Introduction to JavaScript, Variables, Data Types, and Console Basics

What is JavaScript?

JavaScript (JS) is a versatile, high-level, and interpreted programming language primarily used to make web pages interactive. It allows you to create dynamic content, handle multimedia, animate images, and much more. While it's famous for web development (frontend and backend with Node.js), it's also used in mobile apps, desktop apps, and even game development.

Key Characteristics:

- Client-Side Scripting: Runs directly in the user's web browser, reducing server load.
- Interpreted: Code is executed line by line by the browser without a separate compilation step.
- Object-Oriented: Supports object-oriented programming paradigms, allowing for organized and reusable code.
- Dynamic and Flexible: Allows for a lot of flexibility in how you write code, including dynamic typing.
- Variables: Storing Information Think of variables as named containers that hold data. You give them a name, and then you can put different values into them. In JavaScript, you declare variables using var, let, or const.
- **var (Older way):**

Historically used, but has functional scope and can be re-declared and updated. This can lead to unexpected behavior and bugs in larger applications.

```
var myName = "Ali";
var myName = "Ahmed"; // Can be re-declared (reassigns the variable)
myName = "Kamran"; // Can be updated
console.log(myName); // Output: Kamran
```

- **let (Modern way - ES6+):**

Block-scoped. This means it's only accessible within the block of code (e.g., inside {}) where it's defined. It can be updated but not re-declared within the same scope. This is generally preferred when your variable's value might change.

```
let age = 25;
age = 26; // Can be updated
console.log(age); // Output: 26
// let age = 27; // Error: Cannot re-declare block-scoped variable 'age'
```

- **const (Modern way - ES6+):**

Also block-scoped. It cannot be re-declared or updated once assigned. Use const for values that should remain constant throughout your program. This prevents accidental reassignments and makes your code more predictable.

```
const PI = 3.14159;
// PI = 3.14; // Error: Assignment to constant variable.
console.log(PI); // Output: 3.14159
```

☑Best Practice: Always prefer const if the value won't change. If you know the value will need to be updated, then use let. Avoid var in new JavaScript code to prevent potential issues related to its scope.

Data Types: Types of Information

JavaScript supports several data types to categorize the kind of value a variable holds. Understanding these is crucial for performing correct operations. You can use the typeof operator to check the data type of a variable.

Primitive Data Types (represent a single, immutable value):

String:

Represents textual data. Enclosed in single (' '), double (" "), or backticks ( ). Backticks allow for template literals, which enable embedding expressions (${expression}) and multiline strings.

```
let greeting = "Hello, World!";
let userName = "Sara";
let message = `Today is a good day.`; // Using backticks
let fullName = `${userName} Khan`; // Template literal with variable
console.log(typeof greeting); // Output: string
```

Number:

Represents both integer and floating-point numbers. JavaScript handles all numbers as floating-point internally.

```
let score = 100; // Integer
let price = 19.99; // Floating-point
let temperature = -5;
console.log(typeof score); // Output: number
```

Boolean:

Represents a logical entity and can only have two values: true or false. Used for conditional logic.

```
let isLoggedIn = true;
let hasPermission = false;
console.log(typeof isLoggedIn); // Output: boolean
```

Undefined:

A variable that has been declared but not yet assigned a value is undefined. It means "no value has been assigned yet."

```
let phoneNumber; // phoneNumber is undefined
console.log(phoneNumber); // Output: undefined
console.log(typeof phoneNumber); // Output: undefined
```

Null:

Represents the intentional absence of any object value. It means "no value," explicitly set by the programmer.

```
let selectedUser = null; // No user selected yet (intentional absence)
console.log(selectedUser); // Output: null
console.log(typeof selectedUser); // Output: object (This is a historical bug in
JS, null is a primitive type)
```

Symbol (ES6+):

A unique and immutable primitive value. Used for unique object property keys to avoid naming collisions. (More advanced, we'll touch on this later if needed).

BigInt (ES2020):

Represents integers with arbitrary precision. Used for numbers larger than Number can safely handle (i.e., beyond 2^53 - 1). (More advanced, not commonly used in basic web dev).

Non-Primitive Data Type (represent collections of values, and are mutable):

Object:

A complex data type that allows you to store collections of data. Arrays and functions are special types of objects. Objects are fundamental in JavaScript.

```
let person = {
  // An object with properties
  firstName: "John",
  lastName: "Doe",
  age: 30,
```

```
};
let colors = ["red", "green", "blue"]; // An Array (which is a type of object)
console.log(typeof person); // Output: object
console.log(typeof colors); // Output: object
```

- **The Console:** Your Debugging Friend

The browser console is an essential tool for JavaScript developers. It allows you to:

- See the output of your code.
- Check for errors.
- Interact with your code dynamically.
- Debug your programs.

You can open the console in most browsers by pressing F12 (or right-clicking on a page and selecting "Inspect" or "Inspect Element", then navigating to the "Console" tab).

console.log(): The most commonly used console method. It prints whatever you pass to it onto the console.

```
console.log("Hello from JavaScript!"); // Prints a string
let myAge = 30;
console.log("My age is:", myAge); // Prints a string and a variable's value
console.log(10 + 5); // Prints the result of an expression
console.log(true); // Prints a boolean
```

Reassignment (for let):

- Change the value of cityName to another city (e.g., "Islamabad").
- Print the new value of cityName to the console.

Bonus Tips :

- Semicolons are Optional (Mostly): JavaScript doesn't always require semicolons (😉 at the end of statements, thanks to Automatic Semicolon Insertion (ASI). However, it's a good practice to use them consistently, especially when starting out, as it can prevent subtle bugs.
- Meaningful Variable Names: Always use descriptive names for your variables (e.g., productPrice instead of p). This makes your code much easier to read and understand, both for you and others.
- Comments are Your Friends: Use comments (// for single-line, /_ ... _/ for multi-line) to explain complex logic or add notes to your code. This is crucial for remembering why you wrote certain code later.

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

/*
```

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

- Developer Tools are Powerful: Get comfortable with your browser's developer tools (F12). The "Console" is just the beginning. You'll use them extensively for debugging and inspecting web pages.

Operators & Conditional Statements

📚 Topics Covered:

- **Operators:**
  - **Arithmetic:** `+`, `-`, `*`, `/`, `%`, `**`
  - **Comparison:** `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`
  - **Logical:** `&&` (AND), `||` (OR), `!` (NOT)
  - **Increment/Decrement:** `++`, `--`
- **Conditional Statements:**
  - `if`
  - `else if`
  - `else`

💻 Code Examples (`index.js`):

--- Arithmetic Operators ---

```
let num1 = 15;
let num2 = 4;
console.log("Arithmetic:");
console.log("Addition:", num1 + num2); // 19
console.log("Subtraction:", num1 - num2); // 11
console.log("Multiplication:", num1 * num2); // 60
console.log("Division:", num1 / num2); // 3.75
console.log("Modulus (Remainder):", num1 % num2); // 3
console.log("Exponentiation (15^2):", num1 ** 2); // 225
console.log("---");
```

--- Comparison Operators ---

```
let x = 10;
let y = "10";
console.log("Comparison:");
console.log("Loose Equality (10 == '10'):", x == y); // true
console.log("Strict Equality (10 === '10'):", x === y); // false
console.log("Not Equal (10 != '10'):", x != y); // false
console.log("Strict Not Equal (10 !== '10'):", x !== y); // true
```

```
console.log("Greater Than (10 > 5):", x > 5); // true
console.log("Less Than (10 < 10):", x < 10); // false
console.log("---");
```

--- Logical Operators ---

```
let sunny = true;
let warm = false;
console.log("Logical:");
console.log("Sunny AND Warm:", sunny && warm); // false
console.log("Sunny OR Warm:", sunny || warm); // true
console.log("NOT Sunny:", !sunny); // false
console.log("---");
```

--- Increment/Decrement Operators ---

```
let counter = 5;
console.log("Increment/Decrement:");
counter++; // counter is now 6
console.log("After increment:", counter); // 6
counter--; // counter is now 5
console.log("After decrement:", counter); // 5
console.log("---");
```

--- Conditional Statements ---

```
let hour = 14; // 2 PM
console.log("Conditional Statements:");
if (hour < 12) {
  console.log("Good morning!");
} else if (hour < 18) {
  console.log("Good afternoon!");
} else {
  console.log("Good evening!");
}

let score = 85;
if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");
} else {
  console.log("Grade: F");
}
```

🎯 ** Project Ideas:**

1. Even or Odd Checker:

Write a JavaScript program that takes a number (you can hardcode it for now, e.g., let num = 7;) and prints whether it's "Even" or "Odd" using the modulus operator (%) and an if/else statement.

2. Voting Eligibility Checker:

Create variables age (number) and isCitizen (boolean). Write an if/else statement that prints whether a person is "Eligible to Vote" or "Not Eligible to Vote". A person is eligible if they are 18 or older AND are a citizen.

JavaScript Loops - Repetition and Iteration

📚 Core Concepts Covered:

1. for Loop

The for loop is the most commonly used loop when you know exactly how many times you want the loop to run. It's perfect for iterating over arrays by index or performing a fixed number of repetitions.

Syntax:

```
for (initialization; condition; increment / decrement) {
  // code to be executed repeatedly
}
```

initialization:

Executed once before the loop starts. Typically used to declare and initialize a loop counter variable (e.g., let i = 0;).

condition:

Evaluated before each iteration. If true, the loop continues; if false, the loop terminates.

increment/decrement:

Executed after each iteration. Typically used to update the loop counter (e.g., i++, i--, i += 2).

When to use:

- Iterating a specific number of times.
- Looping through arrays (using numerical index).
- Counting up or down.

Example:

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

// Iterate over an array by index
const colors = ["red", "green", "blue"];
for (let i = 0; i < colors.length; i++) {
  console.log(`Color at index ${i}: ${colors[i]}`);
}
// Output:
// Color at index 0: red
// Color at index 1: green
// Color at index 2: blue
```

2. while Loop

The while loop executes a block of code as long as a specified condition evaluates to true. It's ideal when the number of iterations is unknown beforehand, and the loop needs to continue until a certain condition is met.

Syntax:

```
while (condition) {
  // code to be executed repeatedly
  // make sure to update the condition variable inside the loop
}
```

Important: You must ensure that the condition eventually becomes false inside the loop; otherwise, you'll create an infinite loop, which will crash your program.

When to use:

When the number of iterations is not predetermined (e.g., waiting for user input, processing data until a specific state is reached).

Simulating processes that continue until a resource is depleted (like energy, health points).

Example:

```
// Count from 0 to 3
let j = 0;
while (j < 4) {
  console.log(`Current value of j: ${j}`);
  j++; // Increment to avoid infinite loop
}
```

```
// Output: 0, 1, 2, 3

// Simulate energy depletion
let energy = 10;
while (energy > 0) {
  console.log(`Energy remaining: ${energy}`);
  energy -= 3;
}
// Output:
// Energy remaining: 10
// Energy remaining: 7
// Energy remaining: 4
// Energy remaining: 1
```

3. do...while Loop

The `do...while` loop is similar to the while loop, but it guarantees that the loop body executes at least once before the condition is evaluated. If the condition is true after the first execution, the loop continues.

Syntax:

```
do {
  // code to be executed at least once
  // make sure to update the condition variable inside the loop
} while (condition);
```

When to use:

When you need to perform an action at least once, regardless of the initial condition (e.g., prompting user input until valid input is received, or a game loop that runs at least once before checking for game over).

Example:

```
// This loop runs once even if k is not < 0
let k = 5;
do {
  console.log(`Current value of k: ${k}`);
  k++;
} while (k < 0);
// Output: Current value of k: 5 (runs once, then condition is false)
```

4. for...in Loop

The `for...in` loop iterates over the enumerable properties (keys) of an object. It returns the property names as strings.

Syntax:

```
for (variable in object) {
  // code to be executed for each property
}
```

Key Points:

- Primarily for iterating over object properties.
- It will iterate over inherited enumerable properties as well, which might not always be desired. You often need to use hasOwnProperty() to check if the property belongs directly to the object.
- Not recommended for iterating over arrays because it iterates over property names (indices as strings) and not values, and the order of iteration is not guaranteed for numerical indices (though modern JS engines usually maintain order for arrays). Use for or for...of for arrays.

Example:

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

for (let key in person) {
  // Using hasOwnProperty to ensure we only get own properties
  if (person.hasOwnProperty(key)) {
    console.log(`${key}: ${person[key]}`);
  }
}
// Output:
// name: Alice
// age: 30
// city: New York
```

5. for...of Loop

The `for...of` loop iterates over the values of iterable objects. This includes built-in iterables like `Array`, `String`, `Map`, `Set`, `NodeList`, etc.

Syntax:

```
for (variable of iterable) {
  // code to be executed for each value
}
```

Key Points:

- Introduced in ES6 (ECMAScript 2015).
- Provides a simpler and more direct way to iterate over values.
- Recommended for iterating over arrays and other iterable collections.
- Does not iterate over object properties directly (for that, use for...in or Object.keys(), Object.values(), Object.entries()).

Example:

```
const colors = ["red", "green", "blue"];

for (let color of colors) {
  console.log(color);
}
// Output:
// red
// green
// blue

const greeting = "Hello";
for (let char of greeting) {
  console.log(char);
}
// Output:
// H
// e
// l
// l
// o
```

6. Loop Control Statements

These statements allow you to control the flow of execution within loops:

- break: Immediately terminates the innermost loop and transfers control to the statement immediately following the loop.
- continue: Skips the rest of the current iteration of the loop and proceeds to the next iteration.

Example (break):

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    console.log("Breaking loop at 5");
    break; // Loop stops when i is 5
  }
  console.log(i);
}
// Output:
```

```
// 1
// 2
// 3
// 4
// Breaking loop at 5
```

Example (continue):

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    console.log("Skipping 3");
    continue; // Skips the rest of this iteration for i=3
  }
  console.log(i);
}
// Output:
// 1
// 2
// Skipping 3
// 4
// 5
```

Key Takeaways :

- Loops are fundamental for automating repetitive tasks.
- Choose the right loop (for, while, do...while, for...in, for...of) based on your specific iteration needs (fixed count, condition-based, object properties, iterable values).
- Always ensure your loop conditions will eventually become false to avoid infinite loops.
- break and continue provide fine-grained control over loop execution.
- for...in is for object properties (keys), for...of is for iterable values. Use them appropriately!

JavaScript Functions - Reusable Code Blocks

Core Concepts Covered:

1. What are Functions?

A function is a block of code designed to perform a particular task. It's like a mini-program within your main program.

Why use functions?

- Reusability: Write code once, use it many times.

- Organization: Break down complex problems into smaller, manageable pieces.
- Modularity: Makes your code easier to read, understand, and maintain.
- Abstraction: Hide complex implementation details, allowing you to focus on what a function does rather than how it does it.

2. Declaring Functions

There are several ways to define functions in JavaScript:

a. Function Declarations (Named Functions)

This is the most common and traditional way to define a function. They are "hoisted," meaning you can call them before they are declared in your code.

Syntax:

```
function functionName(parameter1, parameter2, ...) {
    // code to be executed
    return value; // Optional: returns a value
}
```

Example:

```
function greet(name) {
  return "Hello, " + name + "!";
}
console.log(greet("Alice")); // Output: Hello, Alice!
```

b. Function Expressions

Functions can also be defined as expressions and assigned to a variable. They are not hoisted, so you must define them before you call them.

Syntax (Anonymous Function Expression):

```
const functionName = function(parameter1, parameter2, ...) {
    // code to be executed
    return value;
};
```

Syntax (Named Function Expression - less common):

```
const functionName = function funcName(parameter1, ...) {
    // funcName is only accessible within the function itself
};
```

Example:

```
const add = function (a, b) {
  return a + b;
};
console.log(add(5, 3)); // Output: 8
```

c. Arrow Functions (ES6 / ECMAScript 2015)

A more concise syntax for writing function expressions, especially useful for short, single-line functions. They also have different this binding behavior (which we'll cover later in more detail).

Syntax:

```
const functionName = (parameter1, parameter2, ...) => {
    // code to be executed
    return value;
};

// Concise syntax for single expression functions (implicit return)
const functionName = (parameter1) => expression;
```

Example:

```
const multiply = (a, b) => a * b; // Implicit return for single expression
console.log(multiply(4, 2)); // Output: 8

const sayHi = () => {
  // No parameters, so empty parentheses
  console.log("Hi there!");
};
sayHi(); // Output: Hi there!
```

3. Calling Functions

To execute the code inside a function, you "call" or "invoke" it using its name followed by `parentheses ()`.

Example:

```
function saySomething() {
  console.log("This function is called!");
}
saySomething(); // Calls the function
```

4. Function Parameters and Arguments

Parameters: These are variables listed inside the function's parentheses in the function definition. They act as placeholders for values that will be passed into the function.

Arguments: These are the actual values that you pass to the function when you call it.

Example:

```
function greetPerson(name) {
  // 'name' is a parameter
  console.log("Hello, " + name);
}
greetPerson("Bob"); // "Bob" is an argument
```

5. Return Values

Functions can return a value using the return keyword. When return is encountered, the function stops executing, and the specified value is sent back to where the function was called. If no return statement is used, the function implicitly returns undefined.

Example:

```
function calculateArea(length, width) {
  return length * width; // Returns the calculated area
}
let area = calculateArea(10, 5); // 'area' will be 50
console.log(area); // Output: 50
```

6. Function Scope (Brief Introduction)

Variables defined inside a function are local to that function. They cannot be accessed from outside the function. This helps prevent naming conflicts and keeps code organized.

Example:

```
function myFunction() {
  let localVariable = "I am local";
  console.log(localVariable); // Accessible here
}
```

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

7. Default Parameters (ES6)

You can assign default values to parameters. If an argument is not provided for that parameter when the function is called, the default value will be used.

Syntax:

```
function functionName(param1 = defaultValue1, param2 = defaultValue2) { ... }
```

Example:

```
function greetUser(name = "Guest") {
  return `Welcome, ${name}!`;
}
console.log(greetUser("Charlie")); // Output: Welcome, Charlie!
console.log(greetUser()); // Output: Welcome, Guest!
```

8. Rest Parameters (ES6)

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array.

Syntax:

```
function functionName(param1, ...restOfArgs) { ... }
```

Example:

```
function sumAll(...numbers) {
  // 'numbers' will be an array of all arguments passed
  let total = 0;
  for (let num of numbers) {
    total += num;
  }
  return total;
}
console.log(sumAll(1, 2, 3)); // Output: 6
console.log(sumAll(10, 20, 30, 40)); // Output: 100
```

Key Takeaways :

- Functions are essential for writing clean, modular, and reusable JavaScript code.
- Understand the different ways to declare functions: declarations, expressions, and arrow functions, and their key differences (hoisting, this binding).
- Master passing arguments to parameters and handling return values.
- Utilize ES6 features like default parameters and rest parameters for more flexible functions.
- Embrace functions to break down complex problems into smaller, manageable pieces.

JavaScript Arrays & Common Methods

Core Concepts Covered:

1. What are Arrays?

An array is an ordered collection of values. Each value is called an element, and each element has a numerical position called an index (starting from 0).

Why use arrays?

- To store lists of related items (e.g., a list of names, numbers, or objects).
- To perform operations on multiple items efficiently. **Declaring Arrays:**
- Array Literal (Most Common):

```
const fruits = ["Apple", "Banana", "Cherry"];
const numbers = [1, 2, 3, 4, 5];
const mixed = [1, "hello", true, { key: "value" }]; // Arrays can hold mixed data
types
```

Array() Constructor (Less Common):

```
const emptyArray = new Array();
const fiveElements = new Array(5); // Creates an array with 5 empty slots
const specificElements = new Array("Red", "Green", "Blue");
```

2. Accessing Array Elements You access array elements using their index (position), which starts at 0

Syntax: arrayName[index]

Example:

```
const colors = ["Red", "Green", "Blue"];
console.log(colors[0]); // Output: Red
```

```
console.log(colors[1]); // Output: Green
console.log(colors[2]); // Output: Blue
console.log(colors[3]); // Output: undefined (index out of bounds)
```

3. Modifying Array Elements

You can change an element's value by assigning a new value to its index. **Example:**

```
const scores = [80, 85, 90];
scores[1] = 95; // Change the second element
console.log(scores); // Output: [80, 95, 90]
```

4. Array length Property

The length property returns the number of elements in an array. It's always one greater than the highest index. **Example:**

```
const items = ["A", "B", "C"];
console.log(items.length); // Output: 3
console.log(items[items.length - 1]); // Accessing the last element: C
```

5. Array Methods5. Adding and Removing Elements (Mutating Methods)

These methods change the original array.

push():

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

```
const fruits = ["apple", "banana"];
fruits.push("orange", "grape");
console.log(fruits); // Output: ["apple", "banana", "orange", "grape"]
```

pop ():

Removes the last element from an array and returns that element. **Example:**

```
const fruits = ["apple", "banana", "orange"];
const lastFruit = fruits.pop();
console.log(fruits); // Output: ["apple", "banana"]
console.log(lastFruit); // Output: orange
```

unshift():

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

Example:

```
const fruits = ["banana", "cherry"];
fruits.unshift("apple");
console.log(fruits); // Output: ["apple", "banana", "cherry"]
```

shift():

Removes the first element from an array and returns that element. **Example:**

```
const fruits = ["apple", "banana", "cherry"];
const firstFruit = fruits.shift();
console.log(fruits); // Output: ["banana", "cherry"]
console.log(firstFruit); // Output: apple
```

splice():

A powerful method that can add, remove, or replace elements at any position. **Syntax:**

```
array.splice(startIndex, deleteCount, item1, item2, ...)
```

- startIndex: Index at which to start changing the array.
- deleteCount: Number of elements to remove from startIndex.
- item1, item2, ...: Elements to add to the array at startIndex.

Example- Example (Remove):

```
const arr = [1, 2, 3, 4, 5];
arr.splice(2, 1); // Remove 1 element starting at index 2 (value 3)
console.log(arr); // Output: [1, 2, 4, 5]
```

- Example (Add):

```
const arr = [1, 2, 5];
arr.splice(2, 0, 3, 4); // At index 2, delete 0 elements, add 3 and 4
console.log(arr); // Output: [1, 2, 3, 4, 5]
```

Example (Replace):

```
const arr = [1, 2, 3, 4, 5];
arr.splice(1, 2, 10, 11); // At index 1, delete 2 elements (2, 3), add 10, 11
console.log(arr); // Output: [1, 10, 11, 4, 5]
```

6. Non-Mutating Array Methods (Return New Array/Value)

These methods do not change the original array; they return a new array or a new value.

indexOf():

Returns the first index at which a given element can be found in the array, or -1 if it is not present.

Example:

```
const arr = ["a", "b", "c", "a"];
console.log(arr.indexOf("b")); // Output: 1
console.log(arr.indexOf("a")); // Output: 0
console.log(arr.indexOf("d")); // Output: -1
```

includes() (ES6):

Checks if an array includes a certain value among its entries, returning true or false.

Example:

```
const arr = [10, 20, 30];
console.log(arr.includes(20)); // Output: true
console.log(arr.includes(5)); // Output: false
```

slice():

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

- Syntax:

```
array.slice(startIndex, endIndex) (endIndex is exclusive)
```

- Example:

```
const original = [1, 2, 3, 4, 5];
const newArr = original.slice(1, 4); // Elements from index 1 up to (but not
including) 4
console.log(newArr); // Output: [2, 3, 4]
console.log(original); // Output: [1, 2, 3, 4, 5] (original unchanged)
```

- **concat():**

Used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

- Example:

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

- **join():**

Joins all elements of an array into a string.

- Syntax:

```
array.join(separator) (default separator is comma)
```

- Example:

```
const words = ["Hello", "World"];
console.log(words.join(" ")); // Output: "Hello World"
console.log(words.join("-")); // Output: "Hello-World"
```

- **split()**

(String Method, but related): This is a string method that splits a string into an array of substrings.

- Syntax:

```
string.split(separator);
```

- Example:

```
const sentence = "JavaScript is fun";
const wordsArray = sentence.split(" ");
console.log(wordsArray); // Output: ["JavaScript", "is", "fun"]
```

7. Higher-Order Array Methods (Introduction)

These methods take a function as an argument (a "callback function") and apply it to each element. They are incredibly powerful for common data transformations.

- **forEach():**

Executes a provided function once for each array element. It does not return a new array.

- Example:

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

- **map():**

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

- Example:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(function(num) {
return num \* 2;
});
console.log(doubled); // Output: [2, 4, 6]
console.log(numbers); // Output: [1, 2, 3] (original array unchanged)
```

- **filter():**

Creates a new array with all elements that pass the test implemented by the provided function.

- Example:

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter(function (num) {
  return num % 2 === 0;
});
console.log(evens); // Output: [2, 4]
console.log(numbers); // Output: [1, 2, 3, 4, 5] (original array unchanged)
```

- **reduce():**

(Briefly): Executes a reducer function on each element of the array, resulting in a single output value. Used for summing, flattening arrays, etc. (More advanced, but good to know it exists).

- Example (Summing):

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0); // 0 is the initial value of accumulator
console.log(sum); // Output: 10
```

💡 Key Takeaways :

- Arrays are ordered collections, indexed from 0.
- Many array methods (e.g., push, pop, shift, unshift, splice) mutate (change) the original array.
- Other methods (e.g., indexOf, includes, slice, concat, map, filter, reduce) return a new array or value and do not modify the original array. This is often preferred for predictable code.
- Higher-order array methods `(forEach, map, filter, reduce)` are powerful for iterating and transforming arrays efficiently using callback functions.

JavaScript DOM Manipulation & Event Handling

DOM Manipulation and Event Handling.

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

Event handling allows your JavaScript code to react to user actions (like clicks, key presses) or browser events.

For our project today, we'll build a Simple Image Carousel/Slider, which is an excellent way to practice dynamic DOM updates and event listening.

🧩 Core Concepts Covered:

1. **What is the DOM** (Document Object Model)?

The DOM is a tree-like representation of the content of a web page. Every HTML element, attribute, and piece of text becomes a "node" in this tree.

JavaScript can access and modify these nodes, thereby changing the web page.

- **Document:** The root of the DOM tree, representing the entire HTML document.
- **Element:** HTML tags (e.g., , <p>, <img>) are represented as element nodes.

- **Text:** The content inside elements is represented as text nodes.

2. Accessing DOM Elements To manipulate an element, you first need to select it.

`document.getElementById('idName'):` Selects a single element by its unique id attribute. Returns null if not found.

Example:

```
const myDiv = document.getElementById("myDiv");
```

`document.querySelector('selector'):`

Selects the first element that matches a specified CSS selector.

Example:

```
const firstParagraph = document.querySelector("p");
```

Example:

```
 const myClassElement = document.querySelector('.my-class');document.
```

`querySelectorAll('selector'):`

Selects all elements that match a specified CSS selector. Returns a NodeList (which is like an array, but not exactly).

Example:

```
const allButtons = document.querySelectorAll("button");
```

`document.getElementsByClassName('className'):`

Selects all elements with a specific class name. Returns an HTMLCollection.

Example:

```
const allItems = document.getElementsByClassName("list-item");
```

`document.getElementsByTagName('tagName')`: Selects all elements with a specific tag name. Returns an HTMLCollection.

Example:

```
const allDivs = document.getElementsByTagName("div");
```

3. Modifying DOM Elements Once you have an element, you can change its properties.

`element.textContent`:

Gets or sets the text content of an element (plain text only).

Example:

```
myDiv.textContent = "New text here";
```

`element.innerHTML`:

Gets or sets the HTML content (can include HTML tags). Use with caution to avoid XSS vulnerabilities if dealing with untrusted user input.

Example:

```
myDiv.innerHTML = "<strong>Bold text!</strong>";
```

`element.setAttribute('attributeName', 'value')`:

Sets the value of an attribute.

Example:

```
myImage.setAttribute("src", "new-image.jpg");
```

`element.removeAttribute('attributeName')`:

Removes an attribute.

Example:

```
myImage.removeAttribute("alt");
```

`element.classList.add('className):`

Adds a CSS class.

Example:

```
myDiv.classList.add("active");
```

`element.classList.remove('className'):`

Removes a CSS class.

Example:

```
myDiv.classList.remove("hidden");
```

`element.classList.toggle('className'):`

Toggles a CSS class (adds if not present, removes if present).

Example:

```
myDiv.classList.toggle("highlight");
```

`element.style.propertyName = 'value':`

Sets inline CSS styles.

Example:

```
myDiv.style.backgroundColor = "blue";
```


```
myDiv.style.backgroundColor = "blue";
```


---

4. Creating and Appending Elements

You can dynamically build new HTML elements and add them to the page.

`document.createElement('tagName'):`

Creates a new HTML element node.

Example:

```
const newParagraph = document.createElement("p");
```

`parentElement.appendChild(childElement):`

Appends a child element to the end of a parent element.

Example:

```
myDiv.appendChild(newParagraph);
```

`parentElement.prepend(childElement):`

Adds a child element to the beginning of a parent element.

Example:

```
myDiv.prepend(newTitle);
```

`parentElement.insertBefore(newElement, referenceElement):`

Inserts newElement before referenceElement within parentElement.

Example:

```
myList.insertBefore(newItem, myList.children[1]);
```

---

5. Removing Elements Elements can be removed from the DOM.parentElement.

`removeChild(childElement):`

Removes a specified child element from its parent.

Example:

```
myList.removeChild(itemToRemove);
``  ``
---
```

```
`element.remove() (Modern):`


A simpler way to remove an element directly.

Example:
```js
itemToRemove.remove();
```

---

6. Event Handling Events are actions that happen in the system you are programming, such as a user clicking a button, a page loading, or a video playing. `element.addEventListener('eventName', functionName):`

- The primary way to register an event handler.eventName:

The type of event (e.g., 'click', 'mouseover', 'keydown', 'submit').

`functionName:`

- The function to execute when the event occurs (the "event handler" or "callback function").

The Event Object: When an event occurs, an `Event` object is automatically passed to the event handler function. This object contains useful information about the event.

- `event.target:` The DOM element that triggered the event.
- `event.preventDefault():` Prevents the default action of an event (e.g., prevents a form from submitting, or a link from navigating).

Example:

```
const myButton = document.getElementById("myButton");
myButton.addEventListener("click", function (event) {
  console.log("Button clicked!");
  console.log("Event target:", event.target);
});
```

Key Takeaways :

- The DOM is your JavaScript's gateway to interacting with web page content.
- You must first select elements before you can manipulate them.
- `textContent` for plain text, `innerHTML` for HTML content (use with caution).
- `classList` is powerful for managing CSS classes.
- `addEventListener()` is the standard way to make your page interactive by reacting to events.
- The Event object provides crucial information about what happened.
```

Scopes - Global, Function & Block

Core Concepts Covered:

1. What is Scope?

Scope determines where in your program you can access a variable. In simple terms, it defines the "lifetime" and "visibility" of a variable. JavaScript has three main types of scope: Global, Function, and Block.

2. Global Scope

A variable declared in the global scope (outside of any function or block) is accessible from anywhere in your code, including inside functions and blocks.

- **Declaring a global variable:**

```
const message = "Hello from the global scope!";

function sayHi() {
  // You can access 'message' here
  console.log(message);
}
sayHi();
```

Warning: It's generally best to avoid creating too many global variables, as they can lead to naming conflicts and make your code harder to manage.

3. Function Scope (Local Scope)

A variable declared inside a function is only accessible within that function. It is a local variable. You cannot access it from outside the function.

- **Declaring a function-scoped variable:**

```
function logMessage() {
  const greeting = "Welcome!"; // This is local to logMessage()
  console.log(greeting);
}

logMessage(); // Output: Welcome!

// The next line will cause an error because 'greeting' is not accessible here
// console.log(greeting); // ReferenceError: greeting is not defined
```

4. Block Scope

Block scope is introduced with let and const. A variable declared with let or const inside a code block (defined by curly braces {}) is only accessible within that block. This includes if statements, for loops, and other code blocks.

- **Declaring a block-scoped variable:**

```
let x = 10;
if (x > 5) {
  const y = 20; // 'y' is block-scoped
  console.log(x); // Output: 10 (x is in a wider scope)
  console.log(y); // Output: 20 (y is accessible here)
}

// The next line will cause an error
// console.log(y); // ReferenceError: y is not defined
```

5. var vs. let and const

This is a critical distinction related to scope:

- **var:**

Is function-scoped. Variables declared with var inside a block will "leak" and be accessible outside that block (but still confined to the function they are in).

- **let and const:**

Are block-scoped. This is the modern and preferred way to declare variables, as it helps prevent accidental variable overwrites and makes your code more predictable.

Key Takeaways:

- Scope defines accessibility. It's the set of rules that determines where a variable can be used.
- Global scope variables are accessible everywhere. Use them sparingly to avoid conflicts.
- Function scope variables are only accessible within the function they are declared in.
- Block scope is created by curly braces {} and applies to variables declared with let and const. This is the modern standard for controlling variable visibility.
- `var` is function-scoped, which can lead to unexpected behavior (like a loop counter "leaking" outside the loop). Always prefer let and const for new code.
- Nested scopes can access variables from their parent scopes, but parent scopes cannot access variables from their nested child scopes.

Conditional Statements

Core Concepts Covered:

1. if, else if, and else

This is the most common way to handle conditional logic. The if statement executes a block of code if its condition is true. You can chain `else if` statements to check for more conditions and use an else block as a fallback for all other cases.

Syntax:

```
if (condition1) {
  // Code to run if condition1 is true
} else if (condition2) {
  // Code to run if condition2 is true
} else {
  // Code to run if all conditions are false
}
```

Example:

```
const age = 19;
if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
```

2. switch Statement

A switch statement is an alternative to a long if...else if... chain, especially when you are comparing a single value against multiple possible cases.

Syntax:

```
switch (expression) {
  case value1:
    // Code to run if expression === value1
    break; // Important! Exits the switch statement
  case value2:
    // Code to run if expression === value2
    break;
  default:
  // Code to run if no cases match
}
```

Example:

```
const day = "Monday";
switch (day) {
  case "Monday":
    console.log("Start of the week!");
    break;
  case "Friday":
    console.log("Weekend is near!");
    break;
  default:
    console.log("Just another day.");
}
```

The `break` keyword is crucial. Without it, the code will "fall through" and execute the code for the next case as well.

JavaScript Objects & OOP Basics (Project: Simple Product Catalog)

Core Concepts Covered:

1. What are Objects?

In JavaScript, almost everything is an object or can behave like one. An object is an independent entity, with properties and type. It's a collection of key-value pairs.Key (or Property Name):

A string(or Symbol) that uniquely identifies a piece of data within the object.Value: Any JavaScript data type (primitive, array, function, or even another object).

2.Why use objects?

To represent real-world entities (e.g., a person, a car, a product).To group related data and functionality together.To organize and structure complex data.

3. Creating Objects.

A Object Literal (Most Common and Recommended)This is the simplest and most widely used way to create objects.

Syntax:

```
const objectName = {
  key1: value1,
  key2: value2,
  // ...
};
```

Example:

```
const person = {
  firstName: "Alice",
  lastName: "Smith",
  age: 30,
  isStudent: false,
};
console.log(person);
// Output: { firstName: 'Alice', lastName: 'Smith', age: 30, isStudent: false }
```

- b. Using the new `Object()` Constructor (Less Common)You can create an empty object and then add properties to it.

Syntax:

```
const objectName = new Object();
objectName.key1 = value1;
objectName.key2 = value2;
```

Example:

```
const car = new Object();
car.make = "Toyota";
car.model = "Camry";
car.year = 2022;
console.log(car);
// Output: { make: 'Toyota', model: 'Camry', year: 2022 }
```

- **c.** Constructor Functions (Traditional OOP)Before ES6 Classes, constructor functions were the primary way to create multiple objects of the same "type" or "blueprint." **Syntax:**

```
function ConstructorName(param1, param2) {
  this.prop1 = param1;
  this.prop2 = param2;
  this.method = function () {
    // object method
    // ...
  };
}
```

- **const instance = new ConstructorName(arg1, arg2);**

Example:

```
function Book(title, author, pages) {
  this.title = title;
  this.author = author;
  this.pages = pages;
  this.getInfo = function () {
    return `${this.title} by ${this.author}, ${this.pages} pages.`;
  };
}
const book1 = new Book("The Hobbit", "J.R.R. Tolkien", 310);
console.log(book1.getInfo()); // Output: The Hobbit by J.R.R. Tolkien, 310 pages.
```

4. Accessing Object Propertiesa.

- Dot Notation (Most Common)Use when you know the property name beforehand and it's a valid JavaScript identifier (no spaces, starts with letter/underscore/$).

Syntax:

```
  objectName.propertyNameExample:const user = { name: "John", email:
"john@example.com" };
console.log(user.name);  // Output: John
console.log(user.email); // Output: john@example.com
```

- b. Bracket Notation Use when:

The property name contains spaces or special characters.The property name is stored in a variable (dynamic access).The property name is a number (though not common for object keys).

Syntax:

```
 objectName['propertyName'] or
objectName[variableHoldingPropertyName]Example:const product = { "product name":
"Laptop", price: 1200 };
console.log(product["product name"]); // Output: Laptop

const prop = "price";
console.log(product[prop]); // Output: 1200 (dynamic access)
```

5. Modifying, Adding, and Deleting Properties:

- **Modifying:** Assign a new value to an existing property.

```
const person = { name: "Alice", age: 30 };
person.age = 31; // Modify age
```

```
console.log(person); // Output: { name: 'Alice', age: 31 }
```

- ***Adding:** Assign a value to a new property name.

```
const person = { name: "Alice" };
person.city = "New York"; // Add new property
console.log(person); // Output: { name: 'Alice', city: 'New York' }
```

- **Deleting:** Use the delete operator.

```
const person = { name: "Alice", age: 30 };
delete person.age; // Delete age property
console.log(person); // Output: { name: 'Alice' }
```

6. Nested Objects

Object values can themselves be other objects, creating nested structures.

Example:

```
const student = {
  id: 101,
  name: "Bob",
  address: {
    street: "123 Main St",
    city: "Anytown",
    zip: "12345",
  },
  grades: {
    math: 90,
    science: 85,
  },
};
console.log(student.address.city); // Output: Anytown
console.log(student.grades.math); // Output: 90
```

7. Object Methods (this Keyword)

Functions stored as object properties are called methods. Inside a method, the this keyword refers to the object that the method belongs to. **Example:**

```
const dog = {
  name: "Buddy",
  breed: "Golden Retriever",
  bark: function () {
```

```
    console.log(this.name + " says Woof!"); // 'this' refers to the 'dog' object
  },
  // Arrow functions behave differently with 'this', capturing 'this' from their
surrounding scope.
  // For methods directly on objects, traditional function syntax is often clearer
for 'this'.
  introduce: () => {
    console.log(`Hi, I'm ${dog.name} the ${dog.breed}.`); // Explicitly using
dog.name
    // console.log(`Hi, I'm ${this.name} the ${this.breed}.`); // 'this' here
would NOT refer to 'dog'
  },
};
dog.bark(); // Output: Buddy says Woof!
dog.introduce(); // Output: Hi, I'm Buddy the Golden Retriever.
```

8. Useful Object Methods (Static Methods on Object constructor)

These are built-in methods on the global Object constructor that help you work with objects.Object.keys(obj): Returns an array of a given object's own enumerable property names (keys). **Example:**

```
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.keys(obj)); // Output: ['a', 'b', 'c']
Object.values(obj): Returns an array of a given object's own enumerable property
values.Example:const obj = { a: 1, b: 2, c: 3 };
console.log(Object.values(obj)); // Output: [1, 2, 3]
Object.entries(obj): Returns an array of a given object's own enumerable string-
keyed property [key, value] pairs.Example:const obj = { a: 1, b: 2, c: 3 };
console.log(Object.entries(obj)); // Output: [['a', 1], ['b', 2], ['c', 3]]
```

💡 Key Takeaways :

- `Objects` are fundamental for structuring complex, related data using key-value pairs.
- Use `dot notation` for known, valid property names; `bracket notation` for dynamic access or invalid names.
- Functions within `objects` are called `methods`; this inside a method refers to the object itself.
- `Object.keys()`, `Object.values()`, and `Object.entries()` are useful for iterating over object data.
- Arrays and objects often work together (e.g., an array of objects) to represent collections of structured data.

Object Manipulation and Destructuring

🔄 Core Concepts Covered:

1. Object Manipulation Methods

These methods are static functions on the global Object class that allow you to convert an object's properties into arrays, which you can then easily process using array methods like forEach, map, filter, and reduce.

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

```
const book = { title: '1984', author: 'George Orwell', year: 1949 };
const keys = Object.keys(book);
console.log(keys); // Output: ['title', 'author', 'year']
```

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

```
const book = { title: '1984', author: 'George Orwell', year: 1949 };
const values = Object.values(book);
console.log(values); // Output: ['1984', 'George Orwell', 1949]
```

- `Object.entries(obj)`: Returns an array of the object's [key, value] pairs. This is often the most versatile method.

```
const book = { title: '1984', author: 'George Orwell', year: 1949 };
const entries = Object.entries(book);
console.log(entries);
// Output:
// [
//   ['title', '1984'],
//   ['author', 'George Orwell'],
//   ['year', 1949]
// ]
```

You can easily iterate over these entries:

```
for (const [key, value] of Object.entries(book)) {
  console.log(`${key}: ${value}`);
}
// title: 1984
// author: George Orwell
// year: 1949
```

2. Object Destructuring

Destructuring is a modern JavaScript syntax that makes it easy to unpack values from arrays or properties from objects into distinct variables. It's an elegant way to avoid repetitive code when accessing object properties.

- `Basic Destructuring`: You can extract specific properties directly into variables.

```
const user = { name: 'Alice', age: 30 };
const { name, age } = user;
console.log(name); // Output: 'Alice'
console.log(age);  // Output: 30
```

Destructuring with Aliases: You can rename the extracted variables.

```
const user = { name: 'Alice', age: 30 };
const { name: userName, age: userAge } = user;
console.log(userName); // Output: 'Alice'
```

Destructuring in Function Parameters: This is a very common and useful pattern for extracting specific properties from an object passed to a function.

```
function printBookDetails({ title, author }) {
  console.log(`Title: ${title}, Author: ${author}`);
}
const book = { title: 'The Hobbit', author: 'J.R.R. Tolkien', year: 1937 };
printBookDetails(book); // Output: Title: The Hobbit, Author: J.R.R. Tolkien
```

Key Takeaways :

- `Object.keys()`, `Object.values()`, and `Object.entries()` are excellent for iterating over object properties.
- Object destructuring is a powerful syntax for extracting data from objects, leading to cleaner and more readable code.
- Combining these techniques with array methods like map and forEach allows for efficient and expressive data transformation.

Introduction to Set and Map Data Structures

📚 Core Concepts Covered:

1. Set Data Structure

A Set is a collection of unique values. Each value can only occur once in a Set. It's useful when you need to store a list of items where duplicates are not allowed, and the order of insertion doesn't matter (though iteration order is preserved).

Key Characteristics:

- *Unique Values:* Automatically handles duplicates; adding an existing value has no effect.
- *No Indexed Access:* You cannot access elements by index (like mySet[0]).
- *Iterable:* You can iterate over Set elements using `for...of` or `forEach()`.

Creating a Set:

```
const mySet = new Set();
const anotherSet = new Set([1, 2, 2, 3, 'hello', 'hello']); // Initializes with
unique values
console.log(anotherSet); // Output: Set(4) {1, 2, 3, "hello"}
```

Common Set Methods:

- `set.add(value):` Adds a new value to the Set. Returns the Set object.

```
mySet.add(10);
mySet.add('apple');
mySet.add(10); // This has no effect
console.log(mySet); // Output: Set(2) {10, "apple"}
```

- `set.delete(value)`: Removes a value from the Set. Returns true if the value was present and removed, false otherwise.

```
mySet.delete('apple');
console.log(mySet); // Output: Set(1) {10}
```

`set.has(value):` Returns true if the value exists in the Set, false otherwise.

```
console.log(mySet.has(10));    // Output: true
console.log(mySet.has('banana')); // Output: false
```

`set.size`: Returns the number of unique values in the Set.

```
console.log(mySet.size); // Output: 1
```

`set.clear()`: Removes all values from the Set.

```

```

```
mySet.clear();
console.log(mySet.size); // Output: 0
```

Iterating a Set:

```
const fruits = new Set(['apple', 'banana', 'orange']);
fruits.forEach(fruit => console.log(fruit));
// OR
for (const fruit of fruits) {
    console.log(fruit);
}
```

Use Cases for Set:

- Removing duplicate elements from an array: `[...new Set(myArray)]`
- Checking for the presence of an item quickly.
- Tracking unique visitors, unique tags, etc.

2. Map Data Structure

A Map is a collection of key-value pairs, similar to an object, but with a few significant differences that make it more powerful and flexible for certain scenarios.

Key Characteristics:

Any Data Type as Key: Unlike objects, where keys are always converted to `strings`, Maps allow you to use any data type `(numbers, booleans, objects, functions, null, NaN)` as keys.

- Order of Insertion: The order of elements is preserved.
- Iterable: You can iterate over Map elements using `for...of` or `forEach()`.

Creating a Map:

```
const myMap = new Map();
const anotherMap = new Map([
    ['name', 'Alice'],
    [1, 'one'],
    [true, 'yes']
]);
console.log(anotherMap);
// Output: Map(3) {"name" => "Alice", 1 => "one", true => "yes"}
```

Common Map Methods:

- `map.set(key, value)`: Adds or updates a key-value pair. Returns the Map object.

```
myMap.set('color', 'blue');
myMap.set(123, 'numberKey');
const objKey = { id: 1 };
myMap.set(objKey, 'object as key');
console.log(myMap);
// Output: Map(3) {"color" => "blue", 123 => "numberKey", {…} => "object as key"}
```

- `map.get(key)`: Retrieves the value associated with a given key. Returns undefined if the key does not exist.

```
console.log(myMap.get('color'));   // Output: "blue"
console.log(myMap.get(objKey)); // Output: "object as key"
console.log(myMap.get('nonExistent')); // Output: undefined
```

- `map.has(key)`: Returns true if the key exists in the Map, false otherwise.

```
console.log(myMap.has('color')); // Output: true
```

- `map.delete(key)`: Removes a key-value pair by key. Returns true if the element existed and was removed, false otherwise.

```
myMap.delete('color');
console.log(myMap.has('color')); // Output: false
```

`map.size`: Returns the number of key-value pairs in the Map.

```
console.log(myMap.size); // Output: 2
```

`map.clear()`: Removes all key-value pairs from the Map.

```
myMap.clear();
console.log(myMap.size); // Output: 0
```

Iterating a Map:

```
const userRoles = new Map([['admin', 'John'], ['editor', 'Jane']]);
userRoles.forEach((value, key) => console.log(`${key}: ${value}`));
// OR
for (const [key, value] of userRoles) {
```

```
    console.log(`${key}: ${value}`);
}
// OR (get keys or values only)
for (const key of userRoles.keys()) { console.log(key); }
for (const value of userRoles.values()) { console.log(value); }
```

Use Cases for Map:

- Storing data where keys are not strings (e.g., DOM elements, objects).
- Maintaining insertion order of key-value pairs.
- When you frequently add/remove key-value pairs.
- Implementing caches or look-up tables.

Key Takeaways :

- Set is ideal for storing collections of unique values. It's great for quickly checking if an item exists or removing duplicates.
- Map is ideal for storing key-value pairs where keys can be of any data type (not just strings, like in plain objects). It maintains insertion order and offers clear methods for manipulation.
- Choose Set when you need a unique collection and Map when you need flexible key-value storage.

Event Handling: A Guide to User Interaction

What is an Event?

In web development, an event is an action or occurrence that happens in the browser, such as a user clicking a button, a page finishing loading, or a form being submitted. Events are the fundamental way that JavaScript connects with and responds to the user.

Why is Event Handling Important?

Event handling is what makes a website interactive. Without it, a webpage would be a static document. By handling events, we can create dynamic experiences like:

- Showing or hiding content based on a user's click.
- Validating form data before it's sent to a server.
- Updating a user interface in real-time.
- Making games and animations.

Key Concepts in Event Handling

1. Event Listeners An event listener is a function that waits for a specific event to occur on a particular element. When the event happens, the function is executed. The primary method for adding an event listener in JavaScript is addEventListener().

Syntax: `element.addEventListener('event-name', function-to-run);`

For example, to run a function when a button is clicked, you would write:

```
myButton.addEventListener('click', sayHello);
```

2. The Event Object When an event occurs, JavaScript creates an Event object. This object is automatically passed as an argument to the event listener's function. It contains important information about the event, such as:

`event.target`: The DOM element that triggered the event.

`event.type`: The type of event that occurred (e.g., 'click').

`event.preventDefault()`: A method to stop the default browser behavior for a given event (e.g., preventing a form from refreshing the page on submit).

3. Common Events While there are many types of events, these are some of the most frequently used:

- **click:** When a user clicks an element.
- **mouseover / mouseout**: When the mouse pointer enters or leaves an element.
- **keydown / keyup**: When a user presses or releases a key on the keyboard.
- **submit**: When a form is submitted.
- **load**: When a page or image has finished loading.

4. Event Flow: Bubbling & Capturing

Events don't just happen on a single element; they travel through the DOM tree. There are two phases to this journey:

- **Capturing Phase:** The event starts at the top of the DOM tree (the window object) and travels down to the target element.
- **Bubbling Phase:** The event starts at the target element and bubbles up the DOM tree to the top.

By default, event listeners trigger in the bubbling phase. Understanding event flow is crucial for advanced topics like event delegation, where you can listen for events on a parent element instead of every child element.

Key Takeaways ✨:

- The `addEventListener()` method is your primary tool for listening for events.

- The Event object is your key to getting information about what happened, like the element that was clicked.
- Always be aware of the default behavior of elements (like a form submitting) and use `event.preventDefault()` when you need to stop it.
- Events have a bubbling phase, which is how they propagate up the DOM tree from the target element.

JavaScript Error Handling & Debugging

Core Concepts Covered:

1. Understanding Errors Errors are problems that occur during the execution of a program. They can prevent your code from running correctly or even cause it to crash.

Types of Errors:

- Syntax Errors:

Occur when you violate JavaScript's grammar rules (e.g., missing a parenthesis, a semicolon). These are usually caught by the browser/interpreter before execution.

// **Example:**

Syntax Error (missing closing parenthesis)

```
// console.log("Hello";
```

Runtime Errors (Exceptions):

Occur during the execution of the program. JavaScript throws an "exception" when something unexpected happens (e.g., trying to access a property of null, dividing by zero, network failure). These are what try...catch blocks are designed to handle.

```
// Example: Runtime Error (TypeError)
// const obj = null;
// console.log(obj.property); // Cannot read properties of null
```

Logical Errors:

The code runs without crashing, but it produces incorrect results because of a flaw in the program's logic. These are the hardest to find and require careful debugging.

```
// Example: Logical Error (intended to add, but subtracts)
// function add(a, b) { return a - b; }
```

```
// console.log(add(5, 3)); // Expected 8, got 2
```

2. try...catch Statement

The `try...catch` statement is JavaScript's primary mechanism for handling runtime errors (exceptions). It allows you to "try" a block of code and "catch" any errors that occur within it, preventing the entire script from crashing.

Syntax:

```
try {
    // Code that might throw an error
    // If an error occurs here, execution jumps to the catch block
} catch (error) {
    // Code to handle the error
    // The 'error' parameter contains information about the error that occurred
    console.error("An error occurred:", error);
} finally {
    // Optional: Code that will always execute, regardless of whether
    // an error occurred or was caught. Useful for cleanup.
}
```

- try block: Contains the code that you want to monitor for errors.

- catch block: Executes if an error is thrown within the try block. It receives an Error object as an argument.

finally block (Optional): Executes after both the try and catch blocks, regardless of whether an error occurred or was handled. It's often used for cleanup operations (e.g., closing files, releasing resources).

Example:

```
function divide(a, b) {
    try {
        if (b === 0) {
            throw new Error("Division by zero is not allowed."); // Manually throw
an error
        }
        console.log("Result:", a / b);
    } catch (error) {
        console.error("Caught an error:", error.message);
    } finally {
        console.log("Division attempt finished.");
    }
}
divide(10, 2);  // Output: Result: 5, Division attempt finished.
divide(10, 0);  // Output: Caught an error: Division by zero is not allowed.,
Division attempt finished.
```

3. throw Statement

The throw statement allows you to create and throw a custom error (or any value) explicitly. When throw is executed, the normal flow of execution stops, and control is passed to the nearest catch block.

Syntax: throw value; (where value can be a string, number, object, or an Error object).

Best Practice: Always throw an Error object (or a custom error object inherited from Error) as it provides useful properties like name, message, and stack (call stack).

throw new Error("Something went wrong!");

4. The Error Object

- When an error occurs, JavaScript creates an Error object (or a more specific error type like TypeError, ReferenceError, etc.) and passes it to the catch block.
- Key Properties of Error objects:
- name: The type of error (e.g., "TypeError", "ReferenceError", "Error" for generic errors).
- message: A human-readable description of the error.
- stack:(Non-standard but widely supported) A string representing the call stack at the moment the error was thrown, useful for debugging.

5. Common Built-in Error Types:

ReferenceError: Occurs when you try to access a variable that hasn't been declared.

```
// console.log(undeclaredVar); // ReferenceError
```

TypeError: Occurs when an operation is performed on a value that is not of the expected type (e.g., calling a non-function, accessing properties of null/undefined).

```
// const x = 10;
// x.toUpperCase(); // TypeError: x.toUpperCase is not a function

SyntaxError: Occurs when JavaScript code is syntactically invalid.

// eval("alert('Hello');;"); // SyntaxError: Unexpected token ';'

RangeError: Occurs when a number is outside an allowable range of values.

// (100).toExponential(200); // RangeError: toExponential() argument must be
between 0 and 100
```

6. Debugging Techniques

Debugging is the process of finding and fixing errors or bugs in your code.

`console.log()`: The simplest and most common debugging tool. You can log values of variables, messages, or objects at different points in your code to understand its flow.

`console.warn()`: For warnings.

`console.error()`: For errors.

`console.info()`: For informational messages.

`console.table()`: To display tabular data (like arrays of objects).

`console.dir()`: To display an interactive list of the properties of a specified JavaScript object.

Browser Developer Tools (DevTools):

Your most powerful debugging ally.

- Opening DevTools: Right-click on your web page -> "Inspect" (or F12, Cmd+Option+I on Mac).
- Elements Tab: Inspect and modify HTML and CSS in real-time.
- Console Tab: View console.log messages, errors, warnings, and execute JavaScript code directly.
- Sources Tab: This is where the magic happens for JavaScript debugging:

Breakpoints: Click on a line number in your JavaScript file to set a breakpoint. When execution reaches this line, it will pause.

Stepping Controls:

Step Over: Execute the current line and move to the next. If the line contains a function call, it executes the entire function without stepping into it.

Step Into: Execute the current line. If it contains a function call, it steps into that function.

Step Out: If you're inside a function, execute the rest of the function and jump back to where it was called.

Resume Script Execution: Continue running the script until the next breakpoint or the end of the script.

Scope Panel: View the values of variables in the current scope (local, closure, global).

Variables Panel: Add specific variables to "watch" their values as you step through code.

Call Stack: See the sequence of function calls that led to the current point of execution.

Network Tab: Monitor network requests (API calls, image loads) – see their status, timing, and response data.

Application Tab: Inspect local storage, session storage, cookies, and more.

---

1. Basic try...catch:

```
function processArray(arr) {
    try {
        if (arr === null) {
```

```
            throw new TypeError("Input array cannot be null.");
        }
        console.log("Accessing index 10:", arr[10]);
    } catch (error) {
        console.error("An error occurred in processArray:", error.message);
        // Also log the full error object for more details
        console.error(error);
    }
}
processArray([1, 2, 3]); // Output: Accessing index 10: undefined, then no error
processArray(null);      // Output: An error occurred in processArray: Input array
cannot be null. (and full error object)
```

2. Custom throw Error:

```
function checkAge(age) {
    if (age < 0 || age > 120) {
        throw new RangeError("Age must be between 0 and 120.");
    }
    console.log(`Age ${age} is valid.`);
}

try {
    checkAge(30);   // Output: Age 30 is valid.
    checkAge(150);  // Throws RangeError
} catch (error) {
    if (error instanceof RangeError) {
        console.error("Age validation error:", error.message);
    } else {
        console.error("Unexpected error:", error.message);
    }
}

try {
    checkAge(-5);   // Throws RangeError
} catch (error) {
    if (error instanceof RangeError) {
        console.error("Age validation error:", error.message);
    } else {
        console.error("Unexpected error:", error.message);
    }
}
// Expected output for invalid ages: Age validation error: Age must be between 0
and 120.
```

3. finally Block Usage:

```
function resourceOperation(shouldFail = false) {
    try {
```

```
        console.log("Resource opened.");
        if (shouldFail) {
            throw new Error("Failed to process resource.");
        }
        console.log("Resource processed successfully.");
    } catch (error) {
        console.error("Caught error during operation:", error.message);
    } finally {
        console.log("Resource closed."); // Always runs
    }
}

console.log("\n--- Successful operation ---");
resourceOperation(false);
// Output:
// Resource opened.
// Resource processed successfully.
// Resource closed.

console.log("\n--- Failed operation ---");
resourceOperation(true);
// Output:
// Resource opened.
// Caught error during operation: Failed to process resource.
// Resource closed.
```

4. Debugging with console.table and console.dir:

```
const users = [
    {id: 1, name: "Alice", email: "alice@example.com"},
    {id: 2, name: "Bob", email: "bob@example.com", isActive: true},
    {id: 3, name: "Charlie", email: "charlie@example.com", role: "admin"}
];

console.log("--- console.table(users) ---");
console.table(users);

console.log("\n--- console.dir(users[0]) ---");
console.dir(users[0]);

console.log("\n--- console.dir(users[1]) ---");
console.dir(users[1]);

// Open browser console (F12) to see the formatted output.
// console.table will show a nice table.
// console.dir will show an expandable object tree.
```

5. Debugging with Breakpoints (Conceptual):

```
- Breakpoint Location: Set a breakpoint on the line: let subtotal = price *
quantity;

- Variables to Watch: price, quantity, subtotal.

- Reasoning: When execution pauses at this line, you can inspect the values of
price and quantity before the multiplication happens. If quantity is a string
(e.g., "2"), you'll immediately see that price * quantity will result in NaN
because JavaScript tries to perform arithmetic on a string that cannot be fully
converted to a number. This tells you the issue is with the quantity's data type
before the calculation. You could then step over to let total = subtotal * (1 +
taxRate); and see total also become NaN, confirming the propagation of the error.
```

Key Takeaways :

- Errors are inevitable; mastering error handling makes your applications robust.
- `try...catch` is your primary tool for gracefully handling runtime errors.
- Use throw new `Error()` for custom, meaningful errors.
- The finally block is perfect for cleanup operations that must always run.
- Browser Developer Tools (especially the Console and Sources tabs with breakpoints) are indispensable for finding and fixing bugs efficiently.
- `console.log` and its variations are simple yet powerful debugging aids.

ES6+ Features Recap

Today, we're solidifying our understanding of three fundamental ES6+ features: Template Literals, the Spread/Rest Operators, and Default Parameters. We'll review each concept and then apply all three in a single practice exercise.

1. Template Literals (``)

Template literals are an improved way to create strings. They use backticks `` instead of single or double quotes, allowing you to embed expressions and variables directly within the string. This makes string concatenation much cleaner and more readable.

Key Features:

- Multi-line Strings: You can write strings that span multiple lines without needing the \n character.
- Expression Interpolation: You can insert variables or JavaScript expressions using ${...}.

Example:

```
const name = 'Alice';
const age = 30;
const message = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(message); // "Hello, my name is Alice and I am 30 years old."
```

2. Spread (...) & Rest (...) Operators

This is a single operator with two distinct uses depending on the context.

The Spread Operator

When used in an array literal or function call, the spread operator unpacks the elements of an iterable (like an array) or the properties of an object. It's often used to create a new array or object without mutating the original.

Example (Arrays):

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArr = [...arr1, ...arr2]; // Spreads the elements of arr1 and arr2
into a new array.

console.log(combinedArr); // [1, 2, 3, 4, 5, 6]
```

Example (Objects):

```
const user = { name: 'Bob', age: 25 };
const updatedUser = { ...user, city: 'New York' }; // Spreads user properties and
adds a new one.

console.log(updatedUser); // { name: 'Bob', age: 25, city: 'New York' }
```

The Rest Operator

When used in a function's parameters, the rest operator gathers all remaining arguments into a single array. This is perfect for functions that can accept a variable number of arguments.

Example:

```
function greet(name = 'Guest') {
    return `Hello, ${name}!`;
}

console.log(greet('Charlie')); // "Hello, Charlie!"
console.log(greet()); // "Hello, Guest!"
```

3. Default Parameters

Default parameters allow you to initialize a function parameter with a default value if an argument is not provided or is undefined. This prevents errors and makes your functions more robust.

Example:

```
function greet(name = 'Guest') {
    return `Hello, ${name}!`;
}

console.log(greet('Charlie')); // "Hello, Charlie!"
console.log(greet()); // "Hello, Guest!"
```

Key Takeaways:

- Readability: `Template literals` make code cleaner and easier to read by eliminating complex string concatenation.
- Immutability: The spread operator is a core tool for writing predictable, bug-free code by creating new arrays and objects instead of modifying existing ones.
- Flexibility: The `rest operator` and `default parameters` make your functions more robust and flexible, allowing them to handle a wider range of inputs without explicit checks.

JSON - Parsing and Stringifying

Core Concepts Covered:

1. What is JSON?

JSON is a text format that looks very similar to a JavaScript object literal. However, there are a few key differences:

- Keys must be double-quoted strings.
  - Valid JSON: `{"name": "Alice"}`
  - Invalid JSON `(JS object literal): {name: "Alice"}`
- Values must be primitive data types (strings, numbers, booleans, null) or other JSON objects/arrays.
  - Valid JSON: `{"isStudent": true, "age": 25}`
  - Invalid JSON: `{ "func": () => {} } (Functions are not allowed in JSON)`

2. Converting a JavaScript Object to a JSON String (`JSON.stringify()`)

This method takes a JavaScript value (usually an object or array) and converts it into a JSON string. This process is called serialization. You use this method when you need to send data to a server or save it to a file.

Syntax:

`JSON.stringify(value, replacer, space)`

value: The value to convert.

- replacer (optional): A function or array that filters which properties are included.
- space (optional): A number or string to use for indentation, making the output human-readable.

Example:

```
const userProfile = {
  name: 'Bob',
  age: 30,
  isLoggedIn: true,
};
// Convert to a compact JSON string

const jsonString = JSON.stringify(userProfile);
console.log(jsonString); // Output: {"name":"Bob","age":30,"isLoggedIn":true}

// Convert with indentation for readability
const readableJsonString = JSON.stringify(userProfile, null, 2);
console.log(readableJsonString);
/*
Output:
{
  "name": "Bob",
  "age": 30,
  "isLoggedIn": true
}
*/
```

3. Converting a JSON String to a JavaScript Object (`JSON.parse()`)

This method takes a JSON string and converts it into a JavaScript object. This process is called deserialization. You use this method when you receive data from an API or read it from a file.

Syntax:

`JSON.parse(text, reviver)`

text:

The JSON string to parse.

- reviver (optional): A function that can transform the parsed values.

Example:

```
const jsonString = '{"name":"Bob","age":30,"isLoggedIn":true}';

// Convert to a JavaScript object
const userObject = JSON.parse(jsonString);
console.log(userObject);
// Output: { name: 'Bob', age: 30, isLoggedIn: true }

// You can now access its properties
console.log(userObject.name); // Output: Bob
```

Key Takeaways:

- JSON is a string format, not a JavaScript object. It is used to represent data for transport.
- Use `JSON.stringify()` to convert a JavaScript object into a JSON string.
- Use `JSON.parse()` to convert a JSON string back into a JavaScript object.
- These two methods are fundamental for working with web APIs, local storage, and server-side communication.

Fetch API

What is an API?

An API (Application Programming Interface) is a set of rules that allows two different applications to communicate with each other. A common analogy is a waiter at a restaurant.

- You (your application) want something (a joke).
- The waiter (the API) takes your request to the kitchen (the server where the jokes are stored).
- The waiter brings back what you asked for (the joke data).
- APIs are everywhere, from social media sites to weather services, and they are the foundation of most modern web applications.

The Fetch API

- The Fetch API is the standard, modern way to make network requests in the browser. It's built on Promises, which you learned about on Day 20 and Day 21.
- When you use `fetch()`, it immediately returns a `Promise`. This Promise will resolve to a Response object once the data is received from the server. This is a two-step process:

- `Request & Response`: The initial `fetch()` call returns a Promise that resolves with a Response object. This object contains information about the response (like its status code) but not the actual data itself.
- Parsing the Data: The data often comes in a format called JSON (JavaScript Object Notation). To turn the JSON into a usable JavaScript object, we need to call the `response.json()` method. This method also returns a Promise, which will resolve with the final, parsed data.

This is why you'll often see a chain of `.then()` calls when using the Fetch API: one to handle the initial response and one to handle the parsed data.

Key Takeaways ✨

- APIs are Your Gateway: APIs are how your application communicates with external services on the web. They are a fundamental building block of dynamic web development.
- Asynchronous Operations: `fetch()` is an asynchronous operation. This means your code doesn't wait for the data to arrive; it continues running while the request is being made in the background. The Promise is the mechanism that manages this.
- The Promise Chain: The `.then()`, `.catch()`, and `.finally()` methods are the standard way to handle the different outcomes of a Promise. They allow you to define what happens when an asynchronous task succeeds, fails, or completes.
- JSON is the Language: Most APIs communicate using JSON (JavaScript Object Notation), which looks a lot like a JavaScript object literal. The `response.json()` method is essential for converting this data into a usable JavaScript object.
- Robust Error Handling: Always plan for failure! The .catch() block is your safety net, ensuring your application remains stable and provides useful feedback to the user, even when things go wrong.

Promises

🗐 Core Concepts Covered:

1. What is a Promise?

A Promise is an object representing the eventual completion or failure of an asynchronous operation. Think of it like a real-life promise: you make a promise to someone. It can either be fulfilled (you keep the promise) or rejected (you break the promise). The person receiving the promise doesn't have to wait for you to act, they can go about their day, and you'll let them know the outcome later.

2. Promise States

- A Promise can be in one of three states:
- pending: The initial state. The asynchronous operation is still in progress.
- fulfilled: The operation completed successfully. The Promise is "resolved" with a resulting value.

- rejected: The operation failed. The Promise is "rejected" with a reason for the failure (an error object).
- A Promise is settled when it is either fulfilled or rejected.

3. Creating a Promise

- You create a new Promise using the Promise constructor. It takes a single argument: a function called the executor. The executor function itself takes two arguments: resolve and reject.
- resolve: A function that you call when the async operation is successful. You pass the resulting value to it.
- reject: A function that you call when the async operation fails. You pass an error or reason for the failure to it.

Example:

```
const myPromise = new Promise((resolve, reject) => {
    // Simulate an async operation (e.g., a network request)
    setTimeout(() => {
        const success = true;
        if (success) {
            resolve('Data fetched successfully!');
        } else {
            reject('Error: Failed to fetch data.');
        }
    }, 2000);
});
```

4. Consuming a Promise

- The real power of Promises comes from consuming them. You use the following methods to handle the different states of a settled Promise.
- `.then(onFulfilled, onRejected)`: This method is called when the Promise is settled.
- onFulfilled: A function that runs if the Promise is fulfilled. It receives the resolved value.
- onRejected: An optional function that runs if the Promise is rejected. It receives the rejection reason.

- `.catch(onRejected)`: A more readable way to handle a rejected Promise. It's equivalent to calling .then(null, onRejected). You should always include a `.catch()` block to prevent unhandled promise rejections.

- `.finally(onFinally)`: A function that runs when the Promise is settled (either fulfilled or rejected). This is perfect for cleanup tasks, like hiding a loading spinner, regardless of the outcome.

Example:

```
myPromise
    .then(result => {
```

```
        console.log(result); // Runs on success
    })
    .catch(error => {
        console.error(error); // Runs on failure
    })
    .finally(() => {
        console.log('Promise has completed.'); // Always runs
    });
```

Key Takeaways :

- A Promise is an object for handling asynchronous operations, providing a cleaner alternative to callbacks.
- A Promise has three states: pending, fulfilled, and rejected.
- Use `.then()` for successful outcomes, `.catch()` for errors, and `.finally()` for cleanup.
- Promises allow for chaining asynchronous operations and handling multiple Promises in parallel with `Promise.all()`.

---

async/await

Core Concepts Covered:

1. What is async/await?

async/await is essentially syntactic sugar built on top of Promises. It allows you to write asynchronous code in a way that looks and behaves synchronously. This makes the code much easier to read, write, and debug, and helps avoid the `.then().then()` promise chain.

2. The async Keyword

- The async keyword is used to declare an asynchronous function.
- An async function always returns a Promise.
- The value that the function returns will be the resolved value of the Promise.
- If the function throws an error, the Promise will be rejected with that error.

Example:

```
async function sayHello() {
    return 'Hello, World!';
}
```

```
// sayHello() returns a Promise, which we can consume with .then()
sayHello().then(message => console.log(message)); // Output: Hello, World!
```

3. The await Keyword

- The await keyword is used inside an async function to pause the execution of the function until a Promise is settled (either fulfilled or rejected).
- When the Promise resolves, await returns the resolved value.
- You can only use await inside a function declared with async.

Example:

```
function getPromise() {
    return new Promise(resolve => {
        setTimeout(() => resolve('Promise resolved!'), 2000);
    });
}

async function handlePromise() {
    console.log('Waiting for the promise...');
    // Execution pauses here until the promise resolves
    const result = await getPromise();
    console.log(result); // Output: Promise resolved!
    console.log('Function continues...');
}

handlePromise();
```

4. Error Handling with async/await (`try...catch`)

Since an await call can throw an error (if the Promise is rejected), the best practice for handling errors is to wrap your await calls in a standard `try...catch` block. This is much cleaner than using a `.catch()` method on every promise.

Example:

```
async function fetchData() {
    try {
        const response = await fetch('https://invalid-url.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('An error occurred:', error);
    }
}

fetchData();
```

Key Takeaways :

- `async/await` is syntactic sugar for Promises that makes asynchronous code more readable.
- An async function always returns a Promise.
- Use await inside an async function to wait for a Promise to resolve.
- Wrap your await calls in a `try...catch` block to handle errors gracefully.

Sending Data with Fetch API

The HTTP Request Methods

- These are the verbs we use to interact with a server's API.
- **GET:** Retrieve data from a server. (You've already done this!)
- **POST:** Create a new resource on the server. For our project, this will be used to add a new task to our list. A POST request typically includes a body with the data you want to send.
- **PUT:** Update an existing resource. This would be used to modify a task, like marking it as complete.
- **DELETE:** Delete a resource. In our project, this will be used to remove a task from the list.

fetch() with Options

To send data, we need to pass a second argument to the `fetch()` function: an options object. This object specifies the HTTP method, headers, and the body of the request.

```
fetch(url, {
  method: "POST", // or 'PUT', 'DELETE'
  headers: {
    "Content-Type": "application/json", // Tells the server we're sending JSON
  },
  body: JSON.stringify({ title: "New Task" }), // The data, converted to a JSON
string
});
```

Practice Set :

1. Make a TO DO App That include these functions:

- *Read (GET)*: The application fetches a hardcoded list of initial tasks when the page first loads.
- *Create (POST)*: A user can add a new task, which is then added to the tasks array after a simulated delay.

- *Update (PUT)*: A user can mark a task as complete by checking a checkbox or edit the task's title. Both actions trigger a simulated update to the task object in the tasks array.
- *Delete (DELETE)*: A user can remove a task from the list by clicking the delete button, which removes the item from the tasks array after a simulated delay.

Solution

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-Do List with API</title>
    <!-- Tailwind CSS for styling -->
    <script src="https://cdn.tailwindcss.com"></script>
    <style>
        @import url('https://fonts.googleapis.com/css2?
family=Inter:wght@400;600;700&display=swap');
        body {
            font-family: 'Inter', sans-serif;
            background-color: #f3f4f6;
        }
        .container {
            background: white;
            box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
        }
        .loading::after {
            content: '...';
            animation: dot-loading 1s linear infinite;
        }
        @keyframes dot-loading {
            0%   { content: ''; }
            33%  { content: '.'; }
            66%  { content: '..'; }
            100% { content: '...'; }
        }
    </style>
</head>
<body class="flex items-center justify-center min-h-screen p-4">

    <div class="container w-full max-w-lg p-8 rounded-2xl">

        <h1 class="text-3xl font-bold text-gray-800 mb-6 text-center">To-Do List
(Simulated API)</h1>

        <!-- Form to add a new task (Simulated POST request) -->
        <form id="addTaskForm" class="flex gap-4 mb-6">
            <input
                id="taskInput"
                type="text"
```

```
                placeholder="Add a new task..."
                class="flex-grow p-3 rounded-lg border border-gray-300
focus:outline-none focus:ring-2 focus:ring-blue-500 transition-all"
            >
            <button
                type="submit"
                id="addTaskBtn"
                class="bg-blue-600 hover:bg-blue-700 text-white font-semibold py-3
px-6 rounded-full transition-colors duration-200"
            >
                Add Task
            </button>
        </form>

        <!-- To-Do List Display Area -->
        <div class="flex items-center justify-between mb-4">
            <h2 class="text-xl font-semibold text-gray-700">Tasks</h2>
            <span id="loading-status" class="text-gray-500 hidden">Loading...
</span>
        </div>

        <ul id="taskList" class="space-y-3">
            <!-- Tasks will be rendered here -->
        </ul>
    </div>

    <script>
        // --- Data to simulate our backend ---
        // This array will act as our "database"
        let tasks = [];

        // --- DOM Elements ---
        const addTaskForm = document.getElementById('addTaskForm');
        const taskInput = document.getElementById('taskInput');
        const taskList = document.getElementById('taskList');
        const loadingStatus = document.getElementById('loading-status');
        const addTaskBtn = document.getElementById('addTaskBtn');

        // --- Helper Function to render tasks ---
        function renderTasks() {
            taskList.innerHTML = tasks.map(task => `
                <li class="flex items-center justify-between p-4 bg-gray-100
rounded-lg">
                    <div class="flex items-center gap-3">
                        <input
                            type="checkbox"
                            data-task-id="${task.id}"
                            class="toggle-complete h-5 w-5 text-blue-600 rounded
focus:ring-blue-500"
                            ${task.completed ? 'checked' : ''}
                        >
                        <span class="${task.completed ? 'line-through text-gray-
500' : 'text-gray-800'}">
                            ${task.title}
```

```
                        </span>
                    </div>
                    <div class="flex gap-2">
                        <!-- Edit button for a simulated PUT request -->
                        <button
                            data-task-id="${task.id}"
                            class="edit-btn text-blue-500 hover:text-blue-700
transition-colors duration-200 font-bold"
                        >
                            Edit
                        </button>
                        <!-- Delete button for a simulated DELETE request -->
                        <button
                            data-task-id="${task.id}"
                            class="delete-btn text-red-500 hover:text-red-700
transition-colors duration-200 font-bold"
                        >
                            &times;
                        </button>
                    </div>
                </li>
            `).join('');
        }

        // --- Simulated API Calls ---

        /**
         * Simulates a GET request to fetch all tasks. (Solution to Practice #3)
         */
        function getTasks() {
            return new Promise(resolve => {
                setTimeout(() => {
                    // This is our hardcoded "backend" data
                    const fetchedTasks = [
                        { id: 1, title: 'Learn Fetch API', completed: false },
                        { id: 2, title: 'Build a To-Do List', completed: false }
                    ];
                    tasks = fetchedTasks;
                    resolve();
                }, 500);
            });
        }

        /**
         * Simulates a POST request to add a new task to the server.
         * @param {string} title - The title of the new task.
         */
        function postTask(title) {
            return new Promise(resolve => {
                setTimeout(() => {
                    const newTask = {
                        id: Date.now(),
                        title: title,
                        completed: false
```

```
                    };
                    tasks.push(newTask);
                    resolve();
                }, 1000);
            });
        }

        /**
         * Simulates a PUT request to update an existing task. (Solution to
Practice #1 & #2)
         * @param {number} taskId - The ID of the task to update.
         * @param {object} updatedTask - The updated task object.
         */
        function putTask(taskId, updatedTask) {
            return new Promise(resolve => {
                setTimeout(() => {
                    const taskIndex = tasks.findIndex(task => task.id === taskId);
                    if (taskIndex !== -1) {
                        // Merge the existing task with the updated properties
                        tasks[taskIndex] = { ...tasks[taskIndex], ...updatedTask
};
                    }
                    resolve();
                }, 1000);
            });
        }

        /**
         * Simulates a DELETE request to remove a task from the server.
         * @param {number} taskId - The ID of the task to delete.
         */
        function deleteTask(taskId) {
            return new Promise(resolve => {
                setTimeout(() => {
                    tasks = tasks.filter(task => task.id !== taskId);
                    resolve();
                }, 1000);
            });
        }

        // --- Event Handlers ---

        /**
         * Handles the form submission to add a new task.
         */
        async function handleAddTask(event) {
            event.preventDefault();
            const taskTitle = taskInput.value.trim();

            if (taskTitle) {
                loadingStatus.classList.remove('hidden');
                addTaskBtn.disabled = true;
                await postTask(taskTitle);
                loadingStatus.classList.add('hidden');
```

```
                addTaskBtn.disabled = false;
                taskInput.value = '';
                renderTasks();
            }
        }

        /**
         * Handles the click on a delete or edit button, or a change on the
checkbox.
         */
        async function handleTaskAction(event) {
            const target = event.target;
            const taskId = parseInt(target.dataset.taskId);

            if (target.classList.contains('delete-btn')) {
                // Handle Delete (simulated DELETE request)
                loadingStatus.classList.remove('hidden');
                document.querySelectorAll('button').forEach(btn => btn.disabled =
true);

                await deleteTask(taskId);

                loadingStatus.classList.add('hidden');
                document.querySelectorAll('button').forEach(btn => btn.disabled =
false);
                renderTasks();
            } else if (target.classList.contains('toggle-complete')) {
                // Handle Toggle Complete (simulated PUT request)
                const task = tasks.find(t => t.id === taskId);
                if (task) {
                    loadingStatus.classList.remove('hidden');
                    document.querySelectorAll('button,
input[type="checkbox"]').forEach(el => el.disabled = true);

                    await putTask(taskId, { completed: !task.completed });

                    loadingStatus.classList.add('hidden');
                    document.querySelectorAll('button,
input[type="checkbox"]').forEach(el => el.disabled = false);
                    renderTasks();
                }
            } else if (target.classList.contains('edit-btn')) {
                // Handle Edit (simulated PUT request)
                const task = tasks.find(t => t.id === taskId);
                const newTitle = prompt('Enter new task title:', task.title);

                if (newTitle && newTitle.trim() !== task.title) {
                    loadingStatus.classList.remove('hidden');
                    document.querySelectorAll('button').forEach(btn =>
btn.disabled = true);

                    await putTask(taskId, { title: newTitle.trim() });

                    loadingStatus.classList.add('hidden');
```

```
                    document.querySelectorAll('button').forEach(btn =>
btn.disabled = false);
                    renderTasks();
                }
            }
        }

        // --- Event Listeners ---
        addTaskForm.addEventListener('submit', handleAddTask);
        taskList.addEventListener('click', handleTaskAction);
        taskList.addEventListener('change', handleTaskAction);

        // --- Initial Call ---
        // Simulates a GET request to fetch initial tasks on page load.
        async function init() {
            loadingStatus.classList.remove('hidden');
            await getTasks();
            loadingStatus.classList.add('hidden');
            renderTasks();
        }

        init();
    </script>
</body>
</html>
```

Key Takeaways: What You've Learned ✨

- **CRUD Operations:** You've now learned about the four fundamental operations for working with data: `Create (POST)`, `Read (GET)`, `Update (PUT)`, and `Delete (DELETE)`. This is known as CRUD.
- **Sending Data:** You now know how to send data to a server using the `fetch()` API's options object, including the method, headers, and body.
- **Asynchronous State Management:** You've seen how to handle asynchronous data manipulation by showing a loading state and re-rendering the UI once the "API call" is complete. This is the core pattern for all modern web applications.

JavaScript Forms & Input Validation

📚 Core Concepts Covered:

1. HTML Forms Basics

HTML forms are used to collect user input. They consist of various input elements (<input>, <select>, <textarea>, <textarea>, <button>, <select>, <option>, <button>) wrapped within a <form> tag.

`<form> tag:` The container for all form elements.

- **action:** Specifies where to send the form data when the form is submitted (usually a server-side script).
- ** method:** Specifies the HTTP method to use (e.g., GET, POST).

Input Types:

- **text:** Single-line text input.
- **password:** Text input where characters are masked.
- **email**: For email addresses; browsers may provide basic validation.
- **number**: For numeric input; browsers may provide up/down arrows.
- **checkbox:** For selecting zero or more options.
- **radio:** For selecting exactly one option from a group.
- **submit:** A button that submits the form.
- **button:** A generic button (requires JavaScript for functionality).
- **name attribute:** Crucial for sending data to the server (JavaScript also uses it).
- **id attribute:** Used by JavaScript (and CSS) to uniquely identify elements.
- **value attribute**: The initial value for an input, or the value sent for radio/checkboxes.

2. Accessing Form Elements with JavaScript

You can access form elements just like any other DOM element.

- **By ID:** document.getElementById('inputId') (most common and reliable).
- **By Name**: document.getElementsByName('inputName') (returns a NodeList).
- **By Query Selector:**_ document.querySelector('#inputId') or document.querySelector('form input[name="username"]').

Getting/Setting Input Values:

`inputElement.value:` Gets or sets the current value of text, password, email, number inputs, or the selected option of a select.

`checkboxElement.checked:` true if checked, false otherwise.

`radioElement.checked:` true if selected, false otherwise.

3. Form Submission Event

The most important event for forms is the `submit` event.

```
formElement.addEventListener('submit', function(event) { ... })
```

`event.preventDefault()`: This is crucial for client-side validation. By default, submitting a form reloads the page. event.preventDefault() stops this default behavior, allowing your JavaScript to handle the submission (e.g., validate data, send it via AJAX).

4. Client-Side Input Validation Client-side validation occurs in the user's browser before data is sent to the server. It provides immediate feedback to the user and reduces unnecessary server requests.

Why Client-Side Validation?

- **User Experience:** Instant feedback, better usability.
- **Reduced Server Load:** Invalid data isn't sent to the server.
- **Limitations:** Client-side validation can be bypassed (e.g., by disabling JavaScript). Always perform server-side validation as well for security and data integrity.

Common Validation Checks:

- **Required Fields:** Check if an input is empty.
- **Min/Max Length:** For text inputs (e.g., passwords, usernames).
- **Format/Pattern:** Using Regular Expressions (RegEx) for emails, phone numbers, passwords (e.g., requiring uppercase, numbers, special characters).
- **Numeric Range:** For number inputs.
- **Matching Fields:** E.g., "Confirm Password" must match "Password."
- **Checkbox/Radio Selection:** Ensuring at least one option is selected.

5. Providing User Feedback

- Clear and immediate feedback is essential for a good user experience.
- Error Messages: Display specific messages next to the invalid input field.
- Styling: Change borders, background colors, or add icons to highlight invalid fields.
- Success Messages: Confirm successful submission or validation.

6. Regular Expressions (RegEx) for Validation (Basic Introduction)

Regular Expressions are powerful patterns used to match character combinations in strings. They are invaluable for validating input formats.

- **Syntax:**

`/pattern/flags`

Common Patterns:

`\d`: Matches any digit (0-9).

`\w`: Matches any word character (alphanumeric + underscore).

`+`: One or more occurrences.

`*`: Zero or more occurrences.

`?`: Zero or one occurrence.

`{n}`: Exactly n occurrences.

`{n,m}`: Between n and m occurrences.

`[abc]`: Matches a, b, or c.

`[^abc]`: Matches anything not a, b, or c.

`^`: Start of string.

`$`: End of string.

`string.match(regex)`: Returns an array of matches or null.

`regex.test(string)`: Returns true if the string matches the pattern, false otherwise (most common for validation).

Example (Email Validation):

```
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
console.log(emailRegex.test("test@example.com")); // true
console.log(emailRegex.test("invalid-email")); // false
```

🚀 Project : Simple Registration Form with Validation

Today's project is a functional Registration Form that incorporates client-side input validation. This will allow you to apply the concepts learned about forms, events, and validation to a practical scenario.

Features:

- Fields for Username, Email, Password, Confirm Password.
- Validation Rules:
- All fields are required.
- Username: Minimum 3 characters.
- Email: Valid email format.
- Password: Minimum 6 characters, must contain at least one uppercase letter, one lowercase letter, one number, and one special character.
- Confirm Password: Must match the Password field.
- Real-time feedback (error messages) for each field.

- A success message upon valid submission.

(This project's code is located in index.html and index.js within this Day10 folder.)

💻 Project Logic Breakdown

The js file handles all the client-side validation logic for the registration form.

1. DOM Element References

```
const registrationForm = document.getElementById("registrationForm");
const usernameInput = document.getElementById("username");
const emailInput = document.getElementById("email");
const passwordInput = document.getElementById("password");
const confirmPasswordInput = document.getElementById("confirmPassword");
const successMessage = document.getElementById("successMessage");
// ... and error message spans for each input
```

These variables provide access to the form itself, all input fields, and the elements where validation error messages will be displayed.

2. Helper Function:

showError(input, message) and showSuccess(input)

```
function showError(input, message) {
  const formControl = input.parentElement; // Get the parent .form-control div
  formControl.classList.remove("success");
  formControl.classList.add("error");
  const small = formControl.querySelector("small"); // Get the small tag for error
message
  small.textContent = message;
}

function showSuccess(input) {
  const formControl = input.parentElement;
  formControl.classList.remove("error");
  formControl.classList.add("success");
  const small = formControl.querySelector("small");
  small.textContent = ""; // Clear error message
}
```

These `functions` are crucial for providing visual feedback. They add/remove error or success CSS classes to the parent div (.form-control) of an input, which then applies styling (e.g., red/green borders). They also update the text content of the <small> tag for the specific error message.

3. Validation Functions (Per Field)

- Each input field has a dedicated validation function:

- checkRequired(inputArray):
- Takes an array of input elements.
- Iterates through each, checks if input.value.trim() is empty.
- `Calls showError()` or showSuccess() accordingly.
- Returns false if any required field is empty.
- `checkLength(input, min, max):`
- Checks if `input.value.length` is within the specified min and max range.
- Calls `showError()` if length is invalid.
- `checkEmail(input):`
- Uses a Regular Expression (/^[^\s@]+@[^\s@]+.[^\s@]+$/) to test if the email format is valid.
- Calls `showError()` if the format is invalid.
- `checkPassword(input):`
- Uses a more complex Regular Expression (/^(?=.*[a-z])(?=.[A-Z])(?=.\d)(?=.[!@#$%^&*()_+-=[]{};':"\|,.<>/?]).{6,}$/) to ensure the password meets complexity requirements (min 6 chars, uppercase, lowercase, number, special char).
- Calls showError() if the password is weak.
- `checkPasswordsMatch(passwordInput, confirmPasswordInput):`
- Compares the value of the password and confirm password fields.
- Calls `showError()` if they don't match.

4. `validateForm()` **Function (Main Validation Logic)**

```
function validateForm() {
  let isValid = true; // Flag to track overall form validity

  // Perform all individual checks
  isValid =
    checkRequired([
      usernameInput,
      emailInput,
      passwordInput,
      confirmPasswordInput,
    ]) && isValid;
  isValid = checkLength(usernameInput, 3, 15) && isValid;
  isValid = checkEmail(emailInput) && isValid;
  isValid = checkPassword(passwordInput) && isValid;
  isValid = checkPasswordsMatch(passwordInput, confirmPasswordInput) && isValid;
```

```
  return isValid;
}
```

This function orchestrates all the individual validation checks.

It uses a isValid flag, which is initially true. Each validation function updates this flag using && (logical AND) to ensure that if any check fails, isValid becomes false and stays false.

It returns the final isValid `boolean.`

5. Form Submission Event Listener

```
registrationForm.addEventListener("submit", (event) => {
  event.preventDefault(); // Prevent default form submission (page reload)

  if (validateForm()) {
    // If all validations pass
    successMessage.classList.remove("hidden"); // Show success message
    registrationForm.reset(); // Clear the form fields
    // Optionally, hide success message after a few seconds
    setTimeout(() => {
      successMessage.classList.add("hidden");
      // Reset all input styles back to default (no success/error)
      document.querySelectorAll(".form-control").forEach((control) => {
        control.classList.remove("success", "error");
        control.querySelector("small").textContent = "";
      });
    }, 5000);
    console.log("Form submitted successfully!");
  } else {
    successMessage.classList.add("hidden"); // Hide success message if validation
fails
    console.log("Form validation failed.");
  }
});
```

`event.preventDefault()`: This is the first and most important step. It stops the browser from reloading the page when the form is submitted, allowing JavaScript to take full control of the validation process.

- It calls validateForm().
- If validateForm() returns true (all valid):
- The successMessage is displayed.
- `registrationForm.reset()` clears all input fields.
- A setTimeout hides the success message and resets input styles after 5 seconds.
- If `validateForm()` returns false (validation failed):

- The successMessage is hidden (if it was previously shown).
- Error messages are already displayed by the individual `showError()` calls.

6. Real-time Validation (Optional but Recommended)

While not explicitly in the provided index.js for simplicity (to focus on submit), in a real application, you'd often add input or blur event listeners to individual fields to provide real-time validation feedback as the user types or leaves a field.

// Example of real-time validation (add these listeners to your inputs)

```
usernameInput.addEventListener("blur", () => checkLength(usernameInput, 3, 15));
emailInput.addEventListener("input", () => checkEmail(emailInput));
passwordInput.addEventListener("input", () => checkPassword(passwordInput));
confirmPasswordInput.addEventListener("input", () =>
  checkPasswordsMatch(passwordInput, confirmPasswordInput)
);
```

Key Takeaways :

- HTML Forms are essential for user input; JavaScript allows dynamic control and validation.
- `event.preventDefault()` is crucial to stop default form submission and handle it with JavaScript.
- Client-side validation provides immediate feedback and improves UX, but always back it up with server-side validation.
- Regular Expressions are powerful tools for pattern-based input validation.
- Clear and immediate visual feedback (error messages, styling) is vital for guiding users.

Modules (ES Modules) - Organizing Your Code

What are ES Modules?

ES Modules (or ECMAScript Modules) are the official, standardized way to use modules in JavaScript. They introduce two simple keywords: export and import.

- *export*: This keyword is used to make variables, functions, or classes available to other files.
- *import:* This keyword is used to bring those exported items into the current file.

The primary benefit of using modules is that they create a local scope for each file, preventing the dreaded "global scope pollution" where variables from one file might accidentally overwrite variables from another.

Key Concepts

1. Exporting

You can export items in two main ways:

- *Named Exports:* Export multiple items from a single file by adding the export keyword before their declaration.

```
// file: math.js
export const pi = 3.14;

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

- *Default Exports:* You can have only one default export per module. This is often used for the main functionality of a file.

```
// file: utils.js
const defaultMessage = "Hello from the default export!";
export default defaultMessage;
```

2. Importing

- When importing, the syntax depends on how the item was exported.
- Importing Named Exports: You must use the same names as the exports, wrapped in curly braces.

```
// file: main.js
import { pi, add } from './math.js';

console.log(pi); // 3.14
console.log(add(5, 3)); // 8
```

- Importing Default Exports: You can give the imported item any name you want.

```
// file: main.js
import greeting from './utils.js';
console.log(greeting); // "Hello from the default export!"
```

- *Important:* When using ES Modules in the browser, you must add type="module" to your script tag: <script type="module" src="index.js">.

Key Takeaways ✨

- Organization: Modules keep your code clean by separating different functionalities into their own files.

- No Global Scope Pollution: Variables in a module are local to that module unless explicitly exported.
- Reusability: You can reuse the same module in many different parts of your application.
- Syntax: Use export to share code and import to use it. Remember to include type="module" in your script tag when linking to the main module file.

Introduction to Classes (OOP in JavaScript)

- A class is a blueprint or a template for creating objects. Think of a class like a cookie cutter and the objects you create from it as the individual cookies. They all share the same shape (properties and methods) but can have unique details (the data they hold).

Key Concepts 💡

1. The class keyword

The class keyword is used to declare a new class. It's a syntactic sugar over JavaScript's prototype-based inheritance, making it much easier to read and work with.

```
class ClassName {
  // class body
}
```

2. The constructor method

The constructor is a special method that is automatically called when a new object is created from the class. Its purpose is to initialize the object's properties. It's where you'll typically set the initial values for your object's data.

3. Properties

Properties are the data or attributes associated with a class. They are defined inside the constructor and are set using the this keyword. For our User class, properties will include name, email, and isLoggedIn.

4. Methods

Methods are the functions or behaviors that an object can perform. They are defined inside the class body, outside of the constructor. For our User class, methods will include login() and `logout()`.

5. The this keyword

The this keyword is a crucial concept in classes. Inside a method, this refers to the specific instance of the object on which the method was called. It allows you to access and modify the properties of that particular object.

Key Takeaways: What You've Learned ✨

- Classes as Blueprints: You now understand that a class is a blueprint for creating objects.

- Encapsulation: Classes allow you to group related data (properties) and behavior (methods) into a single, logical unit.
- Object Instantiation: The new keyword is used to create a new, distinct object from a class.
- this and Scope: You've seen how this is used to refer to the specific object instance within a class.

Inheritance (OOP in JavaScript)

Inheritance

is a way to create a new class that is a modified version of an existing class. The new class is called a child class (or subclass), and the existing class is the parent class (or superclass). The child class automatically gets all the properties and methods of the parent, and you can then add new, specialized features or override existing ones.

Think of a Vehicle class. A Car is a type of Vehicle. It has all the properties of a vehicle (like speed and color) but might have its own unique properties (like numberOfDoors) and methods (like honk()).

Key Concepts 💡

1. The extends keyword

The `extends` keyword is used to create a child class. It tells JavaScript that the new class will inherit from a parent class.

```
class ChildClass extends ParentClass {
  // class body
}
```

2. The super() keyword

The `super()` keyword is used inside the child class's constructor. It's a special function call that executes the parent class's constructor. You must call super() before you can use the this keyword in the child class's constructor.

3. Method Overriding

A child class can define a method with the same name as a method in its parent class. When you call that method on an instance of the child class, the child's version of the method will be executed instead of the parent's. This is called method overriding.

Key Takeaways: What You've Learned

- Code Reusability: Inheritance allows you to reuse common properties and methods from a parent class, avoiding redundant code.

- Creating a Hierarchy: It provides a clear, logical structure to your code, representing "is a" relationships (e.g., a Circle is a Shape).
- Specialization: Child classes can specialize their behavior by adding new features or overriding existing methods.

Composition (OOP in JavaScript)

What Is Composition?

Composition is a fundamental principle of OOP where a more complex object is created by combining simpler, existing objects. Instead of a child class extending a parent class, a parent class contains instances of other classes as its properties. This allows you to build complex systems from smaller, manageable parts.

Our project will create a Library class that has a collection of Book objects. The Library will handle all interactions with its book collection, demonstrating how to use different objects together to create a functional system.

Key Concepts 💡

1. Composition

Composition describes a "has-a" relationship. For example, a Car has a Engine and a Wheel, but a Car is not a type of Engine. Similarly, our Library has a list of Book objects. This approach often leads to more flexible and reusable code than inheritance.

2. Class Interaction

In this project, you'll see how methods of one class (Library) can take instances of another class (Book) as arguments or return them. This shows how objects can communicate and work together to perform a task.

3. Encapsulation

The Library class encapsulates the logic for managing books. A user of the Library class doesn't need to know how it stores the books (e.g., in an array). They only need to use its public methods like `addBook()`, `borrowBook()`, and `returnBook()`.

Key Takeaways: What You've Learned

- "Has-a" vs. "Is-a": You now understand the difference between composition ("has-a") and inheritance ("is-a") and when to use each.
- Building Systems: You can combine simple, focused classes to create a more powerful and complex system.
- Modular Design: By separating the Book and Library logic, your code becomes more organized and easier to maintain.

Most Important Javascript Concepts

Closures

A closure is a function that remembers the environment (or scope) in which it was created. It has access to the variables of its outer function even after that outer function has finished executing.

Think of it like a backpack a function takes with it. It can go anywhere, but it always has access to the items (variables) it packed when it was first created. This allows you to create private data and functions, as the variables aren't accessible from the outside.

```
function createCounter() {
  let count = 0; // 'count' is a private variable

  return function() {
    count++; // This inner function "remembers" and can modify 'count'
    console.log(count);
  };
}

const counter = createCounter();
counter(); // 1
counter(); // 2
```

Here, createCounter runs and returns a new function. Even though createCounter is done, the counter function still has access to count. This is a closure in action.

The Event Loop

The Event Loop is the mechanism that allows JavaScript to handle asynchronous operations without blocking the main execution thread.

JavaScript is single-threaded, so it can only do one thing at a time. The `Event Loop` ensures that long-running tasks, like network requests or timers, don't freeze the entire application.

The key components are:

- Call Stack: Where synchronous code (like function calls) is executed.
- Web APIs: Provided by the browser, they handle asynchronous tasks (e.g., setTimeout, fetch, DOM events).
- Callback Queue: Holds functions that are ready to be executed after a Web API task is complete.
- The Event Loop's job is to constantly check if the Call Stack is empty. If it is, it takes the first function from the Callback Queue and pushes it onto the Call Stack for execution.

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

setTimeout(() => {
  console.log("Inside setTimeout");
}, 0); // This is moved to the Web APIs and then the Callback Queue

console.log("End");

// Output:
// Start
// End
// Inside setTimeout
```

Even with a 0-second delay, "Inside setTimeout" is logged last because the main synchronous code (Start and End) on the Call Stack must finish before the Event Loop can push the setTimeout callback onto it.

The Temporal Dead Zone (TDZ)

- The Temporal Dead Zone (TDZ) is the period of time during which let and const variables exist but cannot be accessed. 🚫 It begins at the start of the block scope and ends when the variable is declared. Accessing a variable in the TDZ results in a ReferenceError.
- The TDZ is a safety feature that prevents using a variable before it has been initialized. Variables declared with var are "hoisted" and initialized with undefined, so they don't have a TDZ.

```


console.log(myVar); // Output: undefined (var is hoisted)

// console.log(myLet); // ERROR: ReferenceError (TDZ for 'myLet')

var myVar = "I'm a var";
let myLet = "I'm a let";
```

Execution Context & Call Stack

An Execution Context is the environment in which JavaScript code runs.

- Every program starts in the Global Execution Context.
- When a function is called, a new execution context is created and pushed onto the call stack.

📌 Example:

```
function first() {
  console.log("First function");
  second();
}

function second() {
```

```
  console.log("Second function");
}

first();
```

Hoisting

Hoisting means moving variable and function declarations to the top of their scope during compilation.

- var → hoisted and initialized with undefined.
- let & const → hoisted but kept in TDZ until initialized.

📌 Example:

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

console.log(b); // ❌ ReferenceError
let b = 20;
```

this Keyword

The value of this depends on how and where the function is called.

- Global scope → window (browser).
- Normal function → undefined in strict mode, window otherwise.
- Object method → the object itself.
- Arrow function → inherits this from outer scope.

📌 Example:

```
const obj = {
  name: "Buddy",
  normalFunc: function() {
    console.log(this.name);
  },
  arrowFunc: () => {
    console.log(this.name);
  }
};

obj.normalFunc(); // "Buddy"
obj.arrowFunc();  // undefined
```

Prototypes & Inheritance

Every object in JS has a hidden [[Prototype]] property. This enables inheritance via the prototype chain.

📌 Example:

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

Person.prototype.greet = function() {
  console.log("Hello " + this.name);
};

const user = new Person("Buddy");
user.greet(); // Hello Buddy
```

Asynchronous JavaScript

JavaScript is single-threaded, but async tasks are handled using callbacks, promises, and async/await.

📌 Example with Promise:

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data received!"), 2000);
  });
}

fetchData().then(data => console.log(data));
```

📌 Example with async/await:

```
async function getData() {
  const data = await fetchData();
  console.log(data);
}

getData();
```

Event Loop & Concurrency Model

The event loop manages how sync and async code runs.

📌 Example:

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

setTimeout(() => console.log("Async Task"), 0);

console.log("End");


👉 Output:

Start
End
Async Task
```

ES6+ Modules

Modules let you split and reuse code across files.

📌 Example:

```
// math.js
export function add(a, b) {
  return a + b;
}

// app.js
import { add } from "./math.js";
console.log(add(5, 3)); // 8
```

🎯 Final Note

✔ Execution Context & Call Stack → controls code flow

✔ Hoisting & TDZ → variable behavior before initialization

✔ Closures → private state and memory

✔ Scope → variable accessibility rules

✔ this → depends on call site

✔ Prototype → inheritance model in JS

✔ Async JS → handles non-blocking tasks

✔ Event Loop → sync vs async execution

✔ Modules → clean & maintainable code structure

☑ Mastering these concepts will give you a rock-solid JavaScript foundation, making it much easier to dive into frameworks like React, Angular, or backend development with Node.js.

Javascript Cheat Sheet

1. Variables and Data Types

- **var:** Old way to declare variables. Function-scoped.
- **let:** Block-scoped variable. Can be reassigned.
- **const:** Block-scoped constant. Cannot be reassigned.

Primitive Data Types:

- **string:** Text, e.g., "Hello".
- **number:** Integers and floats, e.g., 10, 3.14.
- **boolean:** true or false.
- **null:** Intentional absence of any value.
- **undefined:** A variable that has been declared but not assigned a value.
- **symbol:** Unique and immutable primitive value.
- **bigint:** For very large integer values.

Complex Data Types:

- object: Key-value pairs, e.g., { name: "Alice", age: 30 }.
- array: A list of values, e.g., [1, 2, 3].

2. Operators

- **Arithmetic:**

```
 +, -, *, /, % (modulus), ** (exponentiation).
```

- **Assignment:**

```
+=, -=, *=, /=.
```

- **Comparison:**

```
== (loose equality), === (strict equality), !=, !==, >, <, >=, <=.
```

- **Logical:**

```
 && (AND), || (OR), ! (NOT).
```

- **Ternary:**

```
condition ? valueIfTrue : valueIfFalse;.
```

---

3. Functions

- **Function Declaration:**

```
function greet(name) {
  return `Hello, ${name}!`;
}
```

- **Function Expression:**

```
const greet = function (name) {
  return `Hello, ${name}!`;
};
```

- **Arrow Function (ES6):**

```
const greet = (name) => {
  return `Hello, ${name}!`;
};
// Concise syntax for a single return statement
const greet = (name) => `Hello, ${name}!`;
```

4. Control Flow

- **If/Else:**

```
if (score > 100) {
  // do something
} else if (score > 50) {
  // do something else
} else {
  // default case
}
```

- **Switch Statement:**

```
switch (day) {
  case "Monday":
    console.log("Start of the week");
    break;
  default:
    console.log("Some other day");
}
```

- **Loops:**

```
// for loop
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// while loop
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
// forEach (for arrays)
const numbers = [1, 2, 3];
numbers.forEach((num) => console.log(num));
```

5. Array Methods

`.push()`: Adds an element to the end of an array.

`.pop()`: Removes the last element.

`.shift()`: Removes the first element.

`.unshift()`: Adds an element to the beginning.

`.map()`: Creates a new array by applying a function to each element.

`.filter()`: Creates a new array with elements that pass a test.

`.reduce()`: Reduces an array to a single value.

6. Object-Oriented Programming (OOP)

- **Classes (ES6):**

```
class Car {
  constructor(make, model) {
```

```
    this.make = make;
    this.model = model;
  }
  drive() {
    console.log(`Driving the ${this.make} ${this.model}.`);
  }
}
const myCar = new Car("Toyota", "Camry");
myCar.drive(); // Output: Driving the Toyota Camry.
```

- **Inheritance:**

```
class ElectricCar extends Car {
  constructor(make, model, battery) {
    super(make, model); // Call the parent constructor
    this.battery = battery;
  }
}
```

7. Asynchronous JavaScript

- **Promises:**

Represents the eventual completion (or failure) of an asynchronous operation.

```
const myPromise = new Promise((resolve, reject) => {
  // ... async operation
  if (success) {
    resolve("Operation successful!");
  } else {
    reject("Operation failed.");
  }
});
myPromise
  .then((result) => console.log(result))
  .catch((error) => console.error(error));
```

- **async/await:**

Modern syntax for handling promises, making async code look synchronous.

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error);
```

```
  }
}
```

8. The Document Object Model (DOM)

- **Selecting Elements:**

```
const elementById = document.getElementById("my-id");
const elementsByClass = document.getElementsByClassName("my-class");
const firstElement = document.querySelector(".my-class");
const allElements = document.querySelectorAll(".my-class");
```

- **Manipulating Elements:**

```
element.textContent = "New Text"; // Change text content
element.innerHTML = "<strong>New HTML</strong>"; // Change inner HTML
element.style.color = "red"; // Change style
element.classList.add("new-class"); // Add a CSS class
```

- **Event Handling:**

```
button.addEventListener("click", () => {
  console.log("Button clicked!");
});
```

The possibilities are endless. I'm excited to see what you build next! 👍💪.