" id="myLink">Click me</a>


<script>
  const link = document.getElementById("myLink");


  link.addEventListener("click", function (event) {
    // Prevent the default behavior of navigating to the link
    event.preventDefault();
    console.log("Link clicked, but navigation prevented!");
    // Add custom logic here
  });
</script>
```

In this example:

- Clicking the link will prevent the browser from navigating to https://www.example.com. Instead, it logs a message to the console.

3. Preventing Default Keyboard Behavior

You can also use event.preventDefault() to prevent default keyboard actions, such as when the user presses the "Enter" key in a form and you don't want it to trigger form submission.

```
<form id="myForm">
  <input type="text" id="inputField" placeholder="Type here" />
</form>


<script>
  const inputField = document.getElementById("inputField");

  inputField.addEventListener("keydown", function (event) {
    // Prevent form submission when "Enter" is pressed
    if (event.key === "Enter") {
      event.preventDefault();
      alert("Enter key pressed, but default form submission is prevented!");
    }
  });
</script>
```

In this example:

- Pressing the "Enter" key inside the input field will prevent the form from being submitted, and the custom alert message will be shown instead.

4. Preventing Right-Click Menu

You can prevent the context menu (right-click menu) from appearing by using event.preventDefault(). This is commonly used in custom UI elements where you want to override the default right-click behavior.

```
<div id="noRightClick">Right-click on this text is disabled</div>

<script>

const noRightClick = document.getElementById("noRightClick");

  noRightClick.addEventListener("contextmenu", function (event) {

    // Prevent the context menu from appearing on right-click

    event.preventDefault();

    alert("Right-click is disabled on this element.");

  });

</script>
```

In this example:

- **Right-clicking on the element with the ID noRightClick will prevent the default context menu from appearing and show a custom alert instead.**

Key Points:

- **Prevent Default Behavior: event.preventDefault() is used to stop the browser from performing its default action associated with an event.**
- **Common Use Cases: It's used for form validation, link navigation, keyboard handling, right-click menus, and more.**
- **When to Use: Use it when you need to override the default browser behavior and add custom functionality.**

Summary:

- **event.preventDefault() is a method used to prevent the default action associated with an event from occurring.**
- **It's widely used in handling forms, links, keyboard inputs, and mouse events.**
- **By using this method, you can control the flow of an event and define custom behavior in place of the default action.**

DAY-14 (Promise and Observable)

Here is a clear comparison between Promise and Observable in JavaScript.

Feature	Promise	Observable
Definition	Represents a single asynchronous operation that eventually resolves with a value or rejects with an error.	Represents a stream of asynchronous events or multiple values over time.
Multiple Values	Handles only a single value or error (resolves once).	Can emit multiple values (over time) and can continue to emit until unsubscribed.
Eager vs Lazy	Eager: Executes immediately when created.	Lazy: Starts executing only when subscribed to.
Execution	Executes immediately after creation.	Execution happens when a subscriber subscribes to it.
Cancellation	Cannot be cancelled once started.	Can be cancelled by unsubscribing from the Observable.
Error Handling	Handles errors with `.catch()` or `.then()` rejection handler.	Errors are handled inside the stream using `.catchError()` or within subscription.

Chaining	**Supports chaining through `.then()` for success and `.catch()` for errors.**	**Supports chaining with RxJS operators like `map()`, `filter()`, `mergeMap()`, etc.**
State Change	**Resolves once with a value or error and then remains static.**	**Can emit multiple values and dynamically change state over time.**
Return Type	**Returns a single value (resolved or rejected).**	**Returns an Observable that can emit multiple values over time.**
Subscription Management	**No subscription management needed (once resolved, the promise is done).**	**Needs manual subscription and unsubscription to avoid memory leaks.**
Use Case	**Ideal for handling a single asynchronous operation, such as HTTP requests, file reading.**	**Ideal for handling multiple events or streams, like WebSockets, user input, or continuous data streams.**
Memory Management	**Automatically resolves or rejects once.**	**Requires careful subscription management and proper unsubscription.**
Operators	**Limited chaining with `.then()`, `.catch()`.**	**Rich set of operators (e.g., `map()`, `filter()`, `mergeMap()`, `concat()`, etc.) available via RxJS.**
Completion	**A Promise either resolves or rejects and cannot complete afterward.**	**An Observable can complete with `complete()` after emitting all values.**
Stateful	**Once a Promise is settled (resolved or rejected), it cannot change.**	**Observables can change state dynamically and emit new values at any time.**

Example Code Comparison

Promise Example:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data loaded");
  }, 2000);
});

promise.then(result => {
  console.log(result);  // "Data loaded"
}).catch(error => {
  console.log(error);
});
```

Observable Example:

```
import { Observable } from 'rxjs';
```

```
let observable = new Observable(subscriber => {
  setTimeout(() => {
    subscriber.next("First value");
  }, 1000);

  setTimeout(() => {
    subscriber.next("Second value");
  }, 2000);

  setTimeout(() => {
    subscriber.complete();
  }, 3000);
});

observable.subscribe({
  next(value) { console.log(value); },
  }
});
```

Key Differences:

- **Promise: Resolves once with a single value and cannot emit further values.**
- **Observable: Can emit multiple values over time and be dynamically controlled via subscription and unsubscription.**

DAY-15 (Event Loop)

The event loop is a fundamental concept in JavaScript, especially in the context of asynchronous programming. It allows JavaScript to perform non-blocking operations, such as handling user input, processing events, or making network requests, without freezing the entire program. Here's how it works:

Overview:

JavaScript is single-threaded, meaning it executes one command at a time in a sequence. However, for tasks like handling user interactions, processing input, or managing timers, JavaScript needs to handle multiple events efficiently without blocking the main thread.

The event loop is the mechanism that allows JavaScript to do this by using call stacks, event queues, and callback functions. It continually checks if the call stack is empty and, if so, moves tasks from the event queue to the call stack for execution.

Components:

1. **Call Stack:**
    - **This is where the currently executing functions are placed. It's a stack data structure (like a stack of plates) that follows Last In, First Out (LIFO) order.**
    - **When a function is called, it is added to the top of the stack. When the function finishes execution, it is removed from the stack.**
2. **Event Queue (Task Queue):**
    - **This is where the browser or Node.js places events (like user interactions or setTimeout events) that are ready to be executed.**
    - **Functions or callbacks that are triggered by events are placed in the event queue.**
3. **Event Loop:**
    - **The event loop is constantly checking if the call stack is empty. If it is, it will move tasks (callback functions) from the event queue to the call stack for execution.**
    - **If the call stack is not empty, it waits until the stack is cleared.**
4. **Web APIs (or Node.js APIs):**
    - **In the browser environment (or Node.js environment), these are functions provided by the browser (like `setTimeout`, `fetch`, etc.) that allow**

asynchronous operations to be carried out. These APIs handle things like timers, DOM events, and network requests.

- When an asynchronous operation is triggered (for example, a `setTimeout`), the associated callback is sent to the Web APIs. Once the operation is done, the callback is placed in the event queue to be executed later.

Event Loop Cycle:

1. Execution starts: When the program starts, JavaScript begins executing the code in the call stack.
2. Asynchronous Task: If an asynchronous task like `setTimeout` or a fetch request is triggered, it's handed over to the Web APIs.
3. Event Queue: Once the asynchronous task completes, its callback is placed in the event queue.
4. Event Loop: The event loop checks the call stack. If the stack is empty, it moves the callback from the event queue to the call stack to be executed.

Example:

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


setTimeout(function() {
      console.log("Inside setTimeout");
}, 0);


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

Execution Steps:

1. The code execution begins with `console.log("Start")`, and it's immediately logged to the console.
2. The `setTimeout` function is called with a callback that logs "Inside setTimeout". This is an asynchronous task, so the callback is handed off to the Web APIs.

3. `console.log("End")` is called next, and it's logged to the console.
4. After a brief moment (even though the timeout is set to 0), the callback from `setTimeout` is placed in the event queue.
5. The event loop checks the call stack (which is empty) and then moves the callback from the event queue to the call stack, where it's executed and logs "Inside setTimeout".

Output:

```
Start

End

Inside setTimeout
```

Why does the callback from `setTimeout` run last?

This happens because JavaScript is single-threaded, and the event loop will only process the callback once the current code execution is finished. Even though `setTimeout` was set to 0ms, the callback still has to wait until the call stack is empty.

Key Points:

- Synchronous operations block the execution of other code.
- Asynchronous operations (e.g., `setTimeout`, `fetch`) allow JavaScript to continue executing while waiting for external tasks to complete.
- The event loop ensures that asynchronous callbacks are executed only when the call stack is empty.

WEBSITES TO CHECK

1)https://www.jsv9000.app/

2)latent flip