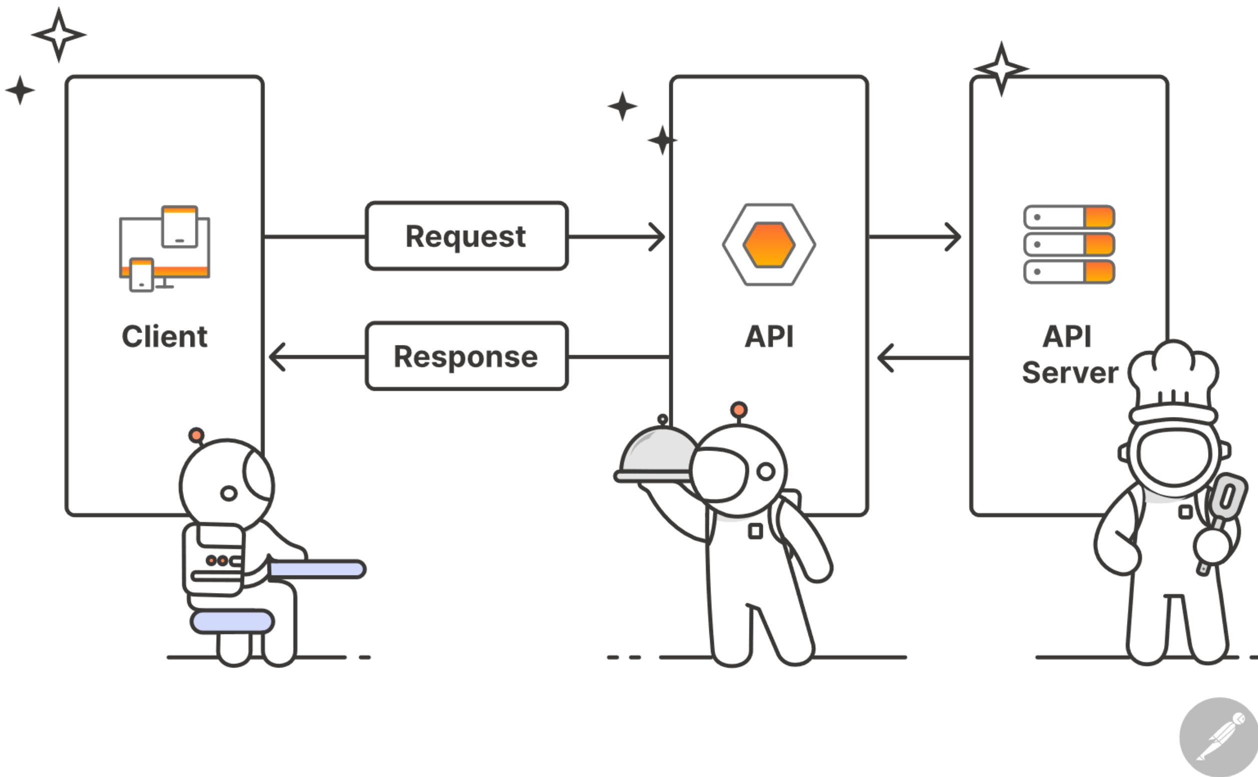
JAVASCRIPT

From Foundations to Advanced Structures

Table of Contents



Introduction ***1***

Purpose 1

Scope 1

Procedure 1

Problem Statement 1

The Console and Dialog Boxes ***2***

The Console 2

Dialog Boxes 2

Var vs Let vs Const ***3***

Data Types ***4***

Type Conversion ***5***

Implicit Conversion (Coercion) 5

Explicit Conversion (Manual) 5

Advanced Type Conversion & Number Methods (Additions) 6

Operators ***7***

Arithmetic Operators 7

Comparison Operators 7

Logical Operators 8

Conditions (Control Flow) ***9***

Loops ***10***

Functions ***11***

Functions Definition 11
Indefinite Arguments (The Rest Parameter) 12
IIFE (Immediately Invoked Function Expression) 12
Function Borrowing (call, apply, bind) 13
Strings *14*
Creating Strings 14
Essential String Methods 14
1. Length and Casing 14
2. Extracting Parts of a String 15
3. Replacing Content 15
4. Converting String to Array 16
5. Searching inside Strings 16
6. String Indexing 16
Regular Expressions (RegExp) *17*
Syntax 17
Error Handling *18*
Date Objects *18*
Arrays *19*
Creating and Accessing 19
Nested Arrays (Multidimensional) 19
Looping Over Arrays 20
Essential Array Methods (Manipulation) 20
Advanced / Functional Array Methods 21
Associative Arrays 23
Object-Oriented JavaScript *24*
Objects 24
Creating Objects 24
Accessing Properties 24
Object Methods 25
Looping Over Objects 25
Objects & Arrays (Passed by Reference) 26
How to Copy Objects Properly (Immutability) 26
The this Keyword 27
Prototypes & Inheritance 27
Classes (ES6 Modern OOP) 28

Basic Class Structure 28
Inheritance 29
Encapsulation (Getters, Setters, Private Fields) 30
Static Methods 31
Abstract Classes 32
Window Object Model 33
Window.location (URL & Routing) 36
Window.history (Session Navigation) 37
Window.navigator (Browser & Device Info) 38
Window.screen (Display Metrics) 39
DOM 40
Selecting Elements 41
Traversing the DOM (Navigating the Tree) 42
Editing Elements 43
Creating and Removing Elements 45
Events 46
Mouse and Keyboard Events 46
Form Events, Window Events, and preventDefault() 47
Event Propagation: Bubbling and Capturing 47
Custom Events 49
Cookies 50
AJAX & JSON 52
JSON 52
AJAX (Asynchronous JavaScript and XML) & fetch() 53
APIs 57
What is an API? (The Restaurant Analogy) 57
Why do we use APIs? 57
How do APIs Work? (The Mechanics) 58
A. The Endpoint (The URL) 58
B. The Method (The "Verb") 58
C. The Headers & Authentication 58
D. The Body (The Payload) 58
Integrated Code Example: A POST Request 59
Conclusion 60
Feedback & Contribution 60

Copyright & Usage 60
Acknowledgments 61

Introduction

Purpose

The purpose of this document is to provide a comprehensive, structured, and practical guide to the core concepts of the JavaScript programming language. It serves as a reference manual for developers, detailing the syntax, logic, and behavior of JavaScript through clear explanations and executable code examples.

Scope

This document covers the fundamental building blocks of JavaScript, proceeding from basic syntax to advanced object-oriented programming.

- **Core Syntax:** Variables, Datatypes, Operators, and Control Flow.
- **Data Structures:** Strings, Arrays, and their associated methods.
- **Logic & Functions:** Loops, Function expressions, Arrow functions, and Closures.
- **Object-Oriented Programming:** Objects, Prototypes, Classes, and Inheritance.
- **Future Scope:** The document will expand to cover Document Object Model (DOM) manipulation, Asynchronous JavaScript (Promises/Async-Await), and modern ES6+ features.

Procedure

Each concept in this document is presented using the following structure:

1. **Concept Definition:** A concise explanation of the feature.
2. **Syntax/Methodology:** The standard way to write the code.
3. **Code Examples:** Practical snippets demonstrating the concept in action.
4. **Output Analysis:** Comments or logs showing exactly what the code produces, ensuring the reader understands the result of execution.

Problem Statement

JavaScript is a loosely typed, dynamic language with many quirks (such as type coercion, hoisting, and prototype-based inheritance) that can be confusing for learners and experienced developers alike. Without a centralized reference that combines theory with practical output, developers often struggle to predict code behavior or debug complex logic errors.

The Console and Dialog Boxes

These are the primary ways to interact with the browser environment and debug code.

The Console

The console is a debugging tool built into web browsers. It allows developers to log messages, inspect variables, and view errors.

```
// 1. Basic Logging
console.log("Hello, World!"); // Output: Hello, World!

// 2. Warning and Error
console.warn("This is a warning."); // Output: (Yellow warning icon) This is a warning.
console.error("This is an error."); // Output: (Red error icon) This is an error.

// 3. Tabular Data (Great for arrays/objects)
const users = [{name: "Alice", age: 25}, {name: "Bob", age: 30}];
console.table(users);
// Output: Displays a neat table with index, name, and age columns.
```

Dialog Boxes

JavaScript has three built-in methods to interact with the user via pop-ups. Note that these stop script execution until the user interacts with them.

```
// 1. Alert: Shows a message with an OK button
alert("Welcome to my website!");

// 2. Prompt: Asks for input (Returns string or null)
let userName = prompt("What is your name?", "Guest");
console.log(userName); // Output: The name typed by user (e.g., "John")

// 3. Confirm: Asks for Yes/No (Returns boolean true/false)
let isSure = confirm("Are you sure you want to delete this?");
console.log(isSure); // Output: true (if OK is clicked) or false (if Cancel is clicked)
```

Var vs Let vs Const

Modern JavaScript (ES6+) introduced let and const to fix issues with the older var. It is best practice to use const by default, let when the value needs to change, and avoid var entirely.

Feature	var	let	const
Scope	Function Scope (ignores code blocks like if or for)	Block Scope { ... }	Block Scope { ... }
Reassignable?	Yes	Yes	No (Immutable reference)
Hoisting	Yes (initialized as undefined)	Yes (but strictly strictly **uninitialized**, causes ReferenceError)	Yes (Uninitialized, causes ReferenceError)

```
// 1. VAR issues
if (true) {
   var x = 10;
}
console.log(x); // Output: 10 (Accessing x outside the 'if' block works, which can cause bugs)

// 2. LET/CONST Scope
if (true) {
   let y = 20;
   const z = 30;
}
// console.log(y); // Error: y is not defined
// console.log(z); // Error: z is not defined

// 3. CONST Immutability
const pi = 3.14;
// pi = 3.14159; // Error: Assignment to constant variable.

// Note: You CAN modify properties of an object declared with const
const user = { name: "Alice" };
user.name = "Bob"; // This is allowed!
```

Data Types

JavaScript variables can hold different types of values. There are two main categories: Primitives and Objects.

These are immutable (cannot be changed) and hold a single value.

```
// String (Text)
let name = "JavaScript";

// Number (Integers and Floats)
let count = 42;
let price = 19.99;

// Boolean (True/False)
let isActive = true;

// Undefined (Variable declared but not assigned)
let notAssigned;
console.log(notAssigned); // Output: undefined

// Null (Intentional absence of value)
let emptyValue = null;

// BigInt (For numbers larger than 2^53 - 1)
let bigNumber = 9007199254740991n;
```

Type Conversion

Sometimes you need to switch between data types (e.g., turning the string "5" into the number 5).

Implicit Conversion (Coercion)

JavaScript automatically converts types in certain contexts.

```
let result = "5" + 2;
console.log(result); // Output: "52" (Number 2 becomes string "2")

let math = "10" - 2;
console.log(math); // Output: 8 (String "10" becomes number 10)
```

Explicit Conversion (Manual)

You manually command JavaScript to change the type.

```
// String to Number
let strNum = "123";
let convertedNum = Number(strNum); // or parseInt(strNum)
console.log(typeof convertedNum); // Output: "number"

// Number to String
let val = 456;
let convertedStr = String(val); // or val.toString()
console.log(typeof convertedStr); // Output: "string"

// Boolean Conversion
console.log(Boolean(1));  // Output: true
console.log(Boolean(0));  // Output: false
console.log(Boolean("")); // Output: false (Empty string is false)
```

Advanced Type Conversion & Number Methods (Additions)

```
// 1. Parsing Integers (parseInt)
// Parses a string and returns a whole number. Stops at the first non-digit.
console.log(parseInt("10"));       // Output: 10
console.log(parseInt("10.99"));    // Output: 10 (Removes decimals, does not round)
console.log(parseInt("15px"));     // Output: 15 (Parses until it hits 'p')
console.log(parseInt("Hello 10")); // Output: NaN (Not a Number)

// 2. Parsing Floats (parseFloat)
// Parses a string and returns a number with decimals.
console.log(parseFloat("10.55"));  // Output: 10.55
console.log(parseFloat("3.14text"));// Output: 3.14

// 3. Checking for Validity (isNaN)
// Checks if a value is "Not-a-Number".
console.log(isNaN("Hello")); // Output: true (String cannot be math)
console.log(isNaN(123));     // Output: false (It is a valid number)

// 4. Checking for Infinity (isFinite)
// Returns true if the value is a finite number.
console.log(isFinite(100));       // Output: true
console.log(isFinite(Infinity));  // Output: false
console.log(isFinite("Hello"));   // Output: false
```

Operators

Arithmetic Operators

Used to perform mathematics.

Operator	Description	Example (x=10, y=5)	Result
+	Addition	x + y	15
-	Subtraction	x - y	5
*	Multiplication	x * y	50
/	Division	x / y	2
%	Modulus (Remainder)	x % 3	1
**	Exponentiation	y ** 2	25
++	Increment	x++	11
--	Decrement	y--	4

Comparison Operators

Used to compare values. Returns true or false.

Operator	Description	Example	Result
==	Equal to (Value only)	5 == "5"	true
===	Strict Equal (Value & Type)	5 === "5"	false
!=	Not Equal	5 != 8	true
!==	Strict Not Equal	5 !== "5"	true
>	Greater than	10 > 5	true
<	Less than	2 < 8	true
>=	Greater or equal	5 >= 5	true

Logical Operators

Used to determine logic between variables.

Operator	Description	Logic	Example	Result
&&	AND	Returns true if **both** are true	true && false	false
||	OR	Returns true if **at least one** is true	true || false	true
!	NOT	Reverses the boolean value	!true	false

Conditions (Control Flow)

Conditions allow code to make decisions based on whether a statement is true or false.

```
let hour = 14; // 2 PM

// 1. If / Else If / Else
if (hour < 12) {
  console.log("Good Morning");
} else if (hour < 18) {
  console.log("Good Afternoon"); // Output: Good Afternoon
} else {
  console.log("Good Evening");
}

// 2. Ternary Operator (Shorthand if/else)
let age = 20;
let access = (age >= 18) ? "Allowed" : "Denied";
console.log(access); // Output: Allowed

// 3. Switch Statement (Good for many specific values)
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("Start of the work week."); // Output matches here
    break;
  case "Friday":
    console.log("Weekend is coming!");
    break;
  default:
    console.log("Just a regular day.");
}
```

Loops

Loops allow you to run the same block of code multiple times.

```
// 1. For Loop (Best when you know how many times to loop)
console.log("--- For Loop ---");
for (let i = 1; i <= 3; i++) {
  console.log("Count: " + i);
}
// Output:
// Count: 1
// Count: 2
// Count: 3

// 2. While Loop (Best when loop depends on a condition)
console.log("--- While Loop ---");
let j = 0;
while (j < 3) {
  console.log("While: " + j);
  j++;
}
// Output:
// Count: 0
// Count: 1
// Count: 2

//3. Do-While Loop
Let r = 0;
do{
console.log(r);
r++;
}while(r<2);
 // Output:
// Count: 1
// Count: 2
```

Functions

Functions are reusable blocks of code designed to perform a particular task.

Functions Definition

```
// 1. Function Declaration (Hoisted - can be used before defined)
function add(a, b) {
   return a + b;
}
console.log(add(5, 3)); // Output: 8

// 2. Function Expression (Not hoisted)
const subtract = function(a, b) {
   return a - b;
};
console.log(subtract(10, 4)); // Output: 6

// 3. Arrow Function (ES6 Modern Syntax)
// Concise syntax, great for one-liners
const multiply = (a, b) => a * b;
console.log(multiply(4, 5)); // Output: 20
```

Indefinite Arguments (The Rest Parameter)

Sometimes you don't know how many arguments a function will receive. The **Rest Parameter** (...) allows you to gather all remaining arguments into an array.

```
// The '...numbers' syntax collects all arguments into an array called 'numbers'
function sumAll(...numbers) {
  let total = 0;
  for (let num of numbers) {
    total += num;
  }
  return total;
}

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

// You can combine it with regular parameters
function raceResults(winner, ...others) {
  console.log(`Winner: ${winner}`);
  console.log(`Others: ${others}`);
}
raceResults("Alice", "Bob", "Charlie", "Dave");
// Output:
// Winner: Alice
// Others: ["Bob", "Charlie", "Dave"]
```

IIFE (Immediately Invoked Function Expression)

An IIFE is a function that runs as soon as it is defined. It is used to create a private scope so variables don't pollute the global environment (very common in older libraries and module patterns).

```
// Syntax: (function definition)(execution)
(function() {
  let secret = "I am safe inside here";
  console.log("IIFE executed immediately!");
})();

// console.log(secret); // Error: secret is not defined (It allows for data privacy)

// Arrow Function IIFE
(() => {
  console.log("Arrow IIFE works too");
})();
```

Function Borrowing (call, apply, bind)

These methods allow you to use a method from one object on a different object. They explicitly set what this refers to.

```
const person1 = {
  fullName: function(city, country) {
    return `${this.firstName} ${this.lastName}, ${city}, ${country}`;
  }
};

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

// 1. call()
// Invokes the function immediately. Arguments are passed individually.
// We "borrow" person1's method to use on person2.
console.log(person1.fullName.call(person2, "Oslo", "Norway"));
// Output: John Doe, Oslo, Norway

// 2. apply()
// Invokes immediately. Similar to call(), but arguments are passed as an ARRAY.
// Useful if your data is already in a list.
console.log(person1.fullName.apply(person2, ["Madrid", "Spain"]));
// Output: John Doe, Madrid, Spain

// 3. bind()
// Does NOT invoke immediately. It returns a NEW function with 'this' permanently bound.
// Useful for event listeners or passing functions around.
const boundFunc = person1.fullName.bind(person2, "Paris", "France");
console.log(boundFunc());
// Output: John Doe, Paris, France
```

Strings

Strings are used to store text. In JavaScript, strings are immutable (methods return a *new* string; they do not change the original).

Creating Strings

```
// Single or Double quotes
let firstName = 'John';
let lastName = "Doe";

// Template Literals (Backticks `) - BEST PRACTICE
// Allows inserting variables directly using ${}
let greeting = `Hello, ${firstName} ${lastName}!`;
console.log(greeting); // Output: Hello, John Doe!
```

Essential String Methods

1. Length and Casing

```
let text = "  Hello World  ";

// Length property (Not a function, so no parenthesis)
console.log(text.length); // Output: 15

// Changing Case
console.log(text.toUpperCase()); // Output: "  HELLO WORLD  "
console.log(text.toLowerCase()); // Output: "  hello world  "

// Removing Whitespace
console.log(text.trim()); // Output: "Hello World" (Removes spaces from start and end)
```

2. Extracting Parts of a String

```
let str = "JavaScript";

// slice(start, end) - Extracts part of a string
// 'end' is usually not included
console.log(str.slice(0, 4)); // Output: "Java"
console.log(str.slice(4));    // Output: "Script" (If end is omitted, goes to the end)

// substring(start, end) - Similar to slice, but handles negative numbers differently
console.log(str.substring(0, 4)); // Output: "Java"
```

---

3. Replacing Content

```
let message = "Visit Microsoft";

// replace(searchFor, replaceWith) - Replaces only the FIRST match
let newMessage = message.replace("Microsoft", "Google");
console.log(newMessage); // Output: "Visit Google"

// replaceAll() - Replaces ALL matches
let text2 = "Cats are cute. I love cats.";
// Note: It is case-sensitive ("Cats" vs "cats")
console.log(text2.replaceAll("cats", "dogs"));
// Output: "Cats are cute. I love dogs."
```

4. Converting String to Array

```
let csv = "Apple,Banana,Orange";

// split(separator)
let fruits = csv.split(",");
console.log(fruits); // Output: ["Apple", "Banana", "Orange"]
```

---

5. Searching inside Strings

```
let sentence = "The quick brown fox";

// indexOf() - Returns position of first match, or -1 if not found
console.log(sentence.indexOf("brown")); // Output: 10
console.log(sentence.indexOf("green")); // Output: -1

// includes() - Returns true/false
console.log(sentence.includes("fox")); // Output: true

// startsWith() / endsWith()
console.log(sentence.startsWith("The")); // Output: true
console.log(sentence.endsWith("dog"));   // Output: false
```

---

6. String Indexing

```
let str = "Hello World";

// 1. charAt(index) - Returns the character at a specific position
console.log(str.charAt(0)); // Output: "H"
console.log(str.charAt(6)); // Output: "W"

// 2. indexOf(searchValue) - Returns the index of the FIRST occurrence
// Returns -1 if not found. Case sensitive.
console.log(str.indexOf("o")); // Output: 4 (The 'o' in Hello)

// 3. lastIndexOf(searchValue) - Returns the index of the LAST occurrence
// Searches backwards from the end.
console.log(str.lastIndexOf("o")); // Output: 7 (The 'o' in World)
```

Regular Expressions (RegExp)

Regular expressions are patterns used to match character combinations in strings. They are powerful for search and validation (like checking emails).

Syntax

/pattern/modifiers;

- **Modifiers:**
    - i: Case-insensitive matching.
    - g: Global match (find all matches, not just the first).

```
let text = "Visit W3Schools and w3schools!";
let pattern = /w3schools/ig; // i = insensitive, g = global

// 1. search() - Returns index of match
let position = text.search(/w3schools/i);
console.log(position); // Output: 6

// 2. match() - Returns an array of matches
let matches = text.match(pattern);
console.log(matches); // Output: ['W3Schools', 'w3schools']

// 3. replace() with Regex
let newText = text.replace(/w3schools/i, "Google");
console.log(newText); // Output: "Visit Google and w3schools!"

// 4. test() - Returns true/false (Best for validation)
let emailPattern = /@/;
console.log(emailPattern.test("user@gmail.com")); // Output: true
console.log(emailPattern.test("usergmail.com"));  // Output: false
```

Error Handling

Code can fail due to programmer error, wrong input, or unforeseen issues. We use try...catch to handle this gracefully so the script doesn't crash.

```
try {
   // Block of code to try
   let x = y + 10; // 'y' is not defined, so this throws an error
   console.log("This line will never run");

} catch (error) {
   // Block of code to handle errors
   console.error("An error occurred: " + error.message);
   // Output: An error occurred: y is not defined

} finally {
   // Block of code to run regardless of the result
   console.log("Validation complete.");
}

// Throwing Custom Errors
function checkAge(age) {
   if (age < 18) {
      throw new Error("Too young"); // Creates a custom error
   }
   return "Access Granted";
}
```

Date Objects

JavaScript stores dates as the number of milliseconds since January 1, 1970.

```
// 1. Creating Dates
const now = new Date(); // Current date and time
const specificDate = new Date("2023-12-25"); // YYYY-MM-DD

// 2. Get Methods (Extracting info)
// Note: Months are 0-indexed (January is 0, December is 11)
console.log(now.getFullYear()); // Output: 2023 (or current year)
console.log(now.getMonth());    // Output: 0-11
console.log(now.getDay());      // Output: 0-6 (0 is Sunday)

// 3. Set Methods (Changing info)
specificDate.setFullYear(2025);
console.log(specificDate); // Output: Thu Dec 25 2025...
```

Arrays

An array is a special variable that can hold more than one value at a time.

Creating and Accessing

```
// Literal Syntax (Recommended)
let cars = ["BMW", "Volvo", "Audi"];

// Constructor Syntax (Avoid using)
// let cars = new Array("BMW", "Volvo", "Audi");

// Accessing Elements
console.log(cars[0]); // Output: BMW
console.log(cars[cars.length - 1]); // Output: Audi (Last element)

// Modifying Elements
cars[0] = "Toyota";
console.log(cars); // Output: ["Toyota", "Volvo", "Audi"]
```

Nested Arrays (Multidimensional)

Arrays can contain other arrays.

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log(matrix[0][0]); // Output: 1 (Row 0, Col 0)
console.log(matrix[1][2]); // Output: 6 (Row 1, Col 2)
```

Looping Over Arrays

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

// 1. Standard For Loop
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}

// 2. For...Of Loop (Modern & Clean)
for (let fruit of fruits) {
  console.log(fruit);
}

// 3. forEach() Method
fruits.forEach(function(fruit, index) {
  console.log(`${index}: ${fruit}`);
});
```

Essential Array Methods (Manipulation)

These are the bread and butter of array management.

Method	Description	Example
push()	Adds to end	arr.push("New")
pop()	Removes from end	arr.pop()
unshift()	Adds to start	arr.unshift("First")
shift()	Removes from start	arr.shift()
splice()	Adds/Removes at specific index	arr.splice(2, 0, "Item")
slice()	Copies a portion (non-destructive)	arr.slice(1, 3)

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

nums.push(4);   // [1, 2, 3, 4]
nums.pop();     // [1, 2, 3]
nums.unshift(0); // [0, 1, 2, 3]
nums.shift();   // [1, 2, 3]

// Splice: (Start index, How many to remove, What to add)
let colors = ["Red", "Green", "Blue"];
colors.splice(1, 0, "Yellow");
console.log(colors); // ["Red", "Yellow", "Green", "Blue"]

// Slice: (Start, End - not included)
let fewColors = colors.slice(0, 2);
console.log(fewColors); // ["Red", "Yellow"]
```

Advanced / Functional Array Methods

```
const numbers = [10, 20, 30, 40];

// 1. map() - Transforms every element and returns a NEW array
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [20, 40, 60, 80]

// 2. filter() - Returns elements that match a condition
const over25 = numbers.filter(num => num > 25);
console.log(over25); // Output: [30, 40]

// 3. reduce() - Reduces array to a single value (e.g., sum), reduceright() starts from right to left
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // Output: 100

// 4. find() - Returns the FIRST element that matches
const found = numbers.find(num => num > 25);
console.log(found); // Output: 30

// 5. sort() - Sorts the array IN PLACE
let messy = [3, 1, 10, 2];
// Note: Standard sort() treats numbers as strings ("10" comes before "2")
// Fix by providing a compare function:
messy.sort((a, b) => a - b);
console.log(messy); // Output: [1, 2, 3, 10]

// 6. reverse() reverses array
Let reversed = [3, 1, 7, 17];
reversed.reverse(); // Output: [17, 7, 1, 3]
```

```
const ages = [32, 33, 16, 40];

// 7. every() - Checks if ALL elements pass a test
// Returns true only if EVERY element meets the condition.
const allAdults = ages.every(age => age >= 18);
console.log(allAdults); // Output: false (because 16 is present)

// 8. some() - Checks if AT LEAST ONE element passes a test
// Returns true if ANY element meets the condition.
const hasAdult = ages.some(age => age >= 18);
console.log(hasAdult); // Output: true

// 9. includes() - Checks if an array contains a specific value
// Returns true/false. (Simpler than indexOf for boolean checks)
const fruits = ["Apple", "Banana", "Mango"];
console.log(fruits.includes("Banana")); // Output: true
console.log(fruits.includes("Grape"));  // Output: false

// 10. flat() - Flattens nested arrays into a single array
const nested = [1, [2, 3], [4, [5, 6]]];
console.log(nested.flat());      // Output: [1, 2, 3, 4, [5, 6]] (Default depth is 1)
console.log(nested.flat(2));     // Output: [1, 2, 3, 4, 5, 6] (Depth of 2)
console.log(nested.flat(Infinity)); // Flattens completely regardless of depth

// 11. flatMap() - Combines map() and flat() (Depth 1 only)
// It is slightly more efficient than calling map() then flat().
const sentence = ["Hello World", "Bye World"];
const words = sentence.flatMap(text => text.split(" "));
console.log(words); // Output: ["Hello", "World", "Bye", "World"]
```


Made By Youssef Amgad Elkhatib

Associative Arrays

Important: JavaScript does **not** support Associative Arrays (arrays with named indexes) like PHP or Python Dictionaries.

- If you use named indexes (arr["name"] = "John"), JavaScript will redefine the array as a standard **Object**.
- The array methods and properties (like .length or .push) will stop working correctly.

Use Objects instead:

```
let badArray = [];
badArray["firstName"] = "John"; // Length remains 0

// DO THIS (Object):
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 46
};
console.log(person.firstName); // Access via dot notation
```

Object-Oriented JavaScript

Objects

In JavaScript, almost "everything" is an object. Objects are collections of key-value pairs where the values can be properties (data) or methods (functions).

Creating Objects

There are two main ways to create objects.

```
// 1. Object Literal (Preferred)
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  isEmployed: true
};

// 2. New Object Keyword (Avoid if possible)
const car = new Object();
car.brand = "Ford";
car.model = "Mustang";
```

Accessing Properties

You can access object properties using dot notation or bracket notation.

```
console.log(person.firstName); // Output: John (Standard)

// Bracket Notation is required if the key has spaces or is a variable
// person["first name"] // Works if key has a space
let key = "age";
console.log(person[key]); // Output: 50
```

Object Methods

Methods are functions stored as object properties.

```
const calculator = {
  add: function(a, b) {
    return a + b;
  },
  // Modern ES6 Shorthand (Preferred)
  subtract(a, b) {
    return a - b;
  }
};

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

Looping Over Objects

Since objects are not ordered like arrays, we use specific loops.

```
const user = { name: "Alice", role: "Admin", id: 101 };

// 1. For...In Loop (Iterates over KEYS)
for (let key in user) {
  console.log(`${key}: ${user[key]}`);
}
// Output:
// name: Alice
// role: Admin
// id: 101

// 2. Object.keys() / Object.values()
// Returns an array of keys or values
const keys = Object.keys(user);
console.log(keys); // Output: ["name", "role", "id"]
```

Objects & Arrays (Passed by Reference)

Objects and Arrays are stored in the **Heap** (memory), and the variable only holds a **reference** (address) to that location. When you assign a variable to another, you copy the **reference**, not the actual object.

```
const user1 = { name: "Alice" };
const user2 = user1; // Both point to the SAME object in memory!

user2.name = "Bob"; // Modify via user2

console.log(user1.name); // Output: Bob (user1 is also changed!)
```

How to Copy Objects Properly (Immutability)

To avoid the reference issue, you must create a genuine copy (clone) of the object.

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

// 1. Spread Operator (Shallow Copy - Best for simple objects)
const copy = { ...original };
copy.a = 99;

console.log(original.a); // Output: 1 (Unchanged)
console.log(copy.a);     // Output: 99

// 2. structuredClone() (Deep Copy - Modern & Best for nested objects)
const nested = {
  data: { id: 1 }
};
const deepCopy = structuredClone(nested);
```

The this Keyword

this refers to the current execution context. Its value changes depending on how a function is called.

```
const hero = {
  name: "Batman",
  greet() {
    // 'this' refers to the 'hero' object
    console.log(`I am ${this.name}`);
  }
};
hero.greet(); // Output: I am Batman

// Arrow Functions behave differently!
// They inherit 'this' from the surrounding scope (usually Window/Global)
const villain = {
  name: "Joker",
  laugh: () => {
    console.log(this.name); // 'this' is NOT the object here
  }
};
villain.laugh(); // Output: undefined (or Window name)
```

Prototypes & Inheritance

JavaScript uses **Prototypal Inheritance**. Every object has a hidden link to another object called its **prototype**. If a property isn't found on the object itself, JS looks up the prototype chain.

```
const animal = {
  eats: true
};

const rabbit = Object.create(animal); // Rabbit inherits from Animal
rabbit.jumps = true;

console.log(rabbit.jumps); // true (Own property)
console.log(rabbit.eats);  // true (Inherited from prototype)
```

Classes (ES6 Modern OOP)

Classes are "syntactic sugar" over JavaScript's existing prototype-based inheritance. They provide a cleaner, more familiar syntax for creating objects and handling inheritance.

Basic Class Structure

```
class Car {
   // The Constructor runs immediately when 'new' is called
   constructor(brand, year) {
      this.brand = brand;
      this.year = year;
   }

   // Method (Notice: no 'function' keyword needed)
   displayInfo() {
      return `${this.brand} is from ${this.year}`;
   }
}

const myCar = new Car("Toyota", 2022);
console.log(myCar.displayInfo()); // Output: Toyota is from 2022
```

Inheritance

Classes can inherit properties and methods from other classes.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

// Child Class
class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Call the parent constructor first
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks!`); // Method Overriding
  }
}

const d = new Dog("Rex", "German Shepherd");
d.speak(); // Output: Rex barks!
```

Encapsulation (Getters, Setters, Private Fields)

Modern JS allows private fields (start with #) that cannot be accessed from outside the class.

```
class BankAccount {
  #balance = 0; // Private Field

  constructor(owner) {
    this.owner = owner;
  }

  // Setter
  deposit(amount) {
    if (amount > 0) {
      this.#balance += amount;
    }
  }

  // Getter
  get balance() {
    return `Balance: $${this.#balance}`;
  }
}

const account = new BankAccount("Alice");
account.deposit(100);
console.log(account.balance); // Output: Balance: $100

// console.log(account.#balance); // Error: Private field '#balance' must be declared in an enclosing class
```

Static Methods

Static methods are called on the class itself, not on instances of the class. They are often used for utility functions.

```
class MathUtils {
  static add(x, y) {
    return x + y;
  }
}

console.log(MathUtils.add(5, 5)); // Output: 10
// const m = new MathUtils();
// m.add(5,5); // Error: m.add is not a function
```

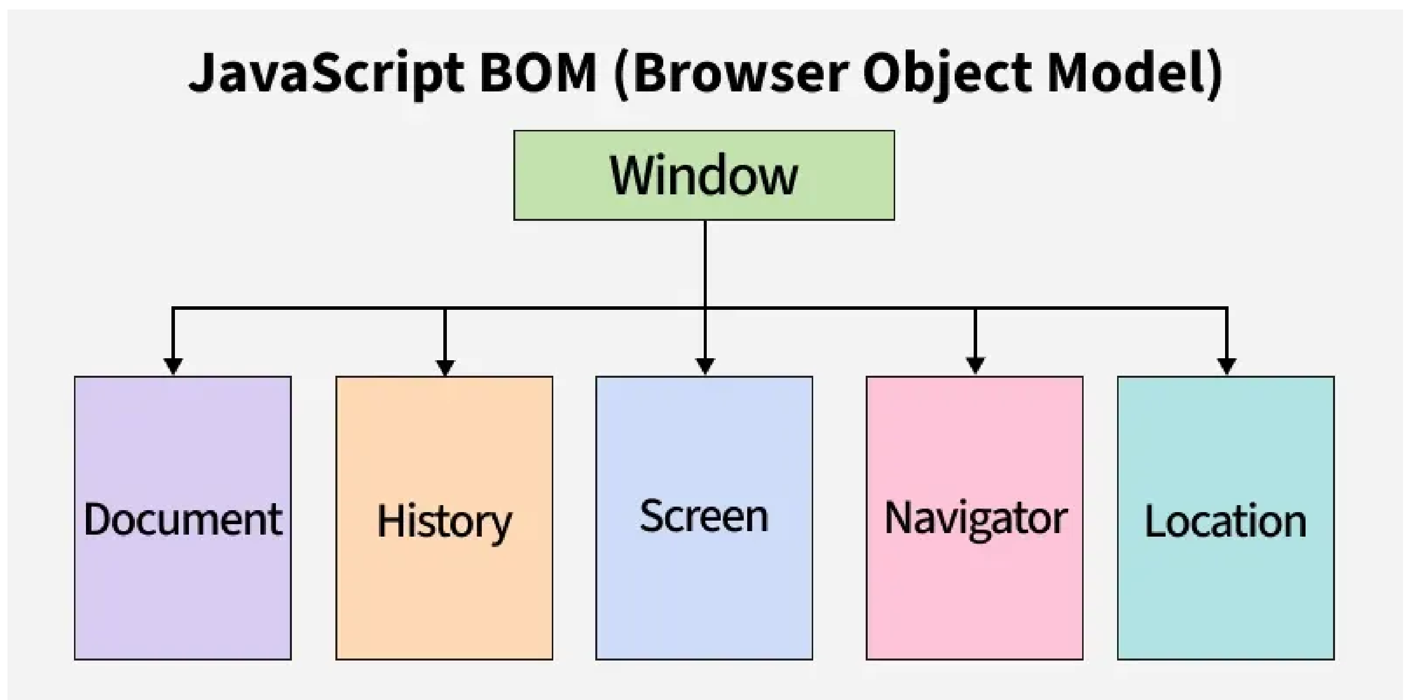
Abstract Classes

Unlike languages like Java or C#, JavaScript does not have a built-in abstract keyword. However, we can simulate abstract classes to ensure a base class is never instantiated directly and that specific methods must be implemented by child classes.

```
class Animal {
   constructor(name) {
      if (this.constructor === Animal) {
         throw new Error("Abstract classes cannot be instantiated.");
      }
      this.name = name;
   }

   // Abstract Method Simulation
   makeSound() {
      throw new Error("Method 'makeSound()' must be implemented.");
   }
}

class Dog extends Animal {
   makeSound() {
      console.log("Woof!");
   }
}

// const a = new Animal("Generic"); // Error: Abstract classes cannot be instantiated.
const d = new Dog("Buddy");
d.makeSound(); // Output: Woof!
```

Window Object Model

The window object is the global object in a browser environment. It represents the browser window (or tab) that contains the **DOM** document. Within this window object are several highly useful sub-objects: **location, history, navigator, and screen**.



- **Key Methods**
  - **window.open(url, name, specs):** Opens a new browser window.
    - **url:** The web address to load.
    - **name:** The target attribute or name of the window (e.g., "_blank" for a new tab).
    - **specs:** A comma-separated string of window features (size, position, whether to show scrollbars). *If you provide sizing specs, the browser will open a separate popup window instead of a new tab.*
  - **window.close():** Closes the current window. (Remember, this generally only works on windows that were opened by a script).
  - **window.resizeTo(width, height)**: Resizes the window to an **absolute** specific width and height in pixels. (e.g., "Make this window exactly 500x500").
  - **window.resizeBy(widthDelta, heightDelta)**: Resizes the window **relative** to its current size. (e.g., "Make this window 50 pixels wider and 20 pixels shorter than whatever it is right now").

- **window.moveTo(x, y)**: Moves the top-left corner of the window to an **absolute** coordinate on the user's screen. (e.g., moveTo(0, 0) pushes the window to the absolute top-left corner of the monitor).
- **window.moveBy(xDelta, yDelta)**: Moves the window **relative** to its current position. (e.g., "Move this window 50 pixels to the right and 50 pixels down from where it is currently sitting").

```
document.addEventListener("DOMContentLoaded", () => {

  // We need a variable to store the reference to the new window we create
  let myPopup = null;

  // 1. OPEN WINDOW
  document.getElementById("btn-open").addEventListener("click", () => {
    // url, name, specs (setting width/height forces a popup instead of a tab)
    myPopup = window.open(
      "https://example.com",
      "ChildWindow",
      "width=300,height=300,left=200,top=200"
    );
  });

  // Helper function to check if the popup actually exists before sending commands
  function isPopupValid() {
    if (!myPopup || myPopup.closed) {
      alert("Please open the popup window first!");
      return false;
    }
    return true;
  }

  // 2. RESIZE TO (Absolute)
  document.getElementById("btn-resize-to").addEventListener("click", () => {
    if (isPopupValid()) {
      myPopup.resizeTo(400, 400); // Forces exact size: 400px by 400px
    }
  });

  // 3. RESIZE BY (Relative)
  document.getElementById("btn-resize-by").addEventListener("click", () => {
    if (isPopupValid()) {
      myPopup.resizeBy(50, 50); // Adds 50px to current width and height
    }
  });

  // 4. MOVE TO (Absolute)
  document.getElementById("btn-move-to").addEventListener("click", () => {
    if (isPopupValid()) {
      myPopup.moveTo(0, 0); // Pushes window to exact coordinates (0,0) on the monitor
    }
```

```
    });

    // 5. MOVE BY (Relative)
    document.getElementById("btn-move-by").addEventListener("click", () => {
      if (isPopupValid()) {
        myPopup.moveBy(100, 100); // Shoves window 100px right and 100px down
      }
    });

    // 6. CLOSE WINDOW
    document.getElementById("btn-close").addEventListener("click", () => {
      if (isPopupValid()) {
        myPopup.close(); // Kills the popup window
        myPopup = null;  // Clear the reference
      }
    });

  });
```

Window.location (URL & Routing)

The location object contains information about the current URL and provides methods to redirect the user or reload the page.

- **Key Properties:**
  - **href:** The entire URL (e.g., "https://www.example.com:8080/path?query=1").
  - **hostname:** The domain name ("www.example.com").
  - **pathname:** The path after the domain ("/path").
  - **search:** The query string ("?query=1").
- **Key Methods:**
  - **assign(url):** Navigates to a new URL (saves the current page in history).
  - **replace(url):** Navigates to a new URL **without** saving the current page in history (the user can't click "Back" to return to it).
  - **reload():** Refreshes the current page.

```
console.log("Full URL:", window.location.href);
console.log("Domain:", window.location.hostname);
console.log("Path:", window.location.pathname);
console.log("Query String:", window.location.search);

// --- Using Location Methods ---
function refreshPage() {
  // Refreshes the current page
  window.location.reload();
}

function goToNewPage() {
  // Navigates to a new URL and saves the current page in the browser's history
  window.location.assign("https://developer.mozilla.org");
}


function replaceCurrentPage() {
  // Navigates to a new URL WITHOUT saving the current page in history
  // (User cannot click 'Back' to return to the previous page)
  window.location.replace("https://developer.mozilla.org");
}
```

Window.history (Session Navigation)

The history object allows you to interact with the browser's session history for that specific tab. It is heavily used in Single Page Applications (SPAs) like React or Vue to change the URL without reloading the page.

- **Key Properties:**
  - **length:** The number of elements in the session history.
- **Key Methods:**
  - **back():** Goes to the previous page (equivalent to clicking the browser's Back button).
  - **forward():** Goes to the next page.
  - **go(n):** Moves n steps in the history (e.g., history.go(-2) goes back two pages).
  - **pushState() / replaceState():** Advanced methods used to change the URL in the address bar without triggering a page refresh.

```
// --- Reading History Properties ---
console.log("Pages in session history:", window.history.length);

// --- Using History Methods ---
function goBackOnePage() {
  // Equivalent to clicking the browser's Back button
  window.history.back();
}

function goForwardOnePage() {
  // Equivalent to clicking the browser's Forward button
  window.history.forward();
}

function jumpBackTwoPages() {
  // Moves back 2 steps in history
  window.history.go(-2);
}

function changeUrlWithoutReloading() {
  // Parameters: state object, title (usually ignored by browsers), new URL
  window.history.pushState({ page: "dashboard" }, "", "/dashboard");
  console.log("URL changed to /dashboard without refreshing the page!");
}
```

Window.navigator (Browser & Device Info)

The navigator object provides information about the user's browser, operating system, and device capabilities. It is useful for feature detection and analytics.

- **Key Properties:**
  - **userAgent:** A string representing the browser and OS (often used, but easily spoofed).
  - **language:** The user's preferred language (e.g., "en-US").
  - **onLine:** A boolean indicating if the browser has an internet connection.
  - **cookieEnabled:** A boolean indicating if cookies are allowed.
- **Key Methods:**
  - **geolocation.getCurrentPosition():** Prompts the user for their GPS coordinates.
  - **clipboard.writeText():** Allows you to copy text to the user's clipboard.

```
// --- Reading Navigator Properties ---
console.log("Browser & OS Info:", window.navigator.userAgent);
console.log("Preferred Language:", window.navigator.language);
console.log("Is Online?", window.navigator.onLine);
console.log("Cookies Enabled?", window.navigator.cookieEnabled);

// --- Using Navigator Methods ---
function getUserLocation() {
  // Check if the browser supports geolocation
  if (window.navigator.geolocation) {
    window.navigator.geolocation.getCurrentPosition(
      (position) => {
        console.log("Latitude:", position.coords.latitude);
        console.log("Longitude:", position.coords.longitude);
      },
      (error) => {
        console.error("Location access denied or failed.", error);
      }
    );
  }
}

function copyToClipboard(text) {
  // Check if the browser supports the Clipboard API
  if (window.navigator.clipboard) {
    window.navigator.clipboard.writeText(text)
      .then(() => console.log("Text successfully copied!"))
      .catch(err => console.error("Failed to copy text:", err));
  }
}
```

Window.screen (Display Metrics)

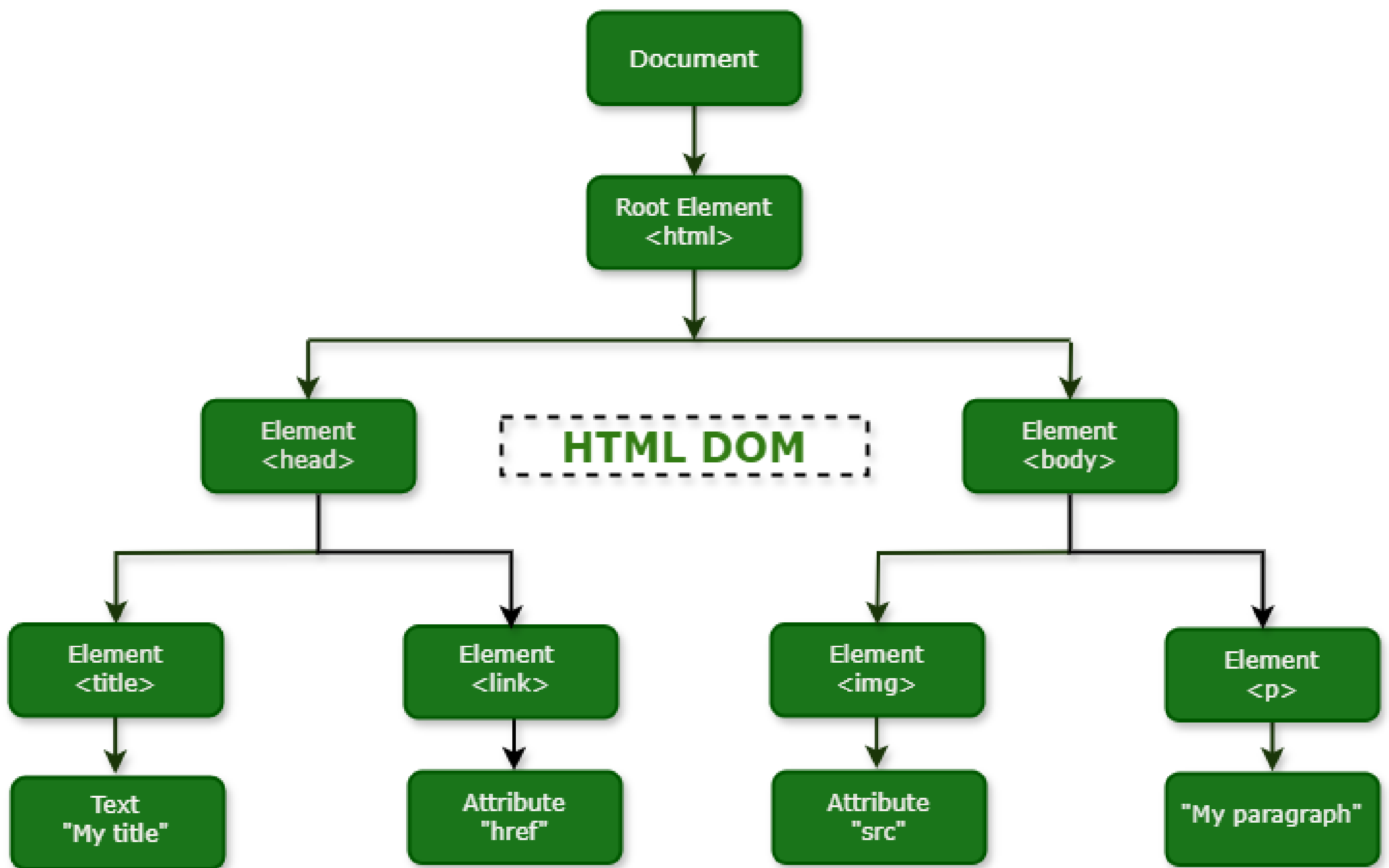
The screen object provides information about the user's physical monitor or display. It is strictly about the hardware screen, not the browser window size (for browser size, you use window.innerWidth and innerHeight).

- **Key Properties:**
  - **width / height:** The total resolution of the screen.
  - **availWidth / availHeight:** The resolution available to the browser (excludes the Windows taskbar or Mac dock).
  - **colorDepth:** The bit depth of the color palette (usually 24 or 32).

```
// --- Reading Screen Properties ---

// The total physical resolution of the user's monitor
console.log("Total Screen Width:", window.screen.width, "px");
console.log("Total Screen Height:", window.screen.height, "px");

// The resolution actually available to the browser (subtracts the OS taskbar/dock)
console.log("Available Width:", window.screen.availWidth, "px");
console.log("Available Height:", window.screen.availHeight, "px");

// The color depth of the user's screen
console.log("Color Depth:", window.screen.colorDepth, "bits per pixel");
```

DOM

The **DOM** is a massive object called document (which is actually a property of the window object, **window.document**). The browser takes your raw HTML and turns it into a giant, branching "tree" of objects (nodes) that JavaScript can read, change, delete, or add to.



To make these code examples easy to follow, imagine we are working with this simple HTML structure:

```
<div id="task-board">
  <h2 class="title">Task Manager</h2>
  <ul id="task-list">
    <li class="task urgent">Buy groceries</li>
    <li class="task">Clean room</li>
  </ul>
</div>
```

Selecting Elements

Before you can change an element, you have to grab it from the DOM.

- **The Modern Way (Highly Recommended):**
  - **document.querySelector('selector'):** Grabs the **first** element that matches a CSS selector (e.g., .my-class, #my-id, button).
  - **document.querySelectorAll('selector'):** Grabs **all** matching elements and returns them in a NodeList (which you can loop through using forEach).
- **The Classic Methods:**
  - **document.getElementById('id'):** Extremely fast, grabs a single element by its ID.
  - **document.getElementsByClassName('class'):** Returns a live HTMLCollection of all elements with that class.
  - **document.getElementsByTagName('div'):** Returns a live HTMLCollection of all <div> elements.

```
// --- Selecting Elements ---

// 1. The Modern Way (CSS Selectors)
const mainBoard = document.querySelector("#task-board"); // Grabs by ID
const firstTask = document.querySelector(".task");      // Grabs the FIRST element with this class
const allTasks = document.querySelectorAll(".task");    // Grabs ALL elements with this class (NodeList)

// Looping through the NodeList we just selected
allTasks.forEach(task => {
  console.log("Found task:", task.textContent);
});

// 2. The Classic Methods
const taskList = document.getElementById("task-list");
const urgentTasks = document.getElementsByClassName("urgent"); // Live HTMLCollection
```

Traversing the DOM (Navigating the Tree)

Sometimes you select an element, but what you actually want is its parent or its neighbor.

- **Going Up: element.parentElement** (grabs the container holding the element).
- **Going Down: element.children** (gets all child elements inside it), **element.firstElementChild**, **element.lastElementChild**.
- **Going Sideways: element.nextElementSibling, element.previousElementSibling** (grabs the next or previous element on the same level).

```
// --- Traversing the DOM ---

// Let's start with a specific element we already selected
const firstListItem = document.querySelector(".task"); // <li class="task urgent">Buy groceries</li>

// Going UP the tree
const parentList = firstListItem.parentElement;
console.log("Parent element:", parentList.id); // Outputs: "task-list"

// Going DOWN the tree
const listChildren = parentList.children;
console.log("Number of children:", listChildren.length); // Outputs: 2
console.log("Last child:", parentList.lastElementChild.textContent); // Outputs: "Clean room"

// Going SIDEWAYS (Siblings)
const secondListItem = firstListItem.nextElementSibling;
console.log("Next sibling:", secondListItem.textContent); // Outputs: "Clean room"
```

Editing Elements

Once you have an element in your JavaScript variables, you can manipulate its content, attributes, and styles.

- **Changing Content:**
  - **element.textContent:** : Gets/sets all text, including hidden text, without HTML tags.
  - **element.innerHTML:** Gets/sets HTML markup, including tags and text.
  - **element.outerHtml:** get or set the entire HTML markup of a specified element, including the element's own tag, its attributes, and all of its descendants.
  - **element.innerText:** Gets/sets only the "rendered" (visible) text, honoring CSS styles.
- **Changing Attributes:**
  - **element.setAttribute**('src', 'image.jpg'): Changes or adds an attribute.
  - **element.getAttribute**('href'): Reads an attribute's current value.
  - **element.removeAttribute**('disabled'): Deletes an attribute.
- **Changing Styles & Classes:**
  - **element.style.color** = "red": Applies inline CSS. (Usually, you want to avoid this and use classes instead).
  - **element.className =** "light": Sets class of the element to light
  - **element.classList.add**('active'): Adds a CSS class.
  - **element.classList.remove**('hidden'): Removes a CSS class.
  - **element.classList.toggle**('dark-mode'): Adds the class if it's missing, or removes it if it's already there.

```
// --- Editing Elements ---
const titleElement = document.querySelector(".title");
const secondTask = document.querySelector("#task-list").lastElementChild;
// Changing text content
titleElement.textContent = "Super Task Manager Pro";

// Changing classes (Best practice for styling)
secondTask.classList.add("completed");
secondTask.classList.remove("urgent"); // Fails silently if the class isn't there, which is perfectly safe
secondTask.classList.toggle("highlight"); // Adds 'highlight' since it doesn't have it yet

// Changing attributes
titleElement.setAttribute("data-status", "active");
console.log("Status is:", titleElement.getAttribute("data-status"));
// Changing inline styles (Use sparingly)
titleElement.style.color = "blue";
titleElement.style.fontSize = "24px"; // Notice we use camelCase in JS for CSS properties
```

Creating and Removing Elements

You don't just have to edit what is already there; you can build entirely new UI components from scratch.

- **Creating:**
  - **document.createElement('div')** creates a new element in memory, but it doesn't appear on the page yet.
  - **document.createTextNode('this is simple text')** creates a text node that can then be put inside the tags.
- **Appending (Adding to the page):**
  - **parentElement.appendChild(newElement):** Drops the new element at the very end of the parent container.
  - **parentElement.prepend(newElement):** Drops the new element at the very beginning of the parent container.
  - **document.insertBefore(text element, position):** inserts the text element before the specified position.
- **Removing:** **element.remove()** deletes the element from the DOM entirely.

```
// --- Creating and Removing Elements ---

const listContainer = document.getElementById("task-list");

// 1. CREATING an element
const newListItem = document.createElement("li");

// 2. Editing our new element before putting it on the page
newListItem.textContent = "Pay electricity bill";
newListItem.classList.add("task", "urgent");
newListItem.setAttribute("data-id", "99");

// 3. ADDING it to the DOM
// appendChild puts it at the BOTTOM of the list
listContainer.appendChild(newListItem);

// 4. REMOVING an element
const firstTaskToRemove = listContainer.firstElementChild;
firstTaskToRemove.remove(); // Deletes "Buy groceries" from the page entirely
```

Events

Events are the beating heart of interactive JavaScript. If the DOM is the physical structure of your webpage, Events are the nervous system. They allow your code to listen for and react to things happening on the screen—whether it is a user clicking a button, typing on their keyboard, or the browser finishing a page load.

Whenever an event fires, the browser automatically creates an **Event Object** (usually written as e or event in the function). This object is packed with details about exactly what just happened (e.g., which key was pressed, or the exact X/Y coordinates of the mouse).

Mouse and Keyboard Events

These are the most common interactions you will program for.

- **Mouse Events:**
  - **click:** Fires when a mouse button is pressed and released on an element.
  - **dblclick:** Fires on a double-click.
  - **mouseenter / mouseleave:** Fires when the mouse pointer enters or leaves an element's physical space (does not bubble).
  - **mousemove:** Fires continuously as the mouse moves over an element.
- **Keyboard Events:**
  - **keydown:** Fires the moment a key is pressed down. (This is the standard, most useful keyboard event).
  - **keyup:** Fires when the key is released.
  - ***Inside the Event Object:*** You can use e.key to find out exactly which character was typed (e.g., "Enter", "a", "Escape").

- **onclick** only allows **one** function per element (assigning a new one overwrites the old), whereas **addEventListener** lets you safely attach **multiple** functions to the exact same event without overwriting anything.

Form Events, Window Events, and preventDefault()

Forms and the browser window have their own unique set of events.

- **Form Events:**
    - **submit:** Fires when a <form> is submitted.
    - **input:** Fires every time the value of an <input>, <textarea>, or <select> changes (fires instantly on every keystroke).
    - **change:** Fires when an input loses focus (the user clicks away) *after* its value has been changed.
- **Preventing Default Behavior (e.preventDefault()):**
    - Browsers have baked-in, default reactions to certain events. If you click a link (<a>), the browser navigates to a new page. If you submit a <form>, the browser refreshes the page and sends a GET/POST request.
    - Calling **e.preventDefault()** tells the browser: *"Stop doing your default action. I am going to handle this myself with JavaScript."* This is crucial for building modern Single Page Applications.

Event Propagation: Bubbling and Capturing

When you click on a nested element (like a <button> inside a <div> inside a <section>), you aren't just clicking the button. You are clicking everything underneath it, too.

The browser handles this via a process called **Event Propagation**, which has two main phases:

1. **The Capturing Phase (Going Down):** The event starts at the top of the DOM (the window) and trickles *down* through the ancestors until it hits your target. (By default, JavaScript ignores this phase).
2. **The Bubbling Phase (Going Up):** The event hits the target, and then bubbles *back up* through the ancestors, triggering any click listeners attached to those parents along the way. (This is the default behavior in JS).

How to control it:

- **e.stopPropagation():** Stops the event from bubbling up any further. It tells the parents, *"I handled this, do not trigger your event listeners."*
- **{ capture: true }:** A third argument you can pass to addEventListener to force it to trigger during the Capturing phase instead of the Bubbling phase.

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    const grandparent = document.getElementById("grandparent");
    const parent = document.getElementById("parent");
    const child = document.getElementById("child");
    const log = document.getElementById("log");
    const stopPropCheckbox = document.getElementById("stop-prop");

    function logEvent(message) {
      log.innerHTML += message + "<br>";
    }

    document.getElementById("clear-log").addEventListener("click", () => log.innerHTML = "");

    // --- BUBBLING PHASE LISTENERS (Default behavior - Goes UP) ---

    grandparent.addEventListener("click", () => logEvent("Grandparent Clicked! (Bubbling)"));
    parent.addEventListener("click", () => logEvent("Parent Clicked! (Bubbling)"));

    child.addEventListener("click", (e) => {
      logEvent("Child Clicked! (Bubbling)");

      // If checkbox is checked, stop the event from reaching the Parent and Grandparent
      if (stopPropCheckbox.checked) {
        e.stopPropagation();
        logEvent("<i>Propagation stopped! Parents will not fire.</i>");
      }
    });

    // --- CAPTURING PHASE LISTENERS (Forces event to fire on the way DOWN) ---
    // Notice the third argument: { capture: true }

    grandparent.addEventListener("click", () => logEvent("Grandparent Captured! (Going Down)"), {
capture: true });
    parent.addEventListener("click", () => logEvent("Parent Captured! (Going Down)"), { capture: true });
    child.addEventListener("click", () => logEvent("Child Captured! (Target Hit)"), { capture: true });

    /* If you click the Child without stopping propagation, the order will be:
     1. Grandparent Captured
     2. Parent Captured
     3. Child Captured
     4. Child Bubbling
     5. Parent Bubbling
     6. Grandparent Bubbling
    */
  });
```

Custom Events

Sometimes you want to trigger an event that isn't a standard browser interaction (like a mouse click). You might want an event called userLoggedIn or dataSaved. You can create and dispatch these yourself!

- **new CustomEvent('eventName', { detail: { ... } }):** Creates the event. The detail property allows you to attach your own custom data payload to the event.
- **element.dispatchEvent(event)**: Fires the event on a specific element.

```
<script>
  document.addEventListener("DOMContentLoaded", () => {

    const statusBoard = document.getElementById("status-board");
    const statusText = document.getElementById("status-text");
    const btnFetch = document.getElementById("btn-fetch");

    // 1. LISTEN for our Custom Event
    // We listen on the status board for an event called 'dataLoaded'
    statusBoard.addEventListener("dataLoaded", (e) => {
      // We access our custom data payload via e.detail
      const payload = e.detail;

      statusText.textContent = `Success! ${payload.user} loaded ${payload.itemsFound} items.`;
      statusBoard.style.background = "#2ecc71"; // Turn green
      statusBoard.style.color = "white";
    });


    // 2. DISPATCH our Custom Event
    btnFetch.addEventListener("click", () => {
      statusText.textContent = "Fetching...";

      // Simulate a 2-second network request
      setTimeout(() => {

        // Create the custom event and attach data
        const myEvent = new CustomEvent("dataLoaded", {
          detail: {
            user: "Youssef",
            itemsFound: 42
          }
        });

        // Fire the event on the status board element!
        statusBoard.dispatchEvent(myEvent);

      }, 2000);
    });
  });
```

Cookies

Cookies are small pieces of text data (up to 4KB) saved directly in the user's browser. They are traditionally used for session management (keeping a user logged in) and tracking. The two types of cookies are persistent (across different sessions and has expiring date) and non persistent (expires once session ends)

Working with cookies in vanilla JavaScript is notoriously clunky because document.cookie acts like one giant, continuous string, rather than a neat object.

- **Creating a Cookie:** You assign a string in the format "key=value" to document.cookie. You should also include an expiration date; otherwise, the cookie is deleted the moment the user closes the browser (a "session cookie").
- **Reading a Cookie:** Calling document.cookie returns all active cookies as one string: "name=Youssef; theme=dark; auth=true". You have to use string manipulation to find the exact cookie you want.
- **Deleting a Cookie:** You cannot explicitly "delete" a cookie. Instead, you overwrite the cookie with an expiration date set to the past, forcing the browser to instantly garbage-collect it.

```
document.addEventListener("DOMContentLoaded", () => {
  const input = document.getElementById("cookie-input");
  const display = document.getElementById("cookie-display");

  // --- 1. CREATE COOKIE ---
  document.getElementById("btn-create").addEventListener("click", () => {
    const username = input.value.trim();
    if (!username) return alert("Please enter a username.");

    // Calculate a date 7 days into the future
    const d = new Date();
    d.setTime(d.getTime() + (7 * 24 * 60 * 60 * 1000));
    const expires = "expires=" + d.toUTCString();

    // Build the cookie string. The "path=/" ensures the cookie is available across the whole site.
    document.cookie = `username=${username}; ${expires}; path=/`;

    alert("Cookie saved!");
    input.value = "";
  });

  // --- 2. READ COOKIE ---
  // Helper function to extract a specific cookie by its name
  function getCookie(cookieName) {
    const nameString = cookieName + "=";
    // Split the giant document.cookie string into an array of individual cookies
    const cookiesArray = document.cookie.split(';');

    for (let i = 0; i < cookiesArray.length; i++) {
      let c = cookiesArray[i].trim();
      // If we find the one that starts with our nameString, return its value
      if (c.indexOf(nameString) === 0) {
        return c.substring(nameString.length, c.length);
      }
    }
    return ""; // Return empty string if not found
  }

  document.getElementById("btn-read").addEventListener("click", () => {
    const savedUser = getCookie("username");
    display.textContent = savedUser ? savedUser : "No cookie found!";
  });

  // --- 3. DELETE COOKIE ---
  document.getElementById("btn-delete").addEventListener("click", () => {
    // To delete, we set the exact same cookie name, but force the expiration to a date in the past
(1970)
    document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
    display.textContent = "Deleted.";
    alert("Cookie successfully deleted!");});});
```

AJAX & JSON

JSON

Before we fetch data from a server, you need to understand the format that data travels in. JSON is the universal language of the web.

Servers and browsers cannot easily send raw JavaScript Objects ({ name: "Youssef" }) back and forth over the network. They can only send **text**. JSON is a way of converting complex JavaScript objects and arrays into a flat string of text, and vice versa.

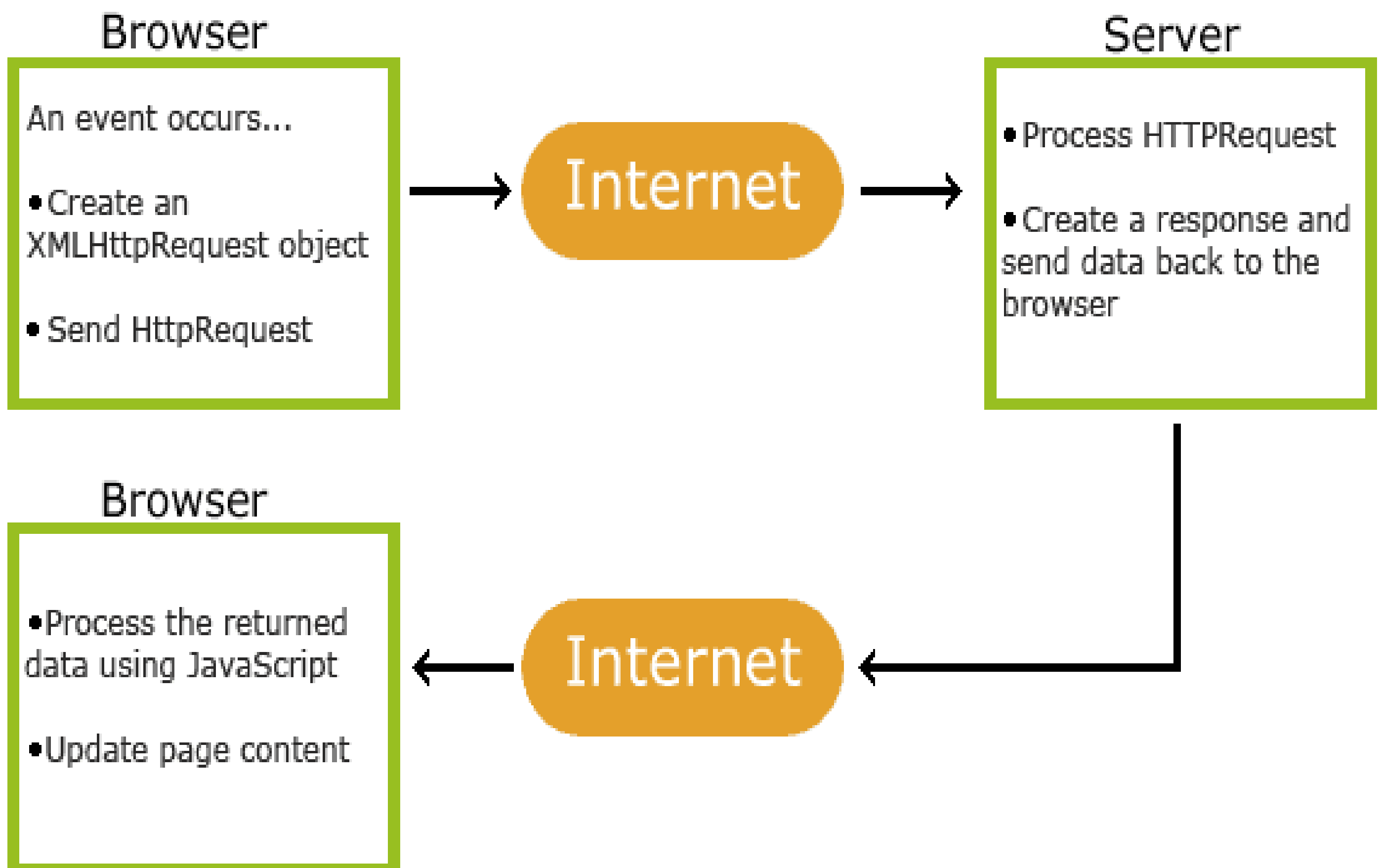
- **JSON.stringify(object)**: Converts a JavaScript object into a JSON text string (used when *sending* data to a server).
- **JSON.parse(string)**: Converts a JSON text string back into a living JavaScript object (used when *receiving* data from a server).

```
// A standard JavaScript Object
const userProfile = {
  name: "Youssef",
  age: 21,
  skills: ["Python", "JS"]
};

// 1. Pack it into text (Stringify)
const jsonString = JSON.stringify(userProfile);
console.log(jsonString);
// Output: '{"name":"Youssef","age":21,"skills":["Python","JS"]}'

// 2. Unpack it back into a usable JS object (Parse)
const receivedData = JSON.parse(jsonString);
console.log(receivedData.skills[0]);
// Output: "Python"
```

AJAX (Asynchronous JavaScript and XML) & fetch()

AJAX is the concept of requesting data from a server in the background and updating the webpage *without* requiring the user to refresh the entire page.

Historically, this was done using a clunky tool called XMLHttpRequest (and it expected data in XML format). Today, we use the modern, built-in **fetch() Web API or async-await**, which is Promise-based and expects JSON.



- **The async / await approach:** This is the modern, cleanest way to write AJAX calls. It makes asynchronous, background code read exactly like normal, top-to-bottom synchronous code.

```
document.addEventListener("DOMContentLoaded", () => {

  const btnFetch = document.getElementById("btn-fetch");
  const loadingIndicator = document.getElementById("loading");
  const userCard = document.getElementById("user-card");

  // DOM elements to inject data into
  const displayName = document.getElementById("display-name");
  const displayEmail = document.getElementById("display-email");
  const displayCity = document.getElementById("display-city");

  // We use the 'async' keyword to tell JS this function handles background tasks
  btnFetch.addEventListener("click", async () => {

    // Show loading state
    btnFetch.disabled = true;
    loadingIndicator.classList.remove("hidden");
    userCard.classList.add("hidden");

    try {
      // Generate a random user ID between 1 and 10
      const randomId = Math.floor(Math.random() * 10) + 1;

      // 1. MAKE THE AJAX REQUEST
      // 'await' tells JS to pause execution here until the server responds
      const response = await fetch(`https://jsonplaceholder.typicode.com/users/${randomId}`);

      // Check if the server responded with an error (e.g., 404 Not Found)
      if (!response.ok) {
        throw new Error(`HTTP error! Status: ${response.status}`);
      }

      // 2. PARSE THE JSON
      // We must 'await' again because parsing a large JSON file takes time
      const userData = await response.json();

      // 3. UPDATE THE DOM
      displayName.textContent = userData.name;
      displayEmail.textContent = userData.email;
      displayCity.textContent = userData.address.city;

      // Show the card
      userCard.classList.remove("hidden");

    } catch (error) {
      // This block runs if the network is down or the URL is wrong
      console.error("AJAX Request Failed:", error);
```

```
            alert("Failed to fetch data. Check the console for details.");
        } finally {
            // This block ALWAYS runs, whether it succeeded or failed
            loadingIndicator.classList.add("hidden");
            btnFetch.disabled = false;
        }
    });
});
```

- **Fetch:** Instead of pausing the code with await, you attach .then() methods to the fetch() call. The code inside .then() will only run once the previous step has successfully completed.

```
document.addEventListener("DOMContentLoaded", () => {
    fetch(`https://jsonplaceholder.typicode.com/posts/${randomPostId}`)
      .then(response => {
        // This block runs when the server says "Hello"
        if (!response.ok) {
          throw new Error(`HTTP error! Status: ${response.status}`);
        }
        // Start parsing the incoming text into a JavaScript Object
        return response.json();
      })
      .then(data => {
        // This block runs ONLY when response.json() is completely finished

        // Update the DOM with the final data
        postTitle.textContent = data.title;
        postBody.textContent = data.body;

        // Reveal the card
        postCard.classList.remove("hidden");
      })
      .catch(error => {
        // This block catches network errors or parsing errors from ANY step above
        console.error("The Promise was rejected:", error);
        alert("Failed to load post. See console.");
      })
      .finally(() => {
        // This runs at the very end, regardless of success or failure
        loadingIndicator.classList.add("hidden");
        btnFetch.disabled = false;
                    });
            });
    });
```

APIs

What is an API? (The Restaurant Analogy)

The easiest way to understand a Server API is to think of a restaurant.

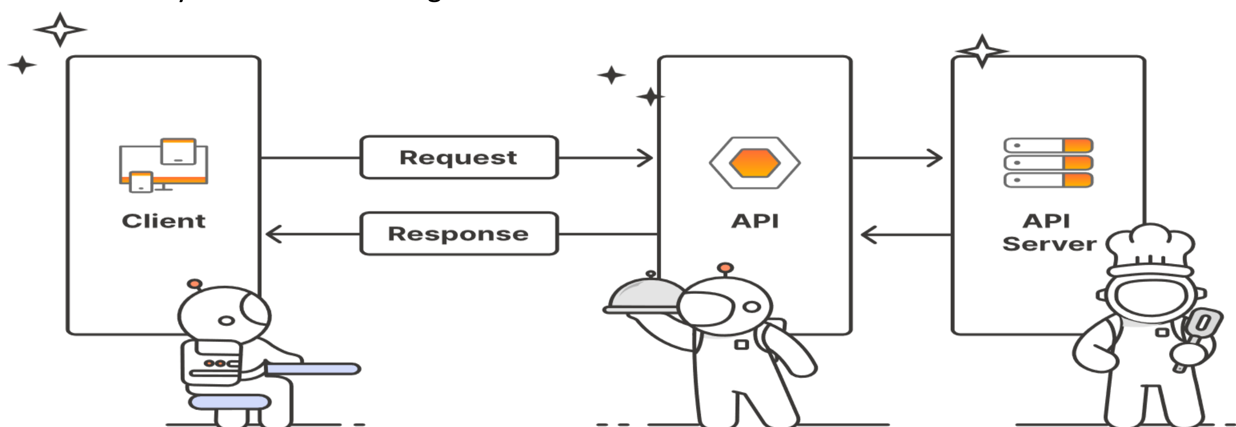
- **You (The Client / Frontend):** You are sitting at the table with a menu. You know what you want to eat, but you aren't allowed to go into the kitchen to cook it yourself.
- **The Kitchen (The Server / Database):** This is where the complex work happens, where the raw ingredients (data) are stored, and where the chefs (backend logic) prepare the food.
- **The Waiter (The API):** You tell the waiter your order. The waiter takes your order to the kitchen, waits for the food to be prepared, and then delivers the final plate right back to your table.

You don't need to know *how* the kitchen cooked the food, and the kitchen doesn't need to know who you are. The API (waiter) handles the transaction using a strict set of rules (the menu).

Why do we use APIs?

Before APIs, if you wanted to build an app that showed a map, processed credit cards, and sent text messages, you had to build a satellite mapping system, a banking network, and a telecom grid from scratch.

- **Reusability & Speed:** APIs allow developers to "plug in" features built by other massive companies. You can use the **Stripe API** to take payments, the **Google Maps API** for navigation, and the **Twilio API** to send SMS texts, all in a few lines of code.
- **Security:** A company will never give you direct access to their raw database. Instead, they give you an API. The API acts as a bouncer, only allowing you to ask for specific data and rejecting requests you don't have permission for.
- **Platform Independence:** Because APIs communicate using universal text strings (like JSON), a frontend written in JavaScript, an iPhone app written in Swift, and a smart fridge can all talk to the exact same Python backend using the exact same API.



How do APIs Work? (The Mechanics)

When you use fetch() to talk to an API, you have to follow its rules. Every API interaction consists of four main parts:

A. The Endpoint (The URL)

An endpoint is a specific web address where the API listens for requests.

- https://api.github.com/users/youssef -> Might return a user profile.
- https://api.github.com/users/youssef/repos -> Might return a list of projects.

B. The Method (The "Verb")

You have to tell the API *what* you want to do with the data at that endpoint. The four most common HTTP methods (often called CRUD operations) are:

- **GET:** Retrieve data (Read). This is the default for fetch().
- **POST:** Send new data to the server (Create).
- **PUT / PATCH:** Update existing data on the server (Update).
- **DELETE:** Remove data from the server (Delete).

C. The Headers & Authentication

APIs usually aren't free or open to the public. To use them, you must send an **API Key** in the "Headers" of your request. This acts like a VIP pass, proving who you are so the server can track your usage or bill you.

D. The Body (The Payload)

If you are sending a POST or PUT request, you have to attach the actual data you want to save. This is sent as a JSON string in the "body" of the request.

Integrated Code Example: A POST Request

```
// Example: Sending a new user profile to a Server API
async function createNewUser() {

  // 1. The data we want to send
  const newUser = {
    name: "Youssef",
    role: "Developer"
  };

  try {
    // 2. The fetch request with a Configuration Object
    const response = await fetch("https://jsonplaceholder.typicode.com/users", {

      method: "POST", // Changing the verb from the default GET

      headers: {
        "Content-Type": "application/json", // Telling the server we are sending JSON
        "Authorization": "Bearer YOUR_SECRET_API_KEY" // How you prove who you are
      },

      // 3. We must stringify our JS object into JSON text before sending!
      body: JSON.stringify(newUser)
    });

    if (!response.ok) {
      throw new Error(`Server rejected request: ${response.status}`);
    }

    // 4. The server usually replies with the saved data (and a new ID)
    const savedData = await response.json();
    console.log("Successfully created user with ID:", savedData.id);

  } catch (error) {
    console.error("Failed to hit API:", error);
  }
}

// Execute the function
createNewUser();
```

Conclusion

We have now covered the complete "logic layer" of JavaScript. You have mastered how to store data (Variables, Arrays, Objects), how to control the flow of an application (Loops, Conditions), and how to structure reusable code (Functions, Classes).

However, a programming language purely in the console is not enough to build a website. The next phase of this document will bridge the gap between this logic and the user interface. We will move into **The Document Object Model (DOM)**, which will allow your JavaScript to reach out and modify the HTML and CSS of a webpage in real-time.

Upcoming Modules:

1. **DOM Manipulation:** Selecting elements, changing text, and modifying styles dynamically.
2. **Event Handling:** Making web pages interactive (clicks, form submissions, keyboard input).
3. **Asynchronous JavaScript:** Fetching data from servers using API calls, Promises, and Async/Await.

By mastering these upcoming sections, you will transition from writing scripts that run in a console to building fully interactive, dynamic web applications.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

© **2025 [Youssef Amgad Elkhatib]**.
This document is intended **for learning and reference purposes only**.
You are free to read, use, and share the content for personal or educational use, but:

- **Do not copy or republish this document as your own.**
- Always provide proper credit when referencing.
- Commercial use or redistribution without permission is not allowed.

You may not reproduce this document in whole or in part without attribution. Suggestions and feedback are welcome, and contributors will be acknowledged, but copyright remains with the author.

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution