Building Trust & Careers

JAVASCRIPT

BASICS OF JAVASCRIPT WITH ES6+

JAVASCRIPT

High-level, versatile language for web development.

Uses

- DOM Manipulation → Modify HTML/CSS dynamically
- Event Handling → Respond to clicks, key presses, etc.
- Asynchronous Communication → Fetch/send data (APIs, AJAX)
- Full-Stack → Client-side & server-side (Node.js)
- Cross-Platform → Web, mobile (React Native), desktop (Electron.js).

EXAMPLE:

```
document.getElementById("btn").addEventListener("click", async () => {
  const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
  const data = await response.json();
  console.log(data);
});
```

LINKING JAVASCRIPT FILES USING <SCRIPT>

Including an External JavaScript File

- To link an external JavaScript file, use the <script> tag with the src attribute.

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript Example</title>
</head>
<body>
    <h1>Welcome to JavaScript</h1>
    <script src="script.js"></script>
</body>
</html>
```

Benefits:

- Place <script> before </body> for better page load performance
- Use defer for scripts that depend on HTML content
- Use async for independent scripts that don't rely on DOM elements.

LOGGING WITH JAVASCRIPT

1.General Logging

- Used to display information in the console.

```
console.log("Hello, World!"); // Outputs general info
```

2.Informational Message

- Logs important information.

```
console.info("Info Message"); // Displays an informational log
```

3.Warning Message

- Displays warnings in the console.

```
console.warn("Warning Message"); // Highlights a potential issue
```

4.Error Logging

- Logs errors in the console.

```
console.error("Error Message"); // Shows an error message
```

LOGGING WITH JAVASCRIPT

5.User Input (Prompt)

- Asks for user input via a pop-up.

```
let name = prompt("Enter your name:"); // Gets input from the user
```

6.Alert Message

- Shows a message in an alert box.

```
alert("Hello!"); // Displays an alert pop-up
```

7.Confirmation Box

- Asks the user to confirm an action.

```
let response = confirm("Are you sure?"); // Returns true/false
```

VARIABLES AND KEYWORDS IN JAVASCRIPT (VAR, LET, CONST)

1.var

- Scope: Function-scoped
- Characteristics: Allows redeclaration and updating.

```
var name = "John"; // Redeclaration and updates allowed
```

2.let

- Scope: Block-scoped
- Characteristics: Allows updates, but not redeclaration in the same scope.

```
let age = 25; // Can be updated, but not redeclared in the same block
```

3.const

- Scope: Block-scoped
- Characteristics: Cannot be updated or redeclared.

```
const country = "India"; // Cannot be reassigned or redeclared
```

VARIABLE DECLARATION, INITIALIZATION, AND UPDATING

1.Declaration

- Declaring a variable without initializing it.

```
let x; // Declaration
```

2.Initialization

- Assigning a value to the variable at the time of declaration.

```
let y = 10; // Initialization
```

3.Updating

- Updating the value of a variable
- let and var allow updates, but const does not.

```
y = 20; // Updating
```

Example

```
let score;            //Declaration
score = 100;      //Initialization
score = 150;      //Updating
console.log(score);  //Output : 150
```

JAVASCRIPT STATEMENTS AND SEMICOLONS

Statements

- Statements are individual instructions in JavaScript, like variable declarations or function calls.

```
let a = 10;  // Declaration
let b = 20;  // Declaration
console.log(a + b);  // Function call
```

Semicolons

- Semicolons are optional in JavaScript, but they help prevent errors, especially when statements are on the same line.
- They are recommended to avoid potential issues with automatic semicolon insertion.

```
let a = 10;       // Declaration with semicolon
let b = 20;       // Declaration with semicolon
console.log(a + b);  // Function call with semicolon
```

ADDING COMMENTS IN JAVASCRIPT

1.Single-line Comment

- Use // to add a comment on a single line.

```
// This is a single-line comment
```

2.Multi-line Comment

- Use /* */ for comments spanning multiple lines.

```
/*
 This is a
 multi-line comment
*/
```

Example:

```
// This is a single-line comment
let x = 5;  // This comment is at the end of a line

/*
 This is a multi-line comment
 explaining the following code
*/
let y = 10;
```

EXPRESSIONS IN JAVASCRIPT AND THEIR DIFFERENCE FROM STATEMENTS

1.Expression

- An expression produces a value (e.g., arithmetic operations or variable assignments).

```
let sum = 5 + 10;  // Expression (produces the value 15)
```

2.Statement

- A statement performs an action (e.g., variable declaration, function call).

```
console.log(sum);  // Statement (performs an action)
```

Example:

```
let sum = 5 + 10;   // Expression (produces a value)
console.log(sum);    // Statement (executes an action)
```

JAVASCRIPT OPERATORS

Arithmetic Operators

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus – Remainder of division)

Example:

```
let num1 = 10;
let num2 = 5;

console.log(num1 + num2); // Output: 15 (Addition)
console.log(num1 % num2); // Output: 0 (Modulus)
```

INCREMENT & DECREMENT OPERATORS

Increment (++)

- Increases the value of a variable by 1.

```
count++; // Increment
```

Decrement (--)

- Decreases the value of a variable by 1.

```
count--; // Decrement
```

Example:

```
let count = 5;
count++;          // Increment
console.log(count); // Output: 6
```

PRIMITIVE DATA TYPES IN JAVASCRIPT

Primitive data types are immutable and stored directly in memory.

1.Number (Integer & Floating-point)

```
let num = 42;
let floatNum = 3.14;
```

2.String (Text enclosed in quotes)

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

3.Boolean (Represents true or false)

```
let isAvailable = true;
```

4.Null (Represents an empty or non-existent value)

```
let emptyValue = null;
```

5.Undefined (Declared but not assigned a value)

```
let notDefined;
```

6.Symbol (Unique and immutable value)

```
let sym = Symbol("unique");
```

7.BigInt (Handles large numbers beyond Number.MAX_SAFE_INTEGER)

```
let bigInt = 1234567890123456789012345678890n;
```

REFERENCE (RELATIVE) DATA TYPES IN JAVASCRIPT

Reference data types are stored in memory by reference and can be modified.

1.Object (Collection of key-value pairs)

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

2.Array (Ordered list of values)

```
let fruits = ["Apple", "Banana", "Cherry"];
```

JAVASCRIPT DATA TYPES

JavaScript has 8 data types, categorized into Primitive and Reference types.

Primitive Data Types:

- Number → let num = 10
- String → let text = "Hello"
- Boolean → let isActive = false
- Null → let data = null
- Undefined → let notDefined
- Symbol → let sym = Symbol("id");

Primitive Data Types:

- Array → let list = [1, 2, 3]
- Object → let obj = { key: "value" };

Example:

```
let num = 10; // number
let text = "Hello"; // string
let isActive = false; // boolean
let data = null; // null
let list = [1, 2, 3]; // array
let obj = { key: "value" }; // object
let sym = Symbol("id"); // symbol
let notDefined; // undefined
```

SOME IMPORTANT VALUES IN JAVASCRIPT

1. undefined (Variable declared but not assigned a value)
2. null (Intentional absence of value)
3. NaN ("Not-a-Number" - Invalid mathematical operations)
4. Infinity (Represents an infinite value)

Example:

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

let price = null; // null (No price assigned yet)
console.log(price); // null

let result = "hello" / 2; // NaN (Invalid operation)
console.log(result); // NaN

let infiniteNumber = 10 / 0; // Infinity
console.log(infiniteNumber); // Infinity
```

BASIC OPERATORS IN JAVASCRIPT

Arithmetic Operators (+,-,*,/,%,++,--)

Example:

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

console.log(a + b); // 15  (Addition)
console.log(a - b); // 5   (Subtraction)
console.log(a * b); // 50  (Multiplication)
console.log(a / b); // 2   (Division)
console.log(a % b); // 0   (Modulus - Remainder)

a++;
console.log(a); // 11  (Increment)

b--;
console.log(b); // 4   (Decrement)
```

Assignment Operators (=,+=,-=,*=,/=,%=)

Example:

```
let x = 10;

x += 5; // Equivalent to x = x + 5
console.log(x); // 15

x -= 3; // Equivalent to x = x - 3
console.log(x); // 12

x *= 2; // Equivalent to x = x * 2
console.log(x); // 24

x /= 4; // Equivalent to x = x / 4
console.log(x); // 6

x %= 5; // Equivalent to x = x % 5
console.log(x); // 1
```

BASIC OPERATORS IN JAVASCRIPT

Comparison Operators (==, ===, !=, !==, >, <, >=, <=)

Example:

```
console.log(5 == "5");  // true  (loose equality, type conversion happens)
console.log(5 === "5"); // false (strict equality, no type conversion)

console.log(10 != "10");  // false (loose inequality)
console.log(10 !== "10"); // true  (strict inequality)

console.log(8 > 5);  // true
console.log(8 < 5);  // false
console.log(8 >= 8); // true
console.log(8 <= 10); // true
```

Logical Operators (&&, ||, !)

Example:

```
console.log(true && false); // false (Both conditions must be true)
console.log(true || false); // true  (At least one condition must be true)
console.log(!true);         // false (Negates the value)
```

VARIABLE HOISTING IN JAVASCRIPT

Hoisting moves variable and function declarations to the top of their scope before execution.

Using var (Hoisted but Undefined)

Using let and const (Hoisted but Not Initialized)

Key Takeaway:

- var is hoisted with an initial value of undefined
- let and const are hoisted but remain in the Temporal Dead Zone until assigned.

CONDITION OPERATORS IN JAVASCRIPT

Hoisting moves variable and function declarations to the top of their scope before execution.

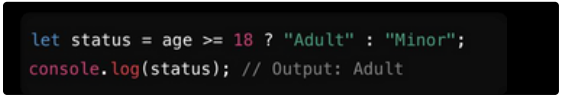
1.if-else Statement

Used for Basic Conditional Checks.

```
let age = 18;
if (age >= 18) {
  console.log("Adult");
} else {
  console.log("Minor");
}
```

2.Ternary Operator

A shorthand way of writing if-else statements



```
let status = age >= 18 ? "Adult" : "Minor";
console.log(status); // Output: Adult
```

Key Takeaway:

- if-else gives more flexibility for multiple conditions and is easy to read
- The ternary operator is great for simple conditions in a single line

LOOPS IN JAVASCRIPT

for Loop → Used when the number of iterations is known beforehand.

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

while Loop → Used when the loop should run as long as a condition is true.

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

do...while Loop → Runs the block of code once, then checks the condition.

```
let j = 0;
do {
  console.log(j);  // Output: 0, 1, 2, 3, 4
  j++;
} while (j < 5);
```

JAVASCRIPT MASTERY

1.Working with Strings in JavaScript

JavaScript provides various methods to manipulate and work with strings. Since strings are immutable, any modification results in a new string.

1.Slice()

- Extract a section of a string and returns a new string without modifying the original.
- Example:

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

2.Template Strings

- Uses backticks(`) to allow embedding expressions within a string.
- Example:

```
let name = "John";
console.log(`Hello, ${name}!`); // Output: "Hello, John!
```

3.Split()

- Splits a string into an array based on a seperator.
- Example:

```
let words = "Hello World".split(" ");
console.log(words); // Output: ["Hello", "World"]
```

JAVASCRIPT MASTERY

1.Working with Strings in JavaScript

4.Replace()

- Replaces a specified substring with another.
- Example:

```
let text = "Hello";
console.log(text.replace("H", "J")); // Output: "Jello"
```

5.includes()

- Checks if a substring exists within a string.
- Example:

```
console.log("JavaScript".includes("Java")); // Output: true
```

JAVASCRIPT MASTERY

2.Conditional Operators in JavaScript

1.if statement

- Excutes code if a condition is true.
- Example:

```
let x = 10;
if (x > 5) {
  console.log("Big number");
}
```

2.if-else statement

- Runs different code blocks based on a condition.
- Example:

```
let num = 4;
if (num > 5) {
  console.log("Greater than 5");
} else {
  console.log("Less than or equal to 5");
}
```

JAVASCRIPT MASTERY

2.Conditional Operators in JavaScript

3.else-if ladder

- Used for multiple conditions.
- Example:

```
let score = 85;
if (score >= 90) {
  console.log("A grade");
} else if (score >= 80) {
  console.log("B grade");
} else {
  console.log("Below B grade");
}
```

4.Ternary Operator

- Shorter way to write an if-else statement.
- Example:

```
let age = 18;
let status = age >= 18 ? "Adult" : "Minor";
console.log(status); // Output: "Adult"
```

JAVASCRIPT MASTERY

2.Conditional Operators in JavaScript

5.Switch statement

- Alternative to multiple if-else conditions.
- Example:

```
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("Start of the week");
    break;
  case "Friday":
    console.log("Weekend is coming!");
    break;
  default:
    console.log("Normal day");
}
```

JAVASCRIPT MASTERY

3.Loops in JavaScript

1.for loop

- Repeats a block of code a specific number of times.
- Example:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

2.while loop

- Executes a block while a condition is true.
- Example:

```
let x = 0;
while (x < 5) {
  console.log(x);
  x++;
}
```

3.do...while loop

- Executes a block at least once before checking the condition.
- Example:

```
let y = 0;
do {
  console.log(y);
  y++;
} while (y < 5);
```

JAVASCRIPT MASTERY

3.Loops in JavaScript

4.forEach loop(for arrays)

- Iterates over each element in an array.
- Example:

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

5.for...in loop(for objects)

- Iterates over object properties.
- Example:

```
let obj = { a: 1, b: 2 };
for (let key in obj) {
  console.log(key, obj[key]);
}
```

6.for...of loop(for iterables)

- Iterates over iterable objects like arrays and strings.
- Example:

```
let arr = [10, 20, 30];
for (let value of arr) {
  console.log(value);
}
```

JAVASCRIPT MASTERY

4.Functions in JavaScript

1.Regular Function

- Example:

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

2.Arrow Function

- Shorter syntax for defining functions.
- Example:

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

3.Immediatly Invoked Function Expression(IIFE)

- A function that runs immediately after being defined.
- Example:

```
(function() {
  console.log("This runs immediately!");
})();
```

JAVASCRIPT MASTERY

4.Functions in JavaScript

4.Higher-Order Function

- A function that takes another function as an argument.
- Example:

```
function operate(fn, a, b) {
  return fn(a, b);
}
console.log(operate((x, y) => x + y, 10, 20));
// Output: 30
```

JAVASCRIPT MASTERY

5.Scoping & Closures in JavaScript

1.Global Scope

- Variables declared outside any function are accessible anywhere.
- Example:

```
let globalVar = "Hello";
function show() {
  console.log(globalVar);
}
```

2.Function Scope

- Variables declared inside a function are not accessible outside.
- Example:

```
function example() {
  let localVar = 10;
}
console.log(localVar); // Error
```

3.closures

- A function that remember variables from its outer functions.
- Example:

```
function outer(x) {
  return function inner(y) {
    return x + y;
  };
}
const addFive = outer(5);
console.log(addFive(10)); // Output: 15
```

LOOPS IN JAVASCRIPT

Loops in Javascript

Loops allow us to execute a block of code multiple times based on a condition. JavaScript supports different types of loops, such as for, while, and for...of

1. Printing Numbers in Reverse (for loop)

Problem: Write a for loop to print numbers from 10 to 1 in reverse.

```
for (let i = 10; i >= 1; i--) {
   console.log(i);
}
```

Explanation:

- The loop starts with i = 10.
- The condition i >= 1 ensures the loop continues until i reaches 1.
- i-- decreases the value of i by 1 in each iteration.

LOOPS IN JAVASCRIPT

Loops in Javascript

2. Multiples of 3 (while loop)

Problem: Use a while loop to print multiples of 3 from 3 to 30.

Solution:

```
let num = 3;
while (num <= 30) {
  console.log(num);
  num += 3;
}
```

Explanation:

- The while loop runs as long as num is less than or equal to 30.
- num starts at 3 and increases by 3 each time, ensuring only multiples of 3 are printed.

LOOPS IN JAVASCRIPT

Loops in Javascript

3. Sum of Numbers from 1 to 100

Problem: Write a program to calculate the sum of numbers from 1 to 100 using a loop.

Solution:

```
let sum = 0;
for (let i = 1; i <= 100; i++) {
   sum += i;
}
console.log("Sum:", sum); // Output: 5050
```

Explanation:

- The loop runs from 1 to 100.
- In each iteration, the current value of i is added to sum.
- The final sum is 5050.

LOOPS IN JAVASCRIPT

Loops in Javascript

4. Star Pattern (Nested Loops)

Problem: Create a nested loop to print a star pattern.

Solution:

```
for (let i = 1; i <= 5; i++) {
    let stars = "";
    for (let j = 1; j <= i; j++) {
        stars += "*";
    }
    console.log(stars);
}
```

Output:

```
*
**
***
****
*****
```

Explanation:

- The outer loop controls the number of rows (1 to 5).
- The inner loop adds * for each row.

LOOPS IN JAVASCRIPT

Loops in Javascript

5. Iterating Over a String (for...of loop)

Problem: Use a for...of loop to iterate over the string "JavaScript".

Solution:

```
let str = "JavaScript";
for (let char of str) {
   console.log(char);
}
```

Explanation:

- The for...of loop iterates over each character in the string and prints it.

ARRAYS

Arrays in javascript

An array is a collection of values stored in a single variable.

JavaScript provides multiple methods to manipulate arrays.

1.Removing Duplicate Values from an Array

Problem: Remove duplicate values from an array.

Solution:

```
let arr = [1, 2, 3, 2, 4, 3, 5];
let uniqueArr = [...new Set(arr)];
console.log(uniqueArr); // [1, 2, 3, 4, 5]
```

Explanation:

- Set stores only unique values.
- The spread operator ... converts the Set back into an array.

ARRAYS

Arrays in javascript

2.Finding the Second Largest Number in an Array

Problem: Find the second largest number in an array.

Solution:

```
function secondLargest(arr) {
    let sorted = [...new Set(arr)].sort((a, b) => b - a);
    return sorted.length > 1 ? sorted[1] : null;
}
console.log(secondLargest([10, 20, 5, 30, 30])); // Output: 20
```

Explanation:

- new Set(arr) removes duplicates.
- .sort((a, b) => b - a) sorts the array in descending order.
- The second element is returned if available.

ARRAYS

Arrays in javascript

3.Sorting an Array in Descending Order

Problem: Sort an array in descending order.

Solution:

```
let numbers = [5, 2, 9, 1, 5, 6];
numbers.sort((a, b) => b - a);
console.log(numbers); // [9, 6, 5, 5, 2, 1]
```

Explanation:

- .sort((a, b) => b - a) sorts the array from largest to smallest.

ARRAYS

Arrays in javascript

4.Reversing an Array Without reverse()

Problem: Reverse an array without using .reverse().

Solution:

```
function reverseArray(arr) {
    let reversed = [];
    for (let i = arr.length - 1; i >= 0; i--) {
        reversed.push(arr[i]);
    }
    return reversed;
}
console.log(reverseArray([1, 2, 3, 4])); // [4, 3, 2, 1]
```

Explanation:

- A new array reversed is created.
- The loop starts from the last index and pushes each element into reversed.

ARRAYS

Arrays in javascript

5.Finding the Most Frequent Element in an Array

Problem: Find the most frequent element in an array.

Solution:

```
function mostFrequent(arr) {
    let freqMap = {};
    let maxFreq = 0, mostFrequentNum = null;

    for (let num of arr) {
        freqMap[num] = (freqMap[num] || 0) + 1;
        if (freqMap[num] > maxFreq) {
            maxFreq = freqMap[num];
            mostFrequentNum = num;
        }
    }
    return mostFrequentNum;
}
console.log(mostFrequent([1, 3, 3, 2, 3, 2, 2, 2, 2]));
// Output: 2
```

Explanation:

- A freqMap object stores the frequency of each number.
- The loop updates the highest frequency found and stores the most frequent number.

ARRAYS & OBJECTS IN JAVASCRIPT

Array Methods

- **map()** → Creates a new array with transformed values.
- **filter()** → Returns elements matching a condition.
- **forEach()** → Runs a function for each element.
- **reduce()** → Reduces an array to a single value by applying a function on each element.

Example 1:

```
let nums = [1, 2, 3];
let squares = nums.map(x => x * x);
console.log(squares); // [1, 4, 9]
```

Example 2:

```
let nums = [10, 25, 30, 5];
let bigNums = nums.filter(x => x > 20);
console.log(bigNums); // [25, 30]
```

Example 3:

```
let fruits = ["apple", "banana", "mango"];
fruits.forEach(f => console.log(f.toUpperCase()));
// APPLE, BANANA, MANGO
```

ARRAYS & OBJECTS IN JAVASCRIPT

Example 4:

```
let nums = [1, 2, 3, 4];
let sum = nums.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10

let numbers = [2, 3, 4];
let product = numbers.reduce((acc, curr) => acc * curr, 1);
console.log(product); // 24
```

Array Iteration

Iterating means going through each element of an array to read or process it.

Methods:

- for loop: Classic, uses index
- forEach() method: Runs a function for each element
- map() method: Returns a new array with transformed values
- filter() method: Returns a subset of elements matching a condition

ARRAYS & OBJECTS IN JAVASCRIPT

Example:

```
let fruits = ["apple", "banana", "mango"];
fruits.forEach(fruit => console.log(fruit));
// Output: apple, banana, mango
let lengths = fruits.map(fruit => fruit.length);
console.log(lengths); // [5, 6, 5]
```

Objects in JavaScript

An Object is a collection of key–value pairs used to store related data. Each key is unique, and its value can be any type, including numbers, strings, arrays, or functions.

Essential Concepts:

- Stored in { key: value } format
- Keys are usually strings; values can be numbers, strings, arrays, functions, or other objects
- Access properties using:
  - Dot notation: obj.key
  - Bracket notation: obj["key"] (useful for spaces or dynamic keys)

Objects can contain methods (functions inside objects)

ARRAYS & OBJECTS IN JAVASCRIPT

Example:

```
let student = { name: "Anjali", age: 21 };
console.log(student.name); // Anjali
student.age = 22;
console.log(student.age); // 22
let student2 = { "full name": "Anjali Sharma", age: 21 };
console.log(student2["full name"]); // Anjali Sharma
let key = "age";
console.log(student2[key]);    // 21
```

Timing Events

Timing events let JavaScript execute code after a delay or repeatedly at intervals, which is useful for animations, notifications, or periodic tasks. They help control when code runs instead of immediately.

ARRAYS & OBJECTS IN JAVASCRIPT

Example:

```
setTimeout(() => console.log("Hello after 2 seconds"), 2000);
let count = 0;
let intervalId = setInterval(() => {
  console.log("Count:", count);
  count++;
  if(count > 3) clearInterval(intervalId);
}, 1000);
```

Event Handling in JavaScript

Events are actions that happen in the browser (clicks, typing, hover) that JavaScript can respond to using functions. Event handling makes web pages interactive and dynamic.

Example:

```
let btn = document.querySelector("button");
btn.addEventListener("click", () => console.log("Button
clicked!"));
```

EVENT HANDLING IN JAVASCRIPT

Scroll, Mouse & Key Events

- **Scroll Events:** Triggered when the user scrolls the page. Useful for lazy loading or animations.
- **Mouse Events:** Triggered by mouse actions like click, hover, or double-click. Used for interactive UI elements.
- **Key Events:** Triggered by keyboard actions. Helps capture user input or shortcuts.

Example:

```
window.addEventListener("scroll", () =>
console.log("Scrolling..."));
document.addEventListener("keydown", (e) =>
console.log(e.key + " pressed"));
```

EVENT HANDLING IN JAVASCRIPT

Form Handling

JavaScript can validate, read, and process form data before submitting it. This improves user experience by preventing incorrect input and controlling data submission.

Example:

```
let form = document.querySelector("form");
form.addEventListener("submit", (e) => {
  e.preventDefault();
  let name = document.querySelector("#name").value;
  console.log("Name:", name);
});
```

EVENT HANDLING IN JAVASCRIPT

Browser Events

Browser events occur on the window or document and allow dynamic reactions to actions like page load, resize, or closing. They help in adapting UI or triggering scripts.

Example:

```
window.addEventListener("load", () => console.log("Page loaded"));
window.addEventListener("resize", () => console.log("Window resized"));
```

ERROR HANDLING IN JAVASCRIPT

Error handling prevents programs from breaking due to unexpected issues. JavaScript provides tools to catch, handle, and create custom errors, making code more reliable.

Types of Errors:

- **Syntax Errors:** Mistakes in code structure
- **Runtime Errors:** Errors while code is running
- **Logical Errors:** Code runs but produces wrong results

Using try-catch:

```
try {
  riskyFunction();
} catch (error) {
  console.log("Error:", error.message);
}
```

Creating Custom Errors:

```
let age = -5;
if(age < 0) throw new Error("Age cannot be negative");
```

Handling Async Errors:

```
async function fetchData() {
  try {
    await fetch("invalid-url");
  } catch (err) {
    console.log("Error fetching data:", err.message);
  }
}
fetchData();
```

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

Understanding Higher-Order Functions (HOFs)

A Higher-Order Function (HOF) is a function that either takes another function as an argument or returns a function. These are widely used in JavaScript to enhance code reusability and maintainability.

1. Delayed Execution Using Callback (HOF + Callback)

Concept:

A function can accept another function as an argument (callback) and execute it after a delay using setTimeout.

Implementation

```
function delayedExecution(callback) {
    setTimeout(callback, 3000);
    // Calls the callback function after 3 seconds
}

// Example usage
delayedExecution(() =>
console.log("Executed after 3 seconds"));
```

Explanation:

- delayedExecution accepts a function callback.
- setTimeout is used to delay the execution of the callback for 3 seconds.
- When called, it logs a message to the console after 3 seconds.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

2. Implementing Custom Function (HOF)

Concept:

The .map() method is used to transform an array by applying a callback function to each element. We can create our own version of .map().

Implementation

```
function customMap(array, callback) {
    let result = [];
    for (let i = 0; i < array.length; i++) {
        result.push(callback(array[i], i, array));
        // Apply callback to each element
    }
    return result;
}

// Example usage
console.log(customMap([1, 2, 3], num => num * 2));
// Output: [2, 4, 6]
```

Explanation:

- customMap takes an array and a callback function.
- Iterates over the array, applies the callback function to each element, and stores the result.
- Works similarly to Array.prototype.map but is implemented manually.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

3. Closures: Creating a Counter Function

Concept:

A closure is a function that retains access to variables from its outer scope even after the outer function has finished execution.

Implementation

```
function createCounter() {
    let count = 0;
    return function() { // Closure retains access to `count`
        return ++count;
    };
}

// Example usage
const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

Explanation:

- createCounter defines a variable count and returns a function.
- The inner function forms a closure, keeping count in memory.
- Each time the inner function is called, count is incremented and returned.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

4. Limiting Function Calls (Closure + HOF)

Concept:

A function should only be executed a limited number of times. This can be achieved using closures.

Implementation

```
function createCounter() {
    let count = 0;
    return function() { // Closure retains access to `count`
        return ++count;
    };
}

// Example usage
const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

Explanation:

- limit takes a function fn and a limit value.
- It tracks the number of times the function has been called.
- Once the limit is reached, further calls do nothing

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

Conclusion

- Higher-Order Functions (HOFs) enable cleaner and reusable code by accepting functions as arguments.
- Callbacks allow asynchronous behavior and function execution control.
- Closures help retain variable states and create private data.
- These concepts are crucial in modern JavaScript development, especially in functional programming and asynchronous operations.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

1. Repeating a Function at Intervals (Using Callbacks)

Concept:

A callback function is a function passed as an argument to another function. Using setInterval, we can execute a callback function repeatedly at specified intervals.

Implementation

```
function repeatFunction(callback, interval) {
    setInterval(callback, interval * 1000);
}

// Example usage
repeatFunction(() => console.log("Repeating..."), 2);
// Logs "Repeating..." every 2 seconds
```

Explanation:

- The function repeatFunction takes two parameters:
  - callback: The function to execute
  - interval: The time in seconds between executions
- setInterval is used to call callback repeatedly after every interval seconds.
- In the example, the function logs "Repeating..." every 2 seconds.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

2. Creating a Function with a Preset Greeting (Using Closures)

Concept:

A closure allows a function to "remember" variables from its outer scope even after the outer function has finished executing.

Implementation

```
function greetUser(greeting) {
    return function (name) {
        return `${greeting}, ${name}!`;
    };
}

// Example usage
const greetHello = greetUser("Hello");
console.log(greetHello("Alice")); // "Hello, Alice!"
console.log(greetHello("Bob"));   // "Hello, Bob!"
```

Explanation:

- greetUser is a higher-order function that returns another function.
- The returned function remembers the greeting value from greetUser (closure property).
- When greetHello("Alice") is called, it uses the stored greeting ("Hello") and returns "Hello, Alice!".
- This technique is useful for creating pre-configured functions.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

3. Executing a Function Only Once (Using HOFs + Closures)

Concept:

A function should only be executed once, no matter how many times it is called.

Implementation

```
function once(fn) {
    let executed = false;
    return function (...args) {
        if (!executed) {
            executed = true;
            return fn(...args);
        }
    };
}

// Example usage
const init = once(() => console.log("Initialized!"));
init(); // "Initialized!"
init(); // (No output)
```

Explanation:

- The once function wraps another function (fn) and ensures it only executes once.
- The variable executed keeps track of whether the function has already been called.
- The first call executes fn, but all subsequent calls do nothing.
- Useful for initialization functions.

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

4. Throttling a Function (Using HOFs + Closures)

Concept:

Throttling ensures that a function is not executed more than once within a specified time period. This is useful in event listeners to improve performance.

Implementation

```
function throttle(fn, delay) {
    let lastCall = 0;
    return function (...args) {
        let now = Date.now();
        if (now - lastCall >= delay) {
            lastCall = now;
            fn(...args);
        }
    };
}

// Example usage
const throttledFn = throttle(() =>
console.log("Throttled Execution"), 2000);
throttledFn();
throttledFn();
throttledFn();
// Only executes the first call,
others are ignored until 2 sec passes
```

JAVASCRIPT ADVANCED HOF'S , CALLBACKS, AND CLOSURES

Explanation:

- throttle ensures that fn runs only once in every delay milliseconds.
- The lastCall variable stores the last execution time.
- If delay hasn't passed, additional calls are ignored.
- Useful in scenarios like scrolling events or resizing windows.