Introduction: Learning JavaScript by Doing

Welcome to "100 JavaScript Exercises," your hands-on guide to mastering the world's most popular programming language. The path to becoming a proficient developer isn't found in passively reading theory; it's paved with the tangible experience of writing code, solving problems, and seeing your work come to life. This book is built on that simple but powerful principle: **we learn best by doing**.

Whether you are taking your very first steps into programming or you're looking to solidify your foundational skills, this collection of exercises is designed for you. We will journey through the

core concepts of JavaScript, not as abstract ideas, but as practical tools to build real things.

What You Will Learn

This book is structured as a series of bite-sized, practical examples. You will start with the absolute basics and progressively build your skills, covering essential topics such as:

- **Variables & Data Types:** Learn how JavaScript stores and manages different kinds of information, from simple text and numbers to complex collections of data.
- **Control Flow:** Discover how to make decisions and repeat actions in your code, giving your applications logic and intelligence.
- **Functions:** Master the art of creating reusable blocks of code—the fundamental building blocks of any program.
- **Arrays & Objects:** Dive into JavaScript's most important data structures for organizing and working with collections of data.
- **DOM Manipulation & Events:** Explore how JavaScript interacts with HTML to create dynamic, interactive web pages that respond to user actions like clicks and keyboard input.
- **Asynchronous JavaScript:** Understand how to handle tasks that don't happen instantly, such as fetching data from a server or waiting for a timer to finish.

Each of the 100 exercises includes a clear explanation, a practical HTML and JavaScript code sample, and a breakdown of how it all works. By the time you complete this collection, you won't just *know* JavaScript—you'll have the confidence and experience that comes from having built with it, one exercise at a time.

Let's start building.

Part 1: Variables, Data Types & Operators

This section covers the absolute basics: how to store information and perform basic operations.

1. Declaring a Variable with var

This shows the traditional way to declare a variable.

```
<!DOCTYPE html>
<html>
<head>
  <title>Var Example</title>
</head>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
var greeting = "Hello, World!";
document.getElementById("output").textContent = greeting;
```

Explanation
var greeting = "Hello, World!"; declares a variable named greeting and assigns it the string value "Hello, World!". var is function-scoped, meaning it's accessible throughout the function it's declared in. The second line finds the paragraph with the ID output and sets its text content to the value of our greeting variable.

2. Declaring a Variable with let

let is the modern, block-scoped way to declare a variable that can be changed.

```
<!DOCTYPE html>
<html>
<head>
  <title>Let Example</title>
</head>
<body>
  <p>Original Value: <span id="original"></span></p>
  <p>New Value: <span id="new"></span></p>
  <script src="script.js"></script>
```

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

JavaScript (script.js)

```
JavaScript
let score = 100;
document.getElementById("original").textContent = score;

score = 150; // Re-assigning the value
document.getElementById("new").textContent = score;
```

Explanation
We declare score with let, allowing its value to be changed later. It is block-scoped, meaning it is only available within the block (e.g., inside {...}) where it is defined. We first assign 100 and display it, then update the value to 150 and display the new value.

3. Declaring a Constant with const

Use const for variables whose values should never change.

```
<!DOCTYPE html>
<html>
<head>
  <title>Const Example</title>
</head>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const pi = 3.14159;
// pi = 3.14; // This line would cause an error!
document.getElementById("output").textContent = "The value of PI is approximately: " + pi;
```

Explanation
const pi declares a constant variable named pi. Once assigned, its value cannot be reassigned. Attempting to change it (like in the commented-out line) will result in a TypeError. This is useful for values that are meant to be permanent throughout your script, like mathematical constants or configuration settings.

4. String Data Type

Textual data is stored as a string.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let firstName = "John";
let lastName = 'Doe'; // You can use single or double quotes
let fullName = firstName + " " + lastName; // Concatenation
document.getElementById("output").textContent = fullName;
```

Explanation
Strings are sequences of characters enclosed in single (') or double (") quotes. Here, we create two strings, firstName and lastName. We then combine them (a process called concatenation) with a space in between to create a fullName.

5. Number Data Type

Numbers can be integers or decimals.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let integer = 25;
let decimal = 10.5;
let sum = integer + decimal;
document.getElementById("output").textContent = "The sum is: " + sum;
```

Explanation
The Number type represents both integer (whole numbers) and floating-point (decimal) numbers. JavaScript doesn't distinguish between them. We simply declare two number

variables and add them together.

6. Boolean Data Type

Represents logical values: true or false.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let isStudent = true;
let isWorking = false;
document.getElementById("output").textContent = "Is the person a student? " + isStudent;
```

Explanation
A boolean can only have one of two values: true or false. They are essential for conditional logic, such as in if statements, to control the flow of your program.

7. Undefined Data Type

A variable that has been declared but not yet assigned a value has the value undefined.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let age; // Declared but not initialized
document.getElementById("output").textContent = "The value of age is: " + age;
```

Explanation
When you declare a variable with let or var but don't give it a value, JavaScript automatically assigns it the value undefined. This indicates the absence of a value.

8. Null Data Type

null is an intentional absence of any object value.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let user = null; // Intentionally set to nothing
document.getElementById("output").textContent = "The value of user is: " + user;
```

Explanation
null represents the intentional absence of a value. It's different from undefined because a developer explicitly assigns null, whereas undefined is automatic. You might set a variable to null to indicate that an object could be there, but currently isn't.

9. Object Data Type

A collection of key-value pairs.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const person = {
  firstName: "Jane",
  lastName: "Doe",
  age: 30
};
document.getElementById("output").textContent = person.firstName + " is " + person.age + "
years old.";
```

Explanation

An object is a complex data type that allows you to store related data in a key: value format. The keys (firstName, lastName, age) are strings, and the values can be any data type. You access values using dot notation (e.g., person.firstName).

10. Array Data Type

A special type of object for storing ordered lists of values.

```
<!DOCTYPE html>
<html>
<body>
 <p id="output"></p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const colors = ["Red", "Green", "Blue"];
document.getElementById("output").textContent = "The second color is: " + colors[1];
```

Explanation
An array is used to store a list of items. Array items are zero-indexed, meaning the first item is at index 0, the second at index 1, and so on. We access the second color, "Green", by using its index 1 inside square brackets [].

11. Arithmetic Operators

Perform mathematical calculations.

```
<!DOCTYPE html>
<html>
<body>
 <p id="add"></p>
 <p id="subtract"></p>
 <p id="multiply"></p>
 <p id="divide"></p>
 <p id="modulus"></p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let a = 10;
let b = 3;
```

```
document.getElementById("add").textContent = "Addition (10 + 3): " + (a + b);
document.getElementById("subtract").textContent = "Subtraction (10 - 3): " + (a - b);
document.getElementById("multiply").textContent = "Multiplication (10 * 3): " + (a * b);
document.getElementById("divide").textContent = "Division (10 / 3): " + (a / b);
document.getElementById("modulus").textContent = "Modulus (10 % 3): " + (a % b); // Remainder
```

Explanation

+: Addition

-: Subtraction

*: Multiplication

/: Division

%: Modulus (returns the remainder of a division)

12. Assignment Operators

Assign values to variables.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let x = 10;
x += 5; // Equivalent to x = x + 5
document.getElementById("output").textContent = "The value of x is: " + x;
```

Explanation
The += operator is a shorthand way to add a value to an existing variable and assign the result back to it. Other common assignment operators include -=, *=, and /=.

13. Comparison Operators (Strict Equality)

Compare two values for equality, without type coercion.

```
<!DOCTYPE html>
<html>
<body>
```

```
  <p id="output1"></p>
  <p id="output2"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let num = 5;
let str = "5";

document.getElementById("output1").textContent = "5 === '5' is " + (num === str); // false
document.getElementById("output2").textContent = "5 === 5 is " + (num === 5); // true
```

Explanation
The strict equality operator (===) returns true only if both the value and the data type of the operands are the same. Here, 5 (a number) is not the same type as "5" (a string), so the result is false.

14. Comparison Operators (Loose Equality)

Compare two values for equality, with type coercion.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let num = 5;
let str = "5";

document.getElementById("output").textContent = "5 == '5' is " + (num == str); // true
```

Explanation
The loose equality operator (==) attempts to convert the operands to the same type before comparing them (this is called type coercion). Because the string "5" can be converted to the number 5, the comparison returns true. It's generally better practice to use strict equality (===) to avoid unexpected results.

15. Logical AND (&&) Operator

Returns true if both operands are true.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let age = 25;
let hasLicense = true;

if (age >= 18 && hasLicense) {
  document.getElementById("output").textContent = "This person can drive.";
} else {
  document.getElementById("output").textContent = "This person cannot drive.";
}
```

Explanation
The && (AND) operator checks if the conditions on both its left and right sides are true. Since age is greater than 18 AND hasLicense is true, the code inside the if block is executed.

16. Logical OR (||) Operator

Returns true if at least one operand is true.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let isWeekend = true;
let isHoliday = false;

if (isWeekend || isHoliday) {
```

```
  document.getElementById("output").textContent = "Time to relax!";
} else {
  document.getElementById("output").textContent = "Time to work.";
}
```

Explanation
The || (OR) operator checks if at least one of its operands is true. Since isWeekend is true, the overall condition is met, and the first message is displayed, even though isHoliday is false.

17. Logical NOT (!) Operator

Inverts the boolean value of its operand.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)
JavaScript

```
let isLoggedIn = false;

if (!isLoggedIn) { // Checks if isLoggedIn is NOT true
  document.getElementById("output").textContent = "Please log in.";
}
```

Explanation
The ! (NOT) operator flips a boolean value. !false becomes true, and !true becomes false. In this code, !isLoggedIn evaluates to true, so the message inside the if statement is displayed.

18. Template Literals

An easier way to create strings with embedded variables.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const name = "Alex";
const userAge = 32;

const message = `Hello, my name is ${name} and I am ${userAge} years old.`;
document.getElementById("output").textContent = message;
```

Explanation
Template literals use backticks (`) instead of single or double quotes. They allow you to embed variables or expressions directly into the string using the ${...} syntax. This is much cleaner and more readable than traditional string concatenation with +.

19. typeof Operator

Returns a string indicating the type of a variable.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output1"></p>
  <p id="output2"></p>
  <p id="output3"></p>
  <p id="output4"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let a = "Hello";
let b = 42;
let c = true;
let d = { key: "value" };

document.getElementById("output1").textContent = `Type of a is: ${typeof a}`;
document.getElementById("output2").textContent = `Type of b is: ${typeof b}`;
document.getElementById("output3").textContent = `Type of c is: ${typeof c}`;
document.getElementById("output4").textContent = `Type of d is: ${typeof d}`;
```

Explanation
The typeof operator is useful for checking what kind of data is stored in a variable. It returns a string representing the data type, such as "string", "number", "boolean", or "object".

20. Using prompt to Get User Input

Get input from the user with a simple dialog box.

```
<!DOCTYPE html>
<html>
<body>
  <h1 id="greeting"></h1>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const userName = prompt("Please enter your name:");
document.getElementById("greeting").textContent = `Hello, ${userName}!`;
```

Explanation
The prompt() function displays a dialog box that prompts the user for input. It returns the text entered by the user as a string, which we store in the userName constant and then display on the page.

Part 2: Control Flow

This section explores how to make decisions and repeat actions in your code.

21. if Statement

Execute code only if a specific condition is true.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let temperature = 25; // in Celsius

if (temperature > 20) {
  document.getElementById("output").textContent = "It's a warm day!";
}
```

Explanation
The if statement evaluates the condition inside the parentheses (temperature > 20). If the condition is true, the code inside the curly braces {} is executed. If it's false, the code block is skipped entirely.

22. if...else Statement

Execute one block of code if a condition is true, and another if it is false.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let score = 75;
```

```
if (score >= 50) {
  document.getElementById("output").textContent = "You passed!";
} else {
  document.getElementById("output").textContent = "You failed. Try again.";
}
```

Explanation
If the score >= 50 condition is true, the first block runs. If it's false, the code inside the else block is executed instead. This ensures that one of the two blocks will always run.

23. if...else if...else Statement

Chain multiple conditions together.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let grade = 88;

if (grade >= 90) {
  document.getElementById("output").textContent = "Grade: A";
} else if (grade >= 80) {
  document.getElementById("output").textContent = "Grade: B";
} else if (grade >= 70) {
  document.getElementById("output").textContent = "Grade: C";
} else {
  document.getElementById("output").textContent = "Grade: F";
}
```

Explanation
JavaScript checks the conditions in order. It executes the block for the first condition that evaluates to true. Since grade (88) is not >= 90, it checks the next condition: grade >= 80. This is true, so it displays "Grade: B" and skips the rest of the chain. The final else is a fallback that runs if none of the preceding conditions are met.

24. switch Statement

An alternative to a long if...else if chain for checking a single variable against multiple values.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let day = new Date().getDay(); // Returns a number (0=Sun, 1=Mon, ...)
let dayName;

switch (day) {
  case 0:
    dayName = "Sunday";
    break;
  case 1:
    dayName = "Monday";
    break;
  case 6:
    dayName = "Saturday";
    break;
  default:
    dayName = "a weekday";
}

document.getElementById("output").textContent = `Today is ${dayName}.`;
```

Explanation
The switch statement evaluates an expression (here, day). It then compares the result with the values of each case. If a match is found, the code block associated with that case is executed. The break keyword is crucial; it stops the execution from "falling through" to the next case. The default case runs if no other case matches.

25. for Loop

Repeat a block of code a specific number of times.

```
<!DOCTYPE html>
<html>
<body>
  <ul id="list"></ul>
```

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

JavaScript (script.js)

```
JavaScript
const list = document.getElementById("list");

for (let i = 1; i <= 5; i++) {
  const listItem = document.createElement("li");
  listItem.textContent = `List item number ${i}`;
  list.appendChild(listItem);
}
```

Explanation
A for loop has three parts inside the parentheses, separated by semicolons:

Initialization (let i = 1): Runs once before the loop starts.

Condition (i <= 5): Checked before each iteration. If true, the loop continues. If false, it stops.

Increment (i++): Runs after each iteration.
This loop runs 5 times, creating and adding a new list item to the <ul> element on each pass.

26. Looping Through an Array with for

Use a for loop to access each element in an array.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const fruits = ["Apple", "Banana", "Cherry", "Date"];
let text = "";

for (let i = 0; i < fruits.length; i++) {
  text += fruits[i] + "<br>";
}

document.getElementById("output").innerHTML = text;
```

Explanation
We use the array's length property to control the loop. The condition i < fruits.length ensures the loop runs exactly once for each item in the array. Inside the loop, fruits[i] accesses the element at the current index i.

27. while Loop

Repeat a block of code as long as a condition is true.

```
<!DOCTYPE html>
<html>
<body>
 <p id="output"></p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let count = 0;
let text = "";

while (count < 3) {
 text += `The number is ${count}<br>`;
 count++;
}

document.getElementById("output").innerHTML = text;
```

Explanation
A while loop checks the condition (count < 3) before each iteration. If the condition is true, the loop body runs. It's crucial to have a way to change the condition inside the loop (here, count++), otherwise you'll create an infinite loop.

28. do...while Loop

Similar to a while loop, but the code block is executed at least once.

```
<!DOCTYPE html>
<html>
<body>
 <p id="output"></p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let i = 5;
let text = "";

do {
  text += `The number is ${i}<br>`;
  i++;
} while (i < 5); // Condition is checked after the loop body runs

document.getElementById("output").innerHTML = text;
```

Explanation
The do...while loop executes the code block first and then checks the condition. This guarantees the loop will run at least once, even if the condition is initially false (as it is in this example, since i starts at 5).

29. break Statement in a Loop

Exit a loop prematurely.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let text = "";
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // Stop the loop when i is 5
  }
  text += i + " ";
}
document.getElementById("output").textContent = text; // Outputs "0 1 2 3 4 "
```

Explanation
The break statement immediately terminates the innermost loop it's in. In this for loop, when

the counter i reaches 5, the if condition becomes true, break is executed, and the loop stops, even though it was set to run up to 10.

30. continue Statement in a Loop

Skip the current iteration and move to the next one.

```
<!DOCTYPE html>
<html>
<body>
 <p id="output"></p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let text = "";
for (let i = 0; i < 5; i++) {
 if (i === 2) {
  continue; // Skip the rest of this iteration when i is 2
 }
 text += i + " ";
}
document.getElementById("output").textContent = text; // Outputs "0 1 3 4 "
```

Explanation
When i is 2, the continue statement is executed. This immediately stops the current iteration and jumps to the next one (the increment i++ and the condition check). The number 2 is never added to the text string.

Part 3: Functions

Functions are reusable blocks of code that perform a specific task. They are a fundamental building block of any JavaScript program.

31. Function Declaration

The classic way to define a named function.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="showMessage()">Click Me</button>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
// Define the function
function showMessage() {
  document.getElementById("output").textContent = "Hello from a function!";
}
```

Explanation
A function declaration defines a function with a specified name (showMessage). This function can be called (invoked) before or after it is defined in the script because declarations are "hoisted" to the top of their scope. Here, the onclick attribute in the HTML calls the function when the button is clicked.

32. Function Expression

A function can also be defined as an expression and assigned to a variable.

```
<!DOCTYPE html>
<html>
<body>
    <p>The result is: <span id="output"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
```

```
const calculateSquare = function(number) {
  return number * number;
};

// Call the function using the variable name
const result = calculateSquare(5);
document.getElementById("output").textContent = result;
```

Explanation
Here, an unnamed (anonymous) function is created and assigned to the constant calculateSquare. Unlike function declarations, function expressions are not hoisted, meaning you can only call them after they have been defined.

33. Arrow Function (ES6)

A more concise syntax for writing function expressions.

```
<!DOCTYPE html>
<html>
<body>
    <p>The sum is: <span id="output"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

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

const sum = add(10, 20);
document.getElementById("output").textContent = sum;
```

Explanation
Arrow functions provide a shorter syntax. (a, b) are the parameters. The => (fat arrow) separates the parameters from the function body. If the function body is a single expression (like a + b), the curly braces {} and the return keyword are implicit. This makes the code much cleaner for simple functions.

34. Function with Parameters

Pass data into a function through parameters.

HTML

```
HTML
<!DOCTYPE html>
```

```
<html>
<body>
  <p id="greeting"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
function greet(name) {
 document.getElementById("greeting").textContent = `Hello, ${name}!`;
}

greet("Alice"); // "Alice" is the argument passed to the 'name' parameter
```

Explanation
Parameters (name in this case) are variables listed in the function definition. Arguments ("Alice") are the actual values passed to the function when it is called. The function then uses this passed-in data to perform its task.

35. Function with a return Value

Functions can process data and return a result.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
function multiply(num1, num2) {
 return num1 * num2; // Sends the result back out of the function
}

const product = multiply(7, 6);
document.getElementById("output").textContent = `7 * 6 = ${product}`;
```

Explanation
The return statement stops the execution of a function and returns a value to where the function was called. We call multiply(7, 6), it calculates 42, returns it, and we store that

returned value in the product constant.

36. Default Parameters

Assign default values to parameters in case no argument is provided.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output1"></p>
    <p id="output2"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
function sayHello(name = "Guest") {
  return `Hello, ${name}!`;
}

document.getElementById("output1").textContent = sayHello("Bob");   // Uses the provided argument
document.getElementById("output2").textContent = sayHello();       // Uses the default parameter
```

Explanation
By writing name = "Guest", we specify that if the sayHello function is called without an argument for name, it should use the default value "Guest". This prevents errors and makes functions more flexible.

37. Immediately Invoked Function Expression (IIFE)

A function that is defined and executed immediately.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
(function() {
  const message = "This function ran automatically!";
```

```
  document.getElementById("output").textContent = message;
})();
```

Explanation
An IIFE is an anonymous function wrapped in parentheses (...), followed by another set of parentheses () which immediately invokes it. This pattern is useful for creating a private scope, preventing variables inside the IIFE from polluting the global scope.

38. Global vs. Local Scope

Understanding where variables are accessible.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)
JavaScript
```
let globalVar = "I am global"; // Accessible everywhere

function someFunction() {
  let localVar = "I am local"; // Only accessible inside this function
  console.log(globalVar); // Can access global variables from inside
  console.log(localVar);
}

someFunction();
// console.log(localVar); // This would cause a ReferenceError

document.getElementById("output").textContent = "Check the browser console (F12) for output.";
```

Explanation
Global scope variables (like globalVar) are declared outside any function and can be accessed from anywhere in your code. Local (or function) scope variables (like localVar) are declared inside a function and can only be accessed within that function.

39. Simple alert Box

Display a simple pop-up message to the user.

```
<!DOCTYPE html>
```

```
<html>
<body>
  <button onclick="showAlert()">Show Alert</button>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
function showAlert() {
 alert("This is a simple alert box!");
}
```

Explanation
The alert() function displays a modal dialog with a specified message and an OK button. It's often used for quick debugging or for important user notifications, but it can be disruptive to the user experience.

40. Simple confirm Box

Ask the user a yes/no question.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="confirmAction()">Delete Item</button>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
function confirmAction() {
 const userChoice = confirm("Are you sure you want to delete this?");
 if (userChoice) { // confirm returns true for OK, false for Cancel
  document.getElementById("output").textContent = "Item deleted.";
 } else {
  document.getElementById("output").textContent = "Deletion cancelled.";
 }
}
```

Explanation
The confirm() function shows a dialog with a question and two buttons: OK and Cancel. It

returns a boolean value: true if the user clicks OK, and false if they click Cancel. This is useful for verifying user intentions for critical actions.

Part 4: Arrays and Objects

Learn to work with collections of data using JavaScript's most important data structures.

41. Modifying Array Elements

Change the value of an element at a specific index.

```
<!DOCTYPE html>
<html>
<body>
    <p>Original: <span id="original"></span></p>
    <p>Modified: <span id="modified"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const seasons = ["Spring", "Sumer", "Fall", "Winter"];
document.getElementById("original").textContent = seasons.join(", ");

seasons[1] = "Summer"; // Correct the typo at index 1
document.getElementById("modified").textContent = seasons.join(", ");
```

Explanation
You can directly modify an array element by accessing it via its index and using the assignment operator (=). Here, we access the second element (seasons[1]) and change its value from "Sumer" to "Summer".

42. push() - Add to End of Array

Add one or more elements to the end of an array.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const numbers = [10, 20, 30];
```

```
numbers.push(40, 50); // Adds 40 and 50 to the end
document.getElementById("output").textContent = numbers.join(", "); // "10, 20, 30, 40, 50"
```

Explanation
The push() method modifies the original array by adding the new elements to the end. It also returns the new length of the array.

43. pop() - Remove from End of Array

Removes the last element from an array.

```
<!DOCTYPE html>
<html>
<body>
    <p>Array: <span id="array"></span></p>
    <p>Removed: <span id="removed"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const tools = ["Hammer", "Screwdriver", "Wrench"];
const removedTool = tools.pop(); // Removes "Wrench"

document.getElementById("array").textContent = `Remaining tools: ${tools.join(", ")}`;
document.getElementById("removed").textContent = `Removed tool: ${removedTool}`;
```

Explanation
The pop() method removes the last element, modifies the original array, and returns the element that was removed.

44. shift() - Remove from Start of Array

Removes the first element from an array.

```
<!DOCTYPE html>
<html>
<body>
    <p>Array: <span id="array"></span></p>
    <p>Removed: <span id="removed"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let queue = ["First", "Second", "Third"];
let served = queue.shift(); // Removes "First"

document.getElementById("array").textContent = `Current queue: ${queue.join(", ")}`;
document.getElementById("removed").textContent = `Now serving: ${served}`;
```

Explanation
The shift() method is the opposite of pop(). It removes the first element, modifies the array, and returns the removed element.

45. unshift() - Add to Start of Array

Adds one or more elements to the beginning of an array.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
let tasks = ["Task 2", "Task 3"];
tasks.unshift("Task 1"); // Adds "Task 1" to the beginning

document.getElementById("output").textContent = tasks.join(", ");
```

Explanation
The unshift() method is the opposite of push(). It adds elements to the beginning of the array, modifying it in place.

46. forEach() Loop

Executes a provided function once for each array element.

```
<!DOCTYPE html>
<html>
<body>
    <ul id="list"></ul>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const pets = ["Cat", "Dog", "Fish"];
const list = document.getElementById("list");

pets.forEach(function(pet) {
  const li = document.createElement("li");
  li.textContent = pet;
  list.appendChild(li);
});
```

Explanation
The forEach() method is a cleaner way to loop over an array than a traditional for loop. It takes a "callback" function as an argument. This function is executed for every item in the array, and the item itself is passed as an argument to the function (pet).

47. map() Method

Creates a new array by performing an operation on every element of an existing array.

```
<!DOCTYPE html>
<html>
<body>
    <p>Original: <span id="original"></span></p>
    <p>Doubled: <span id="mapped"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(function(num) {
  return num * 2;
});

document.getElementById("original").textContent = numbers.join(", ");
document.getElementById("mapped").textContent = doubled.join(", ");
```

Explanation
The map() method iterates through an array, applies a callback function to each element, and returns a new array containing the results. The original array is not changed. It's extremely

useful for transforming data.

48. filter() Method

Creates a new array with all elements that pass a test.

```
<!DOCTYPE html>
<html>
<body>
  <p>All scores: <span id="original"></span></p>
  <p>Passing scores: <span id="filtered"></span></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const scores = [88, 45, 92, 67, 50, 78];
const passingScores = scores.filter(function(score) {
 return score >= 60;
});

document.getElementById("original").textContent = scores.join(", ");
document.getElementById("filtered").textContent = passingScores.join(", ");
```

Explanation
The filter() method creates a new array. It iterates through the original array and includes an element in the new array only if the callback function returns true for that element.

49. Accessing Object Properties

Use dot or bracket notation to get values from an object.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output1"></p>
  <p id="output2"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const car = {
 make: "Honda",
```

```
  model: "Civic",
  "year-of-manufacture": 2022
};

// Dot notation
document.getElementById("output1").textContent = `Make: ${car.make}`;

// Bracket notation (required for keys with special characters or variables)
document.getElementById("output2").textContent = `Year: ${car["year-of-manufacture"]}`;
```

Explanation

Dot notation (car.make) is the most common and is easy to read.

Bracket notation (car['model']) is required when the key is not a valid JavaScript identifier (e.g., it contains spaces or hyphens) or when the key is stored in a variable.

50. Object Methods

An object property can be a function.

```
<!DOCTYPE html>
<html>
<body>
   <p id="output"></p>
   <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};

document.getElementById("output").textContent = person.fullName();
```

Explanation
When a function is a property of an object, it's called a method. Inside a method, the this keyword refers to the object the method belongs to. So, this.firstName refers to the firstName property of the person object. We call the method using parentheses, just like a regular

function.

51. for...in Loop for Objects

Iterate over the properties of an object.

```
<!DOCTYPE html>
<html>
<body>
  <div id="output"></div>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const user = {
 name: "Sam",
 email: "sam@example.com",
 isAdmin: false
};

let text = "";
for (let key in user) {
 text += `${key}: ${user[key]}<br>`;
}

document.getElementById("output").innerHTML = text;
```

Explanation
The for...in loop iterates over the enumerable property names (keys) of an object. In each iteration, the key variable holds the current property name as a string (e.g., "name", "email"). You must use bracket notation (user[key]) to access the value because the key name is stored in a variable.

52. JSON stringify()

Convert a JavaScript object into a JSON string.

```
<!DOCTYPE html>
<html>
<body>
  <p>JavaScript Object:</p>
  <pre id="object-output"></pre>
  <p>JSON String:</p>
  <pre id="json-output"></pre>
```

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

JavaScript (script.js)

```
JavaScript
const product = {
  id: 123,
  name: "Laptop",
  inStock: true
};

const jsonString = JSON.stringify(product, null, 2); // 2 is for indentation

document.getElementById("object-output").textContent = "See console for object";
console.log(product);
document.getElementById("json-output").textContent = jsonString;
```

Explanation
JSON (JavaScript Object Notation) is a text-based format for representing data. JSON.stringify() takes a JavaScript object and converts it into a JSON string. This is essential for sending data to a server or storing it in localStorage. The extra arguments (null, 2) are optional and format the string with indentation for readability.

53. JSON parse()

Convert a JSON string back into a JavaScript object.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const jsonStringFromServer = '{"id":456,"title":"Coffee Mug","price":9.99}';

const productObject = JSON.parse(jsonStringFromServer);

document.getElementById("output").textContent = `Product: ${productObject.title}, Price:
$${productObject.price}`;
```

Explanation
JSON.parse() is the inverse of stringify(). It takes a valid JSON string and transforms it into a JavaScript object that you can then work with in your code, accessing its properties using dot or bracket notation.

Part 5: DOM Manipulation and Events

This is where JavaScript interacts with the HTML page, making websites dynamic and interactive. The **DOM (Document Object Model)** is the browser's tree-like representation of your HTML.

54. Select an Element by ID

The most common way to grab a specific element.

```
<!DOCTYPE html>
<html>
<body>
    <h1 id="main-title">Hello, World!</h1>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const titleElement = document.getElementById("main-title");
titleElement.style.color = "blue";
```

Explanation
document.getElementById() searches the entire HTML document for an element with a matching id attribute. Since IDs must be unique, this method is very fast and reliable for targeting a single, specific element. Once selected, you can manipulate its properties, like its style.

55. Select an Element with querySelector

A more versatile way to select the *first* element that matches a CSS selector.

```
<!DOCTYPE html>
<html>
<body>
    <p class="content">This is the first paragraph.</p>
    <p class="content">This is the second paragraph.</p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
// Selects the FIRST element with the class "content"
```

```
const firstParagraph = document.querySelector(".content");
firstParagraph.style.fontWeight = "bold";
```

Explanation
querySelector() can use any valid CSS selector (like .content, #main-title, div p, etc.). It's very powerful but only ever returns the very first element it finds that matches the selector.

56. Select Multiple Elements with querySelectorAll

Select all elements that match a CSS selector.

```
<!DOCTYPE html>
<html>
<body>
    <ul>
        <li class="item">Apple</li>
        <li class="item">Banana</li>
        <li class="item">Cherry</li>
    </ul>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const listItems = document.querySelectorAll(".item");

// listItems is a NodeList, which we can loop over
listItems.forEach(function(item) {
  item.style.color = "green";
});
```

Explanation
querySelectorAll() returns a static NodeList (which is like an array) containing all elements in the document that match the specified selector. You can then iterate over this list using forEach to apply changes to every matched element.

57. Change Content with innerHTML

Change the entire HTML content inside an element.

```
<!DOCTYPE html>
<html>
<body>
    <div id="container">This is old content.</div>
```

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

JavaScript (script.js)

```
JavaScript
const container = document.getElementById("container");
container.innerHTML = "<h2>This is a new heading!</h2><p>And a new paragraph.</p>";
```

Explanation
The innerHTML property gets or sets the HTML markup contained within an element. When you set it, the browser parses the string as HTML and renders it. Be cautious: setting innerHTML with user-provided content can create a security risk (Cross-Site Scripting - XSS) if not handled properly.

58. Change Text with textContent

Change only the text content of an element, ignoring any HTML.

```
<!DOCTYPE html>
<html>
<body>
    <div id="container">This will be <strong>replaced</strong>.</div>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const container = document.getElementById("container");
// Any HTML tags in the string will be rendered as plain text
container.textContent = "This is just new, plain text.";
```

Explanation
textContent gets or sets only the text portion of an element and its descendants. It's a safer and often faster alternative to innerHTML when you only need to work with text, as it automatically neutralizes any HTML tags.

59. Change an Element's Style

Directly modify the CSS styles of an element.

```
<!DOCTYPE html>
<html>
<body>
```

```
  <p id="my-text">Style me with JavaScript!</p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const myText = document.getElementById("my-text");
myText.style.color = "white";
myText.style.backgroundColor = "navy";
myText.style.padding = "20px";
myText.style.fontSize = "24px";
```

Explanation
Every HTML element has a style property, which is an object. You can change individual CSS properties on this object. Note that CSS property names with hyphens (like background-color) are converted to camelCase in JavaScript (e.g., backgroundColor).

60. Add/Remove a CSS Class

The best way to change an element's style is by toggling CSS classes.

```
<!DOCTYPE html>
<html>
<head>
 <style>
  .highlight {
    background-color: yellow;
    border: 2px solid red;
    font-weight: bold;
  }
 </style>
</head>
<body>
  <p id="message">Click the button to highlight me.</p>
  <button id="toggle-btn">Toggle Highlight</button>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const message = document.getElementById("message");
const toggleBtn = document.getElementById("toggle-btn");
```

```
toggleBtn.addEventListener("click", function() {
  // toggle adds the class if it's not there, removes it if it is
  message.classList.toggle("highlight");
});
```

Explanation
Manipulating classes is better than inline styles for managing appearance. The classList property provides methods like add(), remove(), and toggle() to easily manage an element's classes without manually handling the class string.

61. Get and Set Attributes

Work with HTML attributes like src, href, or disabled.

```
<!DOCTYPE html>
<html>
<body>
    <img id="my-image" src="https://via.placeholder.com/150" alt="Placeholder">
    <button id="change-btn">Change Image</button>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const myImage = document.getElementById("my-image");
const changeBtn = document.getElementById("change-btn");

changeBtn.addEventListener("click", function() {
  // Get the current src attribute
  const currentSrc = myImage.getAttribute("src");
  console.log("Current source:", currentSrc);

  // Set a new src attribute
  myImage.setAttribute("src", "https://via.placeholder.com/250");
});
```

Explanation

getAttribute('attributeName') reads the current value of an HTML attribute.

setAttribute('attributeName', 'newValue') changes the value of an attribute or adds it if it doesn't exist.

62. Create and Append Elements

Dynamically create new HTML elements and add them to the page.

```
<!DOCTYPE html>
<html>
<body>
  <div id="container">
    <h2>Groceries</h2>
    <ul id="grocery-list">
      <li>Milk</li>
    </ul>
  </div>
  <button id="add-btn">Add Item</button>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

JavaScript

```
const addBtn = document.getElementById("add-btn");
const list = document.getElementById("grocery-list");

addBtn.addEventListener("click", function() {
 // 1. Create a new element
 const newItem = document.createElement("li");

 // 2. Add content to it
 newItem.textContent = "Bread";

 // 3. Append it to a parent element on the page
 list.appendChild(newItem);
});
```

Explanation
This is a three-step process:

document.createElement('tagName') creates a new element in memory (it's not on the page yet).

You then modify its properties (like textContent or className).

parentElement.appendChild(newElement) inserts the newly created element as the last child of the parent element, making it visible on the page.

63. Handling a click Event

The most fundamental user interaction.

```
<!DOCTYPE html>
<html>
<body>
  <button id="my-btn">Click Me!</button>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const myBtn = document.getElementById("my-btn");
const output = document.getElementById("output");

myBtn.addEventListener("click", function() {
  output.textContent = "Button was clicked at " + new Date().toLocaleTimeString();
});
```

Explanation
addEventListener() is the modern, preferred way to handle events. It takes two main arguments:

The name of the event to listen for (e.g., 'click', 'mouseover').

A "callback" function to execute when the event occurs.

64. Getting Value from an Input Field

Read what a user has typed into a form input.

```
<!DOCTYPE html>
<html>
<body>
  <input type="text" id="name-input" placeholder="Enter your name">
  <button id="greet-btn">Greet</button>
  <h2 id="greeting"></h2>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const nameInput = document.getElementById("name-input");
const greetBtn = document.getElementById("greet-btn");
```

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

greetBtn.addEventListener("click", function() {
  const userName = nameInput.value; // Get the current value of the input
  greeting.textContent = `Hello, ${userName}!`;
});
```

Explanation
For input elements (<input>, <textarea>, <select>), you use the .value property to get or set the current content.

65. Preventing Form Submission

Stop a form's default behavior to handle it with JavaScript instead.

```
<!DOCTYPE html>
<html>
<body>
    <form id="my-form">
        <input type="text" id="my-input" required>
        <button type="submit">Submit</button>
    </form>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)
JavaScript

```
const myForm = document.getElementById("my-form");
const output = document.getElementById("output");

myForm.addEventListener("submit", function(event) {
  // Prevent the default browser action (page reload)
  event.preventDefault();

  const inputValue = document.getElementById("my-input").value;
  output.textContent = `Form submitted with value: ${inputValue}`;
  // In a real app, you would now send this data to a server.
});
```

Explanation
When a form is submitted, the browser's default action is to reload the page. The event object is automatically passed to the event listener function. Calling event.preventDefault() stops this

default action, allowing you to handle the form data using JavaScript without a page refresh (a technique known as AJAX).

66. The event Object

Event listeners receive an event object with information about the interaction.

```
<!DOCTYPE html>
<html>
<head>
 <style>
  #box { width: 200px; height: 200px; background-color: lightblue; border: 1px solid black; }
 </style>
</head>
<body>
  <div id="box"></div>
  <p id="coords"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const box = document.getElementById("box");
const coords = document.getElementById("coords");

box.addEventListener("mousemove", function(e) {
 // The event object 'e' has properties like clientX and clientY
 coords.textContent = `Mouse coordinates: X=${e.clientX}, Y=${e.clientY}`;
});
```

Explanation
The event object (often abbreviated to e) contains useful information about the event that just happened. For a mouse event, this includes the coordinates of the pointer (clientX, clientY). For a keyboard event, it includes which key was pressed (e.key).

67. Keyboard Events (keydown)

Respond to a user pressing a key on the keyboard.

```
<!DOCTYPE html>
<html>
<body>
  <p>Type in the input field.</p>
  <input type="text" id="my-input">
  <p id="output"></p>
```

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

JavaScript (script.js)

```
JavaScript
const myInput = document.getElementById("my-input");
const output = document.getElementById("output");

myInput.addEventListener("keydown", function(event) {
  // The 'key' property gives the name of the key that was pressed
  if (event.key === "Enter") {
    output.textContent = `You pressed Enter with value: "${myInput.value}"`;
    myInput.value = ""; // Clear the input
  }
});
```

Explanation
The keydown event fires when a key is pressed down. We can inspect the key property of the event object to see which key it was. This is useful for creating shortcuts or submitting forms when the user presses "Enter". Other keyboard events include keyup (when the key is released) and keypress (legacy, avoid using).

68. mouseover and mouseout Events

Change something when the mouse pointer enters or leaves an element.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    #hover-box {
      width: 300px;
      padding: 50px;
      text-align: center;
      background-color: #eee;
      border: 1px solid #ccc;
      transition: background-color 0.3s;
    }
  </style>
</head>
<body>
    <div id="hover-box">Hover over me!</div>
    <script src="script.js"></script>
</body>
```

```
</html>
```

JavaScript (script.js)

```
JavaScript
const hoverBox = document.getElementById("hover-box");

hoverBox.addEventListener("mouseover", function() {
  hoverBox.textContent = "You are hovering!";
  hoverBox.style.backgroundColor = "lightgreen";
});

hoverBox.addEventListener("mouseout", function() {
  hoverBox.textContent = "Hover over me!";
  hoverBox.style.backgroundColor = "#eee";
});
```

Explanation

mouseover fires when the mouse pointer enters the element.

mouseout fires when the mouse pointer leaves the element.
These are great for creating interactive UI effects, like revealing buttons or showing tooltips.

69. The change Event for Inputs

Fire an event when the value of an input element has been changed.

```
<!DOCTYPE html>
<html>
<body>
    <label for="color-select">Choose a color:</label>
    <select id="color-select">
        <option value="">--Please choose an option--</option>
        <option value="blue">Blue</option>
        <option value="green">Green</option>
        <option value="red">Red</option>
    </select>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const colorSelect = document.getElementById("color-select");
```

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

colorSelect.addEventListener("change", function(event) {
  const selectedValue = event.target.value;
  output.textContent = `You selected: ${selectedValue}`;
  document.body.style.backgroundColor = selectedValue;
});
```

Explanation
The change event fires for <input>, <select>, and <textarea> elements when their value is committed by the user. For a dropdown <select>, this happens as soon as a new option is chosen. For a text input, it usually fires when the element loses focus (the user clicks away). event.target refers to the element that triggered the event (the select box in this case).

70. Reading data-* Attributes

Store custom data directly on HTML elements.

```
<!DOCTYPE html>
<html>
<head>
  <style> button { margin: 5px; } </style>
</head>
<body>
    <button class="user-btn" data-user-id="123" data-user-role="admin">User 1</button>
    <button class="user-btn" data-user-id="456" data-user-role="editor">User 2</button>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const buttons = document.querySelectorAll(".user-btn");
const output = document.getElementById("output");

buttons.forEach(button => {
  button.addEventListener("click", function(event) {
    const userId = event.target.dataset.userId; // Access data-user-id
    const userRole = event.target.dataset.userRole; // Access data-user-role
    output.textContent = `Button clicked for User ID: ${userId}, Role: ${userRole}`;
  });
});
```

Explanation
Custom data attributes are prefixed with data-. In JavaScript, you can access these through the element's dataset property. The part of the attribute name after data- is converted to camelCase (e.g., data-user-id becomes dataset.userId). This is a clean way to attach extra information to an element without misusing id or class.

Part 6: Asynchronous JavaScript & Web APIs

This section covers tasks that don't happen instantly, like waiting for a timer, fetching data from a server, or interacting with browser features.

71. setTimeout - Delay Execution

Execute a function once after a specified delay.

```
<!DOCTYPE html>
<html>
<body>
  <button id="show-msg-btn">Show Message in 3 Seconds</button>
  <h2 id="output"></h2>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const showMsgBtn = document.getElementById("show-msg-btn");
const output = document.getElementById("output");

showMsgBtn.addEventListener("click", function() {
 output.textContent = "Waiting for the message...";
 setTimeout(function() {
  output.textContent = "Here is your delayed message!";
 }, 3000); // Delay is in milliseconds (3000ms = 3s)
});
```

Explanation
setTimeout(callbackFunction, delayInMs) schedules a function to run in the future. It does not pause the code. The script continues to run, and after 3000 milliseconds have passed, the browser will execute the callback function.

72. setInterval - Repeat Execution

Execute a function repeatedly at a fixed time interval.

```
<!DOCTYPE html>
<html>
<body>
  <h1>Current Time: <span id="clock"></span></h1>
  <script src="script.js"></script>
</body>
```

```
</html>
```

JavaScript (script.js)

```
JavaScript
const clock = document.getElementById("clock");

// Execute the updateClock function every 1000 milliseconds (1 second)
setInterval(updateClock, 1000);

function updateClock() {
  const now = new Date();
  clock.textContent = now.toLocaleTimeString();
}
```

Explanation
setInterval(callbackFunction, delayInMs) is similar to setTimeout, but it will execute the callback function over and over again, with the specified delay between each execution. It's perfect for things like live clocks or polling a server for updates.

73. clearInterval - Stop setInterval

Stop a repeating timer that was started with setInterval.

```
<!DOCTYPE html>
<html>
<body>
    <p>Timer: <span id="counter">0</span></p>
    <button id="stop-btn">Stop Timer</button>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const counter = document.getElementById("counter");
const stopBtn = document.getElementById("stop-btn");
let count = 0;

// setInterval returns an ID we can use to stop it later
const intervalId = setInterval(function() {
  count++;
  counter.textContent = count;
}, 500);
```

```
stopBtn.addEventListener("click", function() {
  clearInterval(intervalId); // Stop the timer
  counter.textContent += " (Stopped)";
});
```

Explanation
setInterval returns a unique ID for the timer it creates. You can store this ID in a variable and later pass it to clearInterval() to stop the repeating execution.

74. Fetch API - Get Data from a Server

The modern way to make network requests (e.g., to an API).

```
<!DOCTYPE html>
<html>
<body>
    <button id="fetch-btn">Fetch User Data</button>
    <pre id="output"></pre>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const fetchBtn = document.getElementById("fetch-btn");
const output = document.getElementById("output");
// Using a public test API
const API_URL = "https://jsonplaceholder.typicode.com/users/1";

fetchBtn.addEventListener("click", function() {
  fetch(API_URL)
    .then(response => response.json()) // 1. Convert the response to JSON
    .then(data => { // 2. Work with the JSON data
      output.textContent = JSON.stringify(data, null, 2);
    })
    .catch(error => { // Handle any errors
      output.textContent = "Error fetching data: " + error;
      console.error('Fetch Error:', error);
    });
});
```

Explanation
fetch() starts the process of fetching a resource from the network. It returns a Promise.

The first .then() block receives the response object. We call .json() on it to parse the response body as JSON. This also returns a promise.

The second .then() block receives the actual parsed JSON data.

The .catch() block will execute if any error occurs during the network request.

75. async/await - Cleaner Asynchronous Code

A modern syntax that makes promise-based code look and behave more like synchronous code.

```
<!DOCTYPE html>
<html>
<body>
  <button id="fetch-btn-async">Fetch TODO Item (Async/Await)</button>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const fetchBtnAsync = document.getElementById("fetch-btn-async");
const output = document.getElementById("output");
const TODO_URL = "https://jsonplaceholder.typicode.com/todos/1";

fetchBtnAsync.addEventListener("click", async function() {
 output.textContent = "Fetching...";
 try {
  const response = await fetch(TODO_URL); // Pause here until fetch completes
  const data = await response.json(); // Pause here until JSON parsing completes
  output.textContent = `Todo Title: "${data.title}"`;
 } catch (error) {
  output.textContent = "Failed to load data.";
  console.error("Async/Await Error:", error);
 }
});
```

Explanation

The async keyword is placed before a function declaration to turn it into an async function.

The await keyword can only be used inside an async function. It pauses the function's execution until the Promise is resolved and returns its result.

We use a try...catch block to handle errors in a way that is familiar from synchronous

programming. This is often considered more readable than long .then() chains.

76. Saving to Local Storage

Store simple key-value data in the user's browser, persisting even after the browser is closed.

```
<!DOCTYPE html>
<html>
<body>
  <input type="text" id="theme-input" placeholder="e.g., dark or light">
  <button id="save-btn">Save Theme</button>
  <p id="status"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const themeInput = document.getElementById("theme-input");
const saveBtn = document.getElementById("save-btn");
const status = document.getElementById("status");

saveBtn.addEventListener("click", function() {
 const theme = themeInput.value;
 // localStorage only stores strings
 localStorage.setItem("userTheme", theme);
 status.textContent = `Theme '${theme}' saved!`;
});
```

Explanation
localStorage.setItem(key, value) saves a data item in the browser's local storage. The key and value must both be strings. This data will remain until it is explicitly cleared by the web app or the user.

77. Reading from Local Storage

Retrieve data that was previously saved in local storage.

```
<!DOCTYPE html>
<html>
<body style="background-color: white;">
  <h1>My Website</h1>
  <p>Your saved theme will be applied on page load.</p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
// This code runs as soon as the page loads
const savedTheme = localStorage.getItem("userTheme");

if (savedTheme) { // Check if a value was found
  console.log(`Found saved theme: ${savedTheme}`);
  if (savedTheme === 'dark') {
      document.body.style.backgroundColor = 'black';
      document.body.style.color = 'white';
  }
} else {
    console.log("No theme found in local storage.");
}
```

Explanation
localStorage.getItem(key) retrieves the value associated with the given key. If the key does not exist, it returns null. This is useful for remembering user preferences or settings across sessions.

78. Removing from Local Storage

Delete a specific item from local storage.

```
<!DOCTYPE html>
<html>
<body>
    <p>Current Theme: <span id="current-theme"></span></p>
    <button id="remove-btn">Forget My Theme</button>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const removeBtn = document.getElementById("remove-btn");
const currentTheme = document.getElementById("current-theme");

// Display current theme on load
currentTheme.textContent = localStorage.getItem("userTheme") || "Not set";

removeBtn.addEventListener("click", function() {
  localStorage.removeItem("userTheme");
```

```
  currentTheme.textContent = "Not set";
  alert("Theme setting has been removed.");
});
```

Explanation
localStorage.removeItem(key) permanently deletes the key-value pair from the browser's storage for that domain.

79. Working with the Date Object

Create and format dates and times.

```
<!DOCTYPE html>
<html>
<body>
    <p>Full Date/Time: <span id="full-date"></span></p>
    <p>Just the Date: <span id="short-date"></span></p>
    <p>Just the Time: <span id="time"></span></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)
JavaScript

```
const now = new Date(); // Creates a date object for the current moment

document.getElementById("full-date").textContent = now.toString();
document.getElementById("short-date").textContent = now.toLocaleDateString();
document.getElementById("time").textContent = now.toLocaleTimeString();
```

Explanation
Creating new Date() gives you an object representing the current date and time. This object has many useful methods for formatting the date and time into human-readable strings, such as toString(), toLocaleDateString(), and toLocaleTimeString().

80. Get Parts of a Date

Extract individual components like the year, month, or day.

```
<!DOCTYPE html>
<html>
<body>
    <p id="output"></p>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const today = new Date();

const year = today.getFullYear();
const month = today.getMonth() + 1; // getMonth() is 0-indexed (0=Jan, 1=Feb, etc.)
const day = today.getDate();
const weekday = today.getDay(); // 0=Sunday, 1=Monday, etc.

const weekDays = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];

document.getElementById("output").textContent = `Today is ${weekDays[weekday]},
${month}/${day}/${year}.`;
```

Explanation
The Date object provides getter methods to extract specific pieces of information. It's important to remember that getMonth() is zero-indexed, so you typically need to add 1 to get the correct month number.

(This concludes the first 80 samples. The remaining 20 will cover more intermediate concepts and variations.)

81. The Ternary Operator

A compact shortcut for an if...else statement.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const age = 20;

// Syntax: condition ? value_if_true : value_if_false
const message = (age >= 18) ? "Can vote" : "Cannot vote";

document.getElementById("output").textContent = `A person of age ${age}: ${message}.`;
```

Explanation
The ternary operator is the only JavaScript operator that takes three operands. It first evaluates the condition (age >= 18). If the condition is true, it returns the first value ("Can vote"). If the condition is false, it returns the second value ("Cannot vote"). It's excellent for short, conditional assignments.

82. Array.find() Method

Get the first element in an array that satisfies a condition.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const products = [
  { id: 1, name: "Book", price: 15 },
  { id: 2, name: "Pen", price: 2 },
  { id: 3, name: "Laptop", price: 1200 },
  { id: 4, name: "Pen", price: 3 }
];

const foundProduct = products.find(product => product.name === "Pen");

document.getElementById("output").textContent =
  `Found product: ${foundProduct.name} with ID ${foundProduct.id} and price ${foundProduct.price}.`;
```

Explanation
The find() method iterates through the array and executes the callback function for each element. It returns the value of the first element for which the callback function returns true. If no element is found, it returns undefined. Unlike filter(), it stops as soon as it finds a match and only returns one item, not an array.

83. Array.includes() Method

Check if an array contains a certain value.

```
<!DOCTYPE html>
<html>
```

```
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const permissions = ["read", "write", "delete"];

const hasWritePermission = permissions.includes("write"); // true
const hasExecutePermission = permissions.includes("execute"); // false

document.getElementById("output").textContent =
  `User has 'write' permission: ${hasWritePermission}. User has 'execute' permission:
${hasExecutePermission}.`;
```

Explanation
The includes() method determines whether an array includes a certain value among its entries, returning true or false as appropriate. It's a simple and highly readable way to check for the existence of an item.

84. Spread Syntax for Arrays

Expand an array into individual elements. Useful for combining arrays.

```
<!DOCTYPE html>
<html>
<body>
  <p>Combined Array: <span id="output"></span></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const vegetables = ["Carrot", "Broccoli"];
const fruits = ["Apple", "Orange"];

const combined = [...vegetables, "Potato", ...fruits];

document.getElementById("output").textContent = combined.join(', ');
```

Explanation
The spread syntax (...) takes an iterable (like an array) and expands it into its individual elements. Here, we use it to create a new combined array by spreading out the contents of vegetables and fruits into the new array literal, along with another string.

85. Spread Syntax for Objects

Expand an object's properties into a new object. Useful for merging objects.

```
<!DOCTYPE html>
<html>
<body>
  <pre id="output"></pre>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const user = { name: "Alex", isAdmin: false };
const permissions = { canPost: true, canEdit: false };

const mergedUser = { ...user, ...permissions, isAdmin: true }; // Properties on the right overwrite those on the left

document.getElementById("output").textContent = JSON.stringify(mergedUser, null, 2);
```

Explanation
Similar to arrays, the spread syntax can be used on objects to expand their own enumerable properties. It's a common way to create a new object by merging the properties of two or more other objects. If properties have the same key, the value from the last object spread is the one that is used (notice how isAdmin becomes true).

86. Destructuring Arrays

Unpack values from arrays into distinct variables.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

JavaScript
```
const coordinates = [10, 20, 30];

const [x, y] = coordinates; // Assign first element to x, second to y

document.getElementById("output").textContent = `x = ${x}, y = ${y}`;
```

Explanation
Destructuring assignment provides a concise syntax to extract data from arrays or objects. In this example, const [x, y] = coordinates; creates two new constants, x and y, and assigns the first two values from the coordinates array to them, respectively.

87. Destructuring Objects

Unpack properties from objects into distinct variables.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)
JavaScript
```
const settings = {
  theme: 'dark',
  fontSize: 16,
  notifications: true
};

const { theme, fontSize } = settings; // Variable names must match property keys

document.getElementById("output").textContent = `Theme: ${theme}, Font Size: ${fontSize}px`;
```

Explanation
Object destructuring allows you to unpack properties into variables with the same name. The syntax { theme, fontSize } looks for properties named theme and fontSize on the settings object and creates local constants with their values.

88. The Math Object

Use built-in mathematical functions and constants.

```
<!DOCTYPE html>
<html>
<body>
  <p>Random number (0-1): <span id="rand"></span></p>
  <p>Rounded down (floor): <span id="floor"></span></p>
  <p>Largest number (max): <span id="max"></span></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
document.getElementById("rand").textContent = Math.random();
document.getElementById("floor").textContent = Math.floor(4.9); // Returns 4
document.getElementById("max").textContent = Math.max(10, 50, 25); // Returns 50
```

Explanation

Math is a built-in object that has properties and methods for mathematical constants and functions. It is not a constructor; you simply call methods on it directly.

Math.random() returns a random floating-point number between 0 (inclusive) and 1 (exclusive).

Math.floor() rounds a number down to the nearest integer.

Math.max() returns the largest of the given numbers.

89. String Method: toUpperCase()

Convert a string to all uppercase letters.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const originalText = "Hello World";
const upperText = originalText.toUpperCase();

document.getElementById("output").textContent = upperText; // "HELLO WORLD"
```

Explanation
The toUpperCase() method returns the calling string value converted to uppercase. It does not change the original string. There is a corresponding toLowerCase() method as well.

90. String Method: slice()

Extract a section of a string and returns it as a new string.

```
<!DOCTYPE html>
<html>
<body>
 <p id="output"></p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const text = "JavaScript is fun!";

// Extracts from index 4 up to (but not including) index 10
const slicedText = text.slice(4, 10);

document.getElementById("output").textContent = slicedText; // "Script"
```

Explanation
slice(startIndex, endIndex) extracts a portion of a string. The endIndex is optional; if omitted, it slices to the end of the string. The original string is not modified.

91. String Method: split()

Divide a string into an ordered list of substrings.

```
<!DOCTYPE html>
<html>
<body>
 <pre id="output"></pre>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const csvString = "apple,banana,cherry,date";
```

```
const fruitArray = csvString.split(',');

document.getElementById("output").textContent = JSON.stringify(fruitArray, null, 2);
```

Explanation
The split() method takes a pattern (in this case, a comma ,) and divides a string into an array of substrings by searching for that pattern. It's incredibly useful for parsing data.

92. Array Method: join()

Create and return a new string by concatenating all of the elements in an array.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)
JavaScript

```
const words = ["This", "is", "a", "sentence."];

const sentence = words.join(" "); // Joins elements with a space

document.getElementById("output").textContent = sentence;
```

Explanation
join() is the inverse of split(). It takes an optional separator argument that specifies a string to separate each pair of adjacent elements of the array. If omitted, the array elements are separated with a comma.

93. Array.isArray()

Determine if a value is an array.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output1"></p>
  <p id="output2"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const anArray = [1, 2, 3];
const anObject = { a: 1, b: 2 };

document.getElementById("output1").textContent = `Is [1, 2, 3] an array? ${Array.isArray(anArray)}`;
document.getElementById("output2").textContent = `Is {a:1, b:2} an array? ${Array.isArray(anObject)}`;
```

Explanation
Because arrays are technically a type of object in JavaScript, typeof will return "object" for an array. Array.isArray() is the correct and reliable way to check if a variable is specifically an array.

94. Number Method: toFixed()

Format a number using fixed-point notation (for decimals).

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const price = 19.991234;

const formattedPrice = price.toFixed(2); // Rounds to 2 decimal places

document.getElementById("output").textContent = `The price is $${formattedPrice}`; // "$19.99"
```

Explanation
The toFixed() method formats a number to a specified number of decimal places and returns the result as a string. It will round the number if necessary. This is essential for displaying currency.

95. Converting String to Number

Explicitly convert a string that looks like a number into an actual number type.

```
<!DOCTYPE html>
```

```
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const stringValue = "100";
const numericValue = parseInt(stringValue, 10); // The 10 is the radix (base 10)

const result = numericValue + 50; // Mathematical operation is now possible

document.getElementById("output").textContent = `Result of 100 + 50 is ${result}. Type is ${typeof numericValue}.`;
```

Explanation
parseInt() parses a string argument and returns an integer. It's crucial to specify the radix (the base in mathematical systems, almost always 10 for decimal numbers) to avoid unexpected behavior. For decimal numbers, parseFloat() can be used. A simpler modern method is to use the Number() constructor: Number("100.5").

96. The window Object

The top-level global object in a browser environment.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
// These functions are actually methods of the window object
// window.alert("Hello!");
// window.console.log("This is a log");

const browserWidth = window.innerWidth;
const browserHeight = window.innerHeight;
```

```
document.getElementById("output").textContent =
  `Your browser viewport is ${browserWidth}px wide and ${browserHeight}px tall.`;
```

Explanation
The window object represents the browser window. All global JavaScript objects, functions, and variables automatically become members of the window object. This includes things like document, console, alert, and localStorage. You can usually omit window. when calling these, but it's good to know where they come from.

97. Error Handling with try...catch

Gracefully handle runtime errors without crashing the script.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
try {
  // Code that might fail
  const data = JSON.parse("{ 'name': 'John', 'age': 30 }"); // Invalid JSON (uses single quotes)
  document.getElementById("output").textContent = `Welcome, ${data.name}!`;
} catch (error) {
  // This block runs if an error occurs in the 'try' block
  document.getElementById("output").textContent = "Something went wrong! See console for
details.";
  console.error("An error occurred:", error.message);
}
```

Explanation
The try...catch statement allows you to test a block of code for errors. The try block contains the code to be executed. The catch block contains the code to be executed if an error occurs. This prevents a single error from halting the execution of the entire script.

98. Creating a Simple Promise

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation.

```
<!DOCTYPE html>
```

```
<html>
<body>
 <p id="output">Waiting for the promise...</p>
 <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
const myPromise = new Promise((resolve, reject) => {
 // Simulate an async operation like a network request
 setTimeout(() => {
   const success = true; // Change to false to see the reject case
   if (success) {
     resolve("The operation was successful!"); // Send back data on success
   } else {
     reject("The operation failed."); // Send back an error on failure
   }
 }, 2000);
});

myPromise
 .then(message => {
   document.getElementById("output").textContent = `Success: ${message}`;
 })
 .catch(error => {
   document.getElementById("output").textContent = `Error: ${error}`;
 });
```

Explanation
A Promise takes a function with two arguments: resolve and reject. Inside, you perform your async task. If the task completes successfully, you call resolve() with the result. If it fails, you call reject() with an error. You then use .then() to handle the success case and .catch() to handle the failure.

99. Promise.all()

Wait for multiple promises to all complete.

```
<!DOCTYPE html>
<html>
<body>
 <p id="output">Fetching multiple resources...</p>
 <script src="script.js"></script>
```

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

JavaScript (script.js)

```
JavaScript
const promise1 = fetch('https://jsonplaceholder.typicode.com/todos/1').then(res => res.json());
const promise2 = fetch('https://jsonplaceholder.typicode.com/posts/1').then(res => res.json());

Promise.all([promise1, promise2])
  .then(results => {
    // results is an array containing the results of each promise
    console.log(results);
    const todoTitle = results[0].title;
    const postTitle = results[1].title;

    document.getElementById("output").innerHTML =
      `Todo Title: ${todoTitle}<br>Post Title: ${postTitle}`;
  })
  .catch(error => {
    document.getElementById("output").textContent = "One of the requests failed.";
    console.error(error);
  });
```

Explanation
Promise.all() takes an array of promises and returns a single new promise. This new promise resolves when all of the input promises have resolved, returning an array of their results. If even one of the input promises rejects, the entire Promise.all rejects immediately.

100. Adding a Comment in JavaScript

Leave notes in your code for yourself and others.

```
<!DOCTYPE html>
<html>
<body>
  <p id="output"></p>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

```
JavaScript
// This is a single-line comment. The code below gets an element.
const outputElement = document.getElementById("output");
```

```
/* This is a multi-line comment.
  It can span across several lines.
  The following line will set the text content of the element.
*/
outputElement.textContent = "Comments are useful for explaining your code!"; // You can also add comments here.
```

Explanation
Comments are ignored by the JavaScript engine and are used purely for documentation.

// starts a single-line comment.

/* and */ enclose a multi-line comment.
Well-commented code is crucial for maintainability and collaboration.

Conclusion: Your Journey Doesn't End Here

Congratulations! You've made it through 100 exercises, and if you've put in the work, you've done far more than just read a book. You've actively built, debugged, and solved problems, which is the true work of a developer. From declaring your first variable to handling asynchronous API calls, you've covered the essential bedrock of modern JavaScript, and for that, you should be incredibly proud.

The journey you've just completed was designed around a principle I've built my entire teaching career on: you can't learn to code from a manual; you learn by creating. Each exercise was a small step, a single building block. Now, looking back, you have a solid foundation and a versatile toolkit at your disposal. You understand how to manipulate the DOM, respond to user events, manage data with arrays and objects, and control the flow of your applications. These aren't just academic concepts—they are the practical skills you will use every single day.

So, What's Next?

The most exciting part of your journey begins now. This book was the training ground; the real adventure is applying what you've learned. Here are my recommendations for where to go from here:

1. Build, Build, Build! The single most important thing you can do now is to start building your own projects. Don't wait for the "perfect" idea. Think of a problem you have, a tool you wish existed, or a simple game you'd like to play, and build it.

- Create a personal portfolio website from scratch.
- Build a to-do list application that saves data to the browser's local storage.
- Fetch data from a free public API (like a weather or movie API) and display it in a clean, user-friendly way. Your first projects won't be perfect, and that's the point. The goal is to face new challenges, search for solutions, and solidify the knowledge you've gained here.

2. Explore the Ecosystem You now have the foundational knowledge to understand modern JavaScript frameworks. These tools provide structure and efficiency for building large, complex applications. Don't feel you need to learn them all at once, but start exploring one:

- **React:** The most popular library for building user interfaces.
- **Vue.js:** Known for its gentle learning curve and excellent documentation.
- **Svelte:** A newer approach that compiles your code to tiny, framework-less vanilla JS.

3. Never Stop Learning The world of web development is always evolving, which is what makes it so exciting. The skills you have now are the gateway to countless other technologies

and concepts. Continue to be curious.

I hope this collection of exercises has served you well. My goal has always been to provide practical, hands-on resources that empower you to create amazing things for the web. This book is just one part of that mission.

The developer community is vast and supportive. Continue to engage with it, ask questions, and share what you've learned. If you're looking for more resources, tutorials, or courses, I invite you to visit my website at **https://basescripts.com/**

You've taken a huge step forward. Now go out there and build something incredible.

All the best,

Laurence Svekis