JavaScript

Content

1. Introduction
2. JavaScript syntax
3. Control structure and function
4. Array and object
5. Basic DOM and Events

HUST

hust.edu.vn fb.com/dhbkhn

1. Introduction

JavaScript Fundamentals

- With HTML and CSS, could websites become interactive?
- Front-end
  - HTML = Structure
  - CSS = Design
  - JavaScript = Interaction

Role of JavaScript

- Client slide (Frontend)
  - Role: event handling, validation, animations, etc.
  - Framework/Libarary: React, Vue, Angular
- Server slide (Backend)
  - Role: hande requests, databases, API, etc.
  - Framework/Environment: Node.js, Express
- Full stack: one language for both Client and Server

Aspect	Client-side JavaScript	Server-side JavaScript
Runs on	User's browser	Server (Node.js runtime)
Main purpose	Interact with webpage (UI, DOM, events)	Handle requests, databases, and APIs
Access to files / OS	❌ No (for security reasons)	✅ Yes
Examples	Form validation, animations	Reading files, connecting to DB

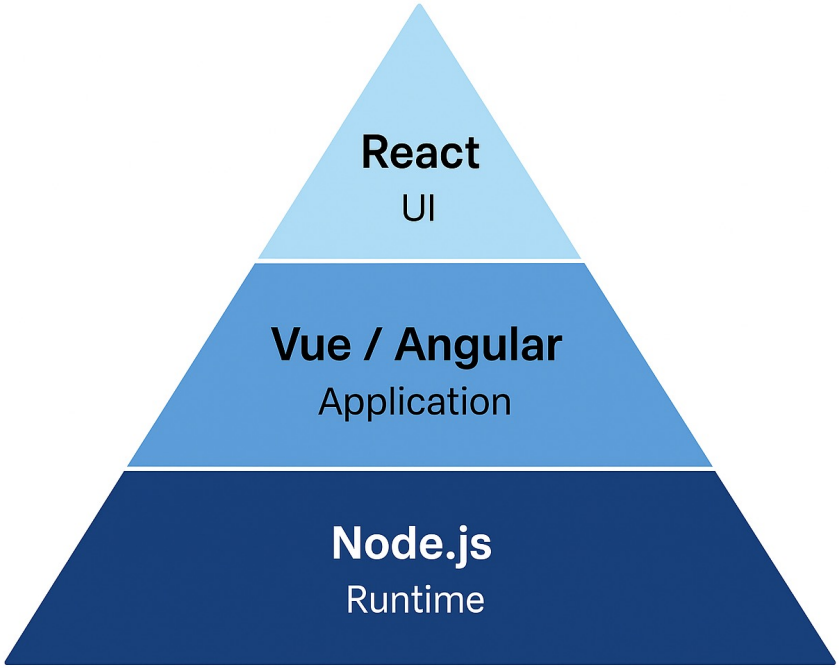
Questions

- Why has JavaScript become one of the world's most widely used programming languages?

- Run everywhere
  - Browser: Chrome (V8), Firefox (SpiderMonkey), Safari (JavaScriptCore)
  - Server: Node.js, Deno
  - Mobile: React Native, Ionic
  - Desktop: Electron
  - AI/Data: TensorFlow.js, WebGPU

Questions

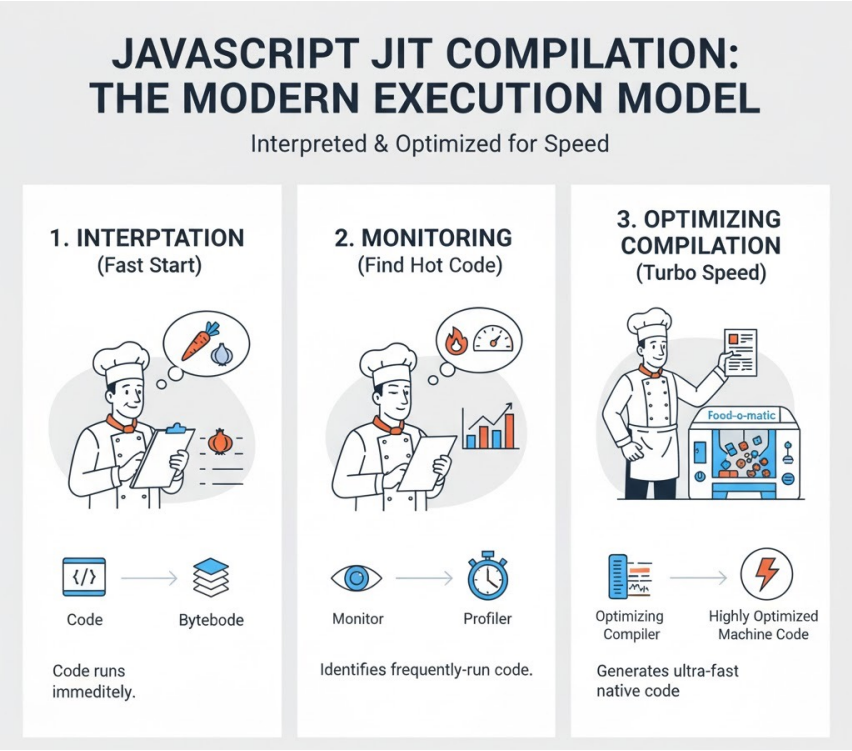
- What is the difference between framework and library?
- Library = You call it, e.g., React
- Framework = It calls you, e.g., Vue, Angular



What is JavaScript?

- Dynamic, prototype-based, and single-threaded language.
- Supports object-oriented and event-driven paradigms.
- Modern JavaScript is considered an interpreted language, but is actually executed using Just-In-Time Compilation (JIT) techniques

```
let x = 10;  // kiểu động
x = "Hello"; // hợp lệ
```



JavaScript vs Java

Feature	Java	JavaScript
Role	Large, robust applications	Lightweight, simple scripting language
Typing	Static, Strong	Dynamic, Weak
Execution	Compiled by JVM Multi-threads	Interpreted/Just-in-time complied by JS Engine Single thead
Runtime	JVM	Browser or Node.js
Object model	Class-based	Prototype-based
Platform	Cross-platform	Cross-platform

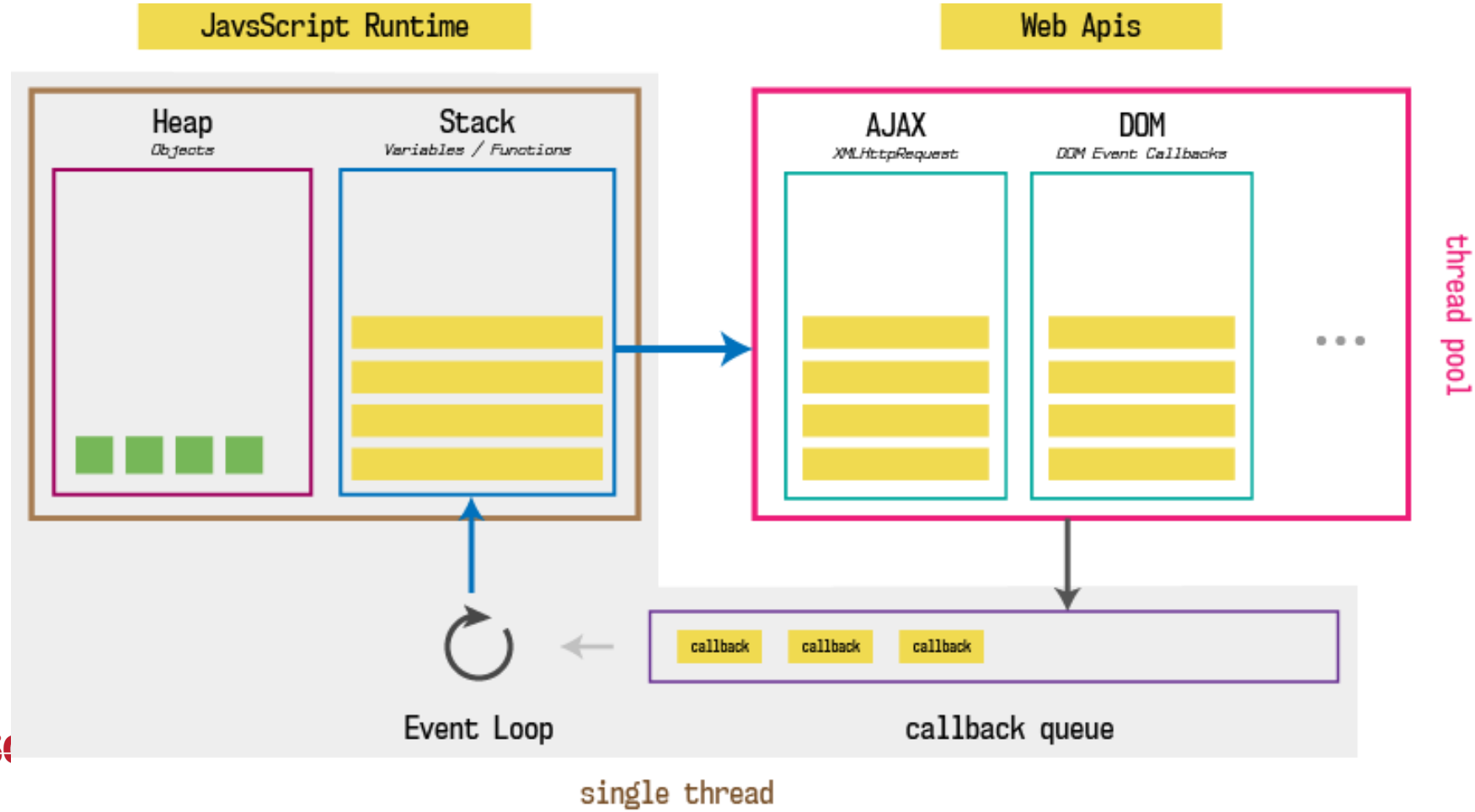
JavaScript vs Java

- Why do two languages (Java, JavaScript) with similar names have such opposite design philosophies?

Feature	Java	JavaScript
Year	1995, Sun Microsystemts	1995, Netscape
Role	Large, robust applications	Lightweight, simple scripting language
Initial Name	Oak	Mocha -> LiveScript -> JavaScript

How JavaScript executes in the browser?

- Can JavaScript run multiple tasks at once?
  - No
  - However it can handle multiple tasks concurrently



Embedding JavaScript in HTML

- Inline: <button onclick="alert('Hi')">
- Internal: <script> ... </script>
- External: <script src="main.js"></script>
  - Prefer: external JS for scalability
  - Place scripts at the **end of <body>**

Why does placing the script tag at the bottom?

```
<!DOCTYPE html>
<html>
<head>
    <title>External JS Example</title>
</head>
<body>
    <h1>Hello World</h1>

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

HUST

hust.edu.vn fb.com/dhbkhn

2. JavaScript syntax

Variable and Scope

- var: function scope/global (old, avoid using)
- let: block scope, can update value
- const: block scope, cannot update value

```
function testScope() {
  if (true) {
    var x = 1;       // function scoped
    let y = 2;       // block scoped
    const z = 3;     // block scoped
  }
  console.log(x); // ✅ works
  console.log(y); // ❌ ReferenceError
}
```

Why is it considered a bad practice to use var in JavaScript?

Variable and Scope

Feature	var	let / const
Old JS (before ES6, 2015)	✅	❌
Safe in block-scope	❌	✅
Hoisting	✅ (initialized as undefined)	⚠️ Yes (but in Temporal Dead Zone)
Re-declaration	✅ (Allowed)	❌
Global Object Binding	✅ (adds to window)	❌

Variable and Scope

- Block-scope

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

```
if (true) {
  let y = 10;
}
console.log(y); // ❌ ReferenceError
```

- Hoisting: var decalaration is moved to the top of the scope; let and const: are putted in a Temporal Dead Zone (TDZ) -> safer

```
console.log(a);
var a = 5;
```

→

```
var a;
console.log(a); // undefined
a = 5;
```

```
console.log(b); // ❌ ReferenceError
let b = 5;
```

- Re-declaration

```
var count = 1;
var count = 2;
```

- Global Object Binding: a var variable is binded to the window object

```
var user = "Hieu";
console.log(window.user);
```

Primitive Data Types

- Primitive Data Types
  - string
  - number
  - boolean
  - null
  - undefined
  - symbol
  - bigint (ES2020)
- Primitive values are *copied by value*, not by reference.

```
let a = "Hello";
let b = a;
b = "Hi";
console.log(a); // "Hello" → unchanged
```

Reference Types (Objects, Arrays)

- Reference types are stored as *pointers* to memory objects.
- When we assign one variable to another, they are *copied by reference.* Changes via one variable affect the other.

```
let person1 = { name: "Alice" };
let person2 = person1; // both refer to same object

person2.name = "Bob";
console.log(person1.name); // "Bob" → shared reference
```

Equality Operator (== vs ===)

- Loose equality
  - == compares values only
  - allow type coercion: conversion of values from one data type to another (e.g., string to number)
- Strict equality
  - === compares both value and type, no coercion.
  - Always prefer === for predictable logic

```
console.log(2 == "2");   // true  → type coerced
console.log(2 === "2");  // false → different types
console.log(null == undefined); // true
console.log(null === undefined); // false
```

Automatic Semicolon Insertion (ASI)

- JavaScript engine automatically inserts semicolons (;) to fix syntax errors during parsing.

```
let x = 5 // ASI inserts ';' here
```

- ASI doesn't always work as expected — especially with return statements, array literals, or function calls.

```
function getValue() {
  return
  {
    value: 10
  };
}
console.log(getValue());
```

```
return;   // automatic semicolon
{ value: 10 };
```

```
function getValue() {
  return {
    value: 10
  };
}
console.log(getValue()); // ✅
```

```
function getValue() {
  return (
    {
      value: 10
    }
  );
}
console.log(getValue()); // ✅
```

Template Literals and Expression Interpolation

- Template literals use backticks (`) instead of quotes to represent multi-line strings

```
// Cách cũ (rất khó đọc)
const oldString = "Xin chào!\n" +
                  "Đây là đoạn thông báo\n" +
                  "trên nhiều dòng.";

// Với Template Literals (Rất dễ đọc)
const newString = `Xin chào!
Đây là đoạn thông báo
trên nhiều dòng.`;
```

- Expression interpolation use ${...} syntax to simplify string concatenation and dynamic content generation.

```
let name = "Alice";
let age = 22;

console.log(`Hello, my name is ${name} and I am ${age} years old.`);
```

Console and Debugging

- The console is part of every JavaScript runtime
  - console.log(): general information
  - console.table(): object information in table form
  - console.dir(): object, DOM element in tree form
- Chrome DevTools (F12 → Console tab)

```
let user = { name: "Bob", age: 25, skills: ["JS", "CSS", "HTML"] };
let users = [
  { name: "Alice", age: 22 },
  { name: "Bob", age: 25 }
];

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

console.log("→ console.dir(user)");
console.dir(user);
```

```
→ console.table(users)
```

(index)	name	age
0	'Alice'	22
1	'Bob'	25

```
▸ Array(2)
→ console.dir(user)
▾ Object i
    age: 25
    name: "Bob"
  ▾ skills: Array(3)
      0: "JS"
      1: "CSS"
      2: "HTML"
      length: 3
    ▸ [[Prototype]]: Array(0)
  ▸ [[Prototype]]: Object
```

Dynamic Typing in JavaScript

- Variables can hold any type and change type at runtime.
- No need to declare a variable's type — JS infers it automatically.
  - Pros: Flexible, fast prototyping
  - Cons: Harder to debug type errors; runtime crashes

```
let age = 25;       // number
age = "twenty";     // now a string
console.log(age);   // "twenty"
```

- TypeScript adds static typing on top of JS to catch errors (need to install TypeScript Compiler - TSC)

```
let age: number = 25;
age = "twenty"; // ❌ Error:
```

HUST

hust.edu.vn fb.com/dhbkhn

3. Control Statement and Function

Control Structure

- Condition: if, else if, else, switch
- Loop: for, while, do…while
- Jump: break, continue, return

```
let score = 85;
if (score >= 90) {
  console.log("Excellent");
} else if (score >= 70) {
  console.log("Good");
} else {
  console.log("Try again");
}
```

Control Structure

```
// Duyệt qua các số từ 1 đến 10
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    console.log("Bỏ qua số 5");
    continue; // Bỏ qua phần còn lại của vòng lặp khi i = 5
  }

  if (i === 8) {
    console.log("Dừng vòng lặp tại số 8");
    break; // Thoát khỏi vòng lặp khi i = 8
  }

  console.log("Giá trị i =", i);
}

console.log("Kết thúc vòng lặp!");
```

```
Giá trị i = 1
Giá trị i = 2
Giá trị i = 3
Giá trị i = 4
Bỏ qua số 5
Giá trị i = 6
Giá trị i = 7
Dừng vòng lặp tại số 8
Kết thúc vòng lặp!
```

Control Structure

```
let i = 0;

while (i < 10) {
  i++;

  if (i === 3) {
    console.log("Bỏ qua số 3 (continue)");
    continue; // bỏ qua bước này, sang lần lặp kế tiếp
  }

  if (i === 8) {
    console.log("Dừng vòng lặp tại số 8 (break)");
    break; // thoát khỏi vòng lặp hoàn toàn
  }

  console.log("Giá trị i =", i);
}

console.log("Kết thúc vòng while!");
```

```
Giá trị i = 1
Giá trị i = 2
Bỏ qua số 3 (continue)
Giá trị i = 4
Giá trị i = 5
Giá trị i = 6
Giá trị i = 7
Dừng vòng lặp tại số 8 (break)
Kết thúc vòng while!
```

Operators

Operators

+, -, *, /, %, ++, --, …

==, !=, <, >, <=, >=

&&, ||, !,===,!==

```
<script type="text/javascript">
   const distanceToSun = 93.3e6 * 5280 * 12;
   let thickness = .002;
   let foldCount = 0;
   while (thickness < distanceToSun)       {
      thickness *= 2;
      foldCount++;
   }
   document.write("Number of folds = "
+foldCount);
  </script>
```

Ternary Operator

- condition ? expressionIfTrue : expressionIfFalse;

```
let age = 20;
let access = (age >= 18) ? "Allowed" : "Denied";
console.log(access); // "Allowed"
```

- When does using the ternary operator improve clarity and when does it hurt readability?
  - Good: condition is simple & expressions are short
  - Not good: condition or expressions are complex

✅
```
let ageGroup = age >= 18 ? "Adult" : "Minor";
```

❌
```
let result = score > 90 ? "A" : score > 75 ? "B" : score > 60 ? "C" : "D";
```

for...of and for...in

Loop Type	Iterates Over	Works Best With
for...of	**Values**	Arrays, strings, collections
for...in	**Keys/property names**	Objects (key–value pairs)

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

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

```
let person = { name: "Alice", age: 25 };

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
// Output:
// name: Alice
// age: 25
```

Function: Declaration and Expression

- A function is a reusable block of code designed to perform a specific task.
- Two main ways to define a function

Type	Syntax	Hoisting	Use Case
Declaration	function name() { ... }	✅ Fully hoisted	named reusable functions
Expression	const name = function() {... };	❌ Not hoisted	passing as arguments

```
sayHi(); // ✅ Works

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

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

greet(); // ✅ Works
```

```
greet(); // ❌ ReferenceError
const greet = function() { console.log("Hi!"); };
```

```
const greet = function sayHello() {
  console.log("Hi!");
};
greet();      // ✅ Works
sayHello();   // ❌ Error (not accessible outside)
```

Functions as Data

- In JavaScript, functions are objects (**First-class functions**)
  - Assign a function to a variable
  - Return a function from another function
  - Pass a function as an argument to another function (callback)

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

greet(); // ✅ works – greet
```

```
function multiplier(x) {
  return function(y) {
    return x * y;
  };
}

const double = multiplier(2);
console.log(double(5)); // ✅ 10
```

```
function greet(name) {
  console.log("Hello, " + name);
}

function processUserInput(callback) {
  const name = "Alice";
  callback(name); // invoke the function passed as an argument
}

processUserInput(greet);
```

Arrow Function

- A concise way to write anonymous functions.
- Syntax uses the => (arrow) symbol.
- They are often used for: short callbacks, array methods, inline event handlers

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

// Arrow function
const addArrow = (a, b) => a + b;

console.log(addArrow(2, 3)); // ✅ 5
```

```
let numbers = [1, 2, 3];
let squares = numbers.map(n => n * n);

console.log(squares); // [1, 4, 9]
```

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

btn.addEventListener("click", () => {
  alert("Button clicked!");
});
```

Closure

- A closure: is a function that “remembers” variables from the scope where it was created — even after that scope has finished executing.

```
function outer() {
  let count = 0;  // variable in outer scope

  function inner() {
    count++;                  // inner function "remembers" count
    console.log(count);
  }

  return inner;  // return the inner function
}

const counter = outer(); // outer() runs once
counter(); // 1
counter(); // 2
counter(); // 3
```

```
outer()
  └── count = 0
        ↓
    inner() → remembers → count
```

Closure

- Closure can maintain a **private, persistent state** that is protected from external code.
- counterA and counterB are separate closures

```
function createCounter() {
    // 'count' is the private variable that the closure remembers
    let count = 0;

    return function() {
        count += 1; // Accessing the remembered 'count'
        return count;
    };
}

const counterA = createCounter(); // Creates environment A (count = 0)
const counterB = createCounter(); // Creates environment B (count = 0)

console.log(counterA()); // Output: 1
console.log(counterB()); // Output: 1 (Independent from A)
console.log(counterA()); // Output: 2 (Continues A's state)
```

Programming Mindset: Flexibility over Safety

- JavaScript gives you freedom — but freedom without discipline is chaos
  - The compiler won't stop you from doing risky things.
  - Mistakes often show up only at runtime, not compile time.
- Approaches
  - Typing: validate inputs and use TypeScript
  - ASI: syntax conventions
  - Implicit conversion: understand coercion rules

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

console.log(add(2, 3));     // 5 ✅
console.log(add("2", 3));   // "23" ⚠️ (type coercion)
console.log(add(true, {})); // "true[object Object]" 😅
```

HUST

hust.edu.vn fb.com/dhbkhn

4. Array and Object

Array

- Dynamic size:
  - JavaScript arrays can grow (add elements) or shrink (remove elements) at runtime.
  - Users don’t need to predefine their length as in C
- Untyped / Heterogeneous:
  - A JavaScript array can hold different data types in the same structure.
  - Example: numbers, strings, objects, even functions — all in one array.

```
let arr = [42, "Hello", true, { name: "Alice" }, [1, 2, 3]];
console.log(arr[1]); // "Hello"
arr.push(99);        // Add new element dynamically
console.log(arr.length); // 6
```

Basic Array Operations

Method	Action	Position
`push()`	Add element	End
`pop()`	Remove element	End
`shift()`	Remove element	Beginning
`unshift()`	Add element	Beginning

```
let fruits = ["apple", "banana"];
fruits.push("orange");
console.log(fruits); // ["apple", "banana", "orange"]
```

```
let last = fruits.pop();
console.log(last);   // "orange"
console.log(fruits); // ["apple", "banana"]
```

Array Iteration

- forEach(): runs a function on each element
- map(): creates a new array with tranformed values
- filter(): creates a new array with a condition
- reduce(): reduces an array to one value (e.g, sum)

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

```
let doubled = numbers.map(n => n * 2);
console.log(doubled); // [2, 4, 6]
```

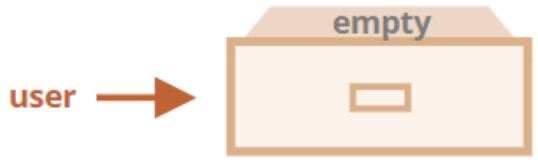
```
let even = numbers.filter(n => n % 2 === 0);
console.log(even); // [2]
```

```
let sum = numbers.reduce((acc, n) => acc + n, 0);
console.log(sum); // 6
```

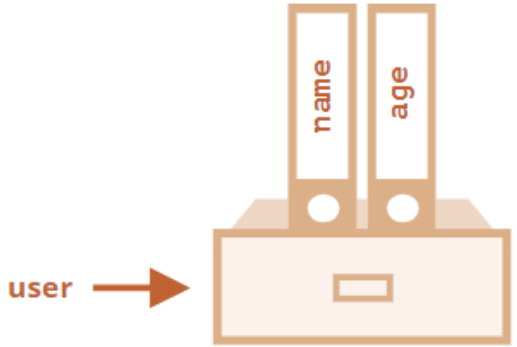
Objects

- An object contains list of properties.
- A property is a “key:value” pair, where key is a string and value can be anything.

```
1 let user = new Object(); // "object constructor" syntax
2 let user = {};  // "object literal" syntax
```

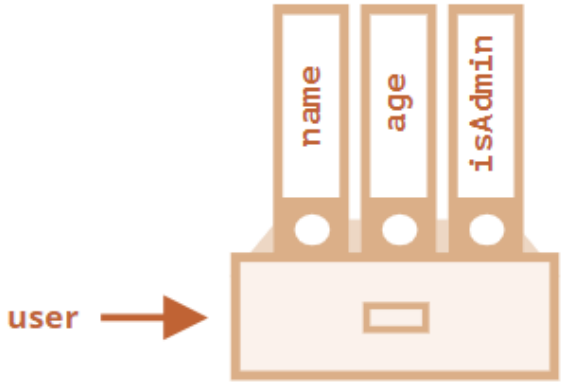



```
1 let user = {     // an object
2   name: "John",  // by key "name" store value "John"
3   age: 30        // by key "age" store value 30
4 };
5 console.log(user.name) // John
6 console.log(user.age)  // 30
```

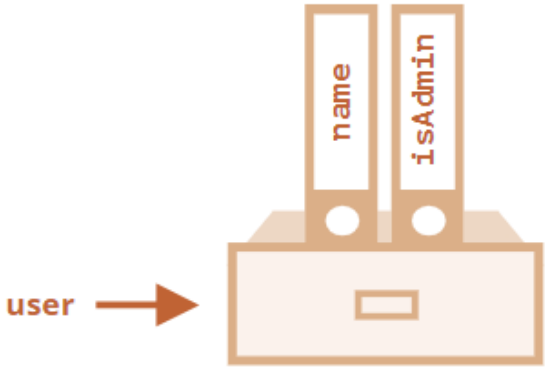


Objects

```
1 user.isAdmin = true;
```

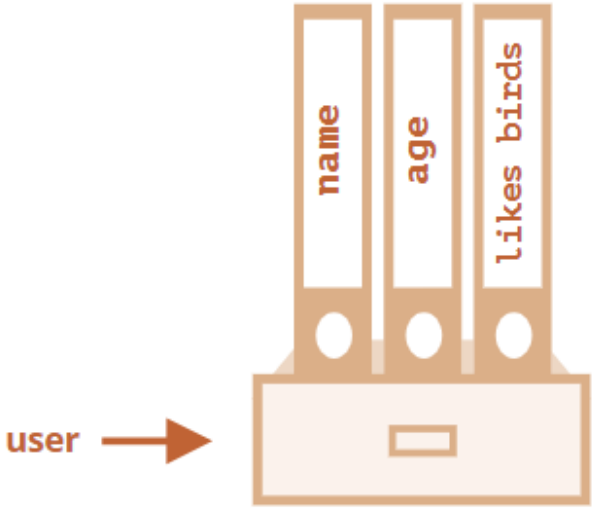



```
1 delete user.age
```



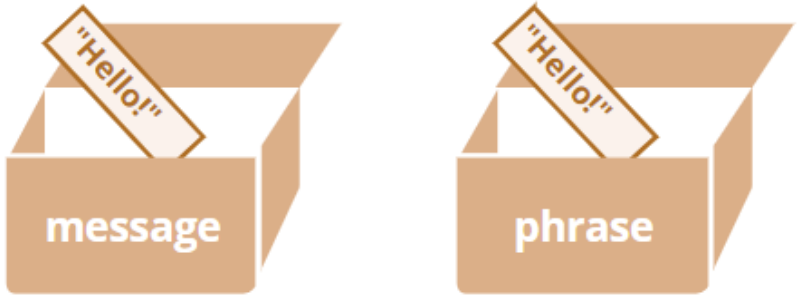
Objects

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // multiword property name must be quoted
};
console.log(user.like birds) // syntax error
console.log(user["like birds"]) //true
```

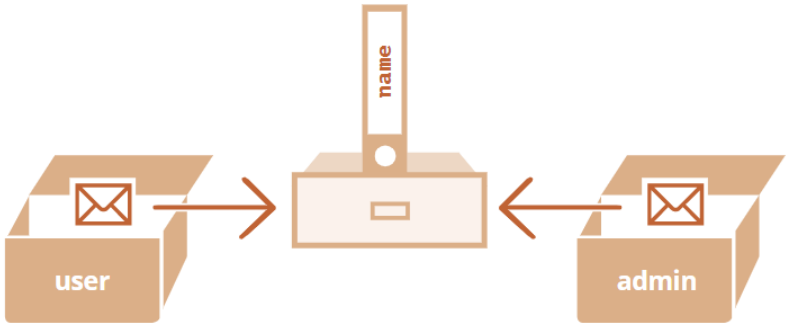


Copy Object

```
1 let message = "Hello!";
2 let phrase = message;
```

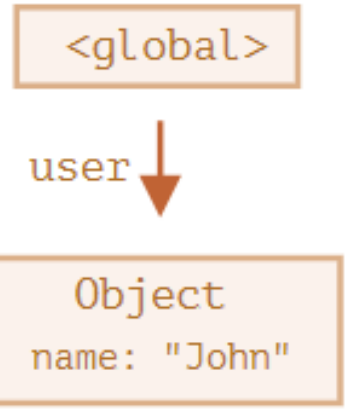



```
1 let user = { name: "John" };
2
3 let admin = user; // copy the reference
4
5 admin.name = "Peter"
6 console.log(user.name) // Peter
```

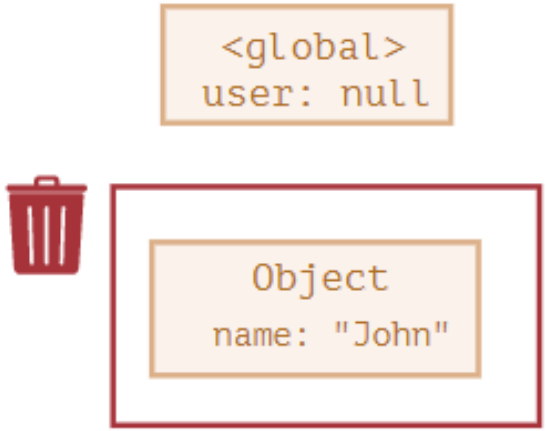


Garbage collection

```
1 // user has a reference to the object
2 let user = {
3   name: "John"
4 };
```

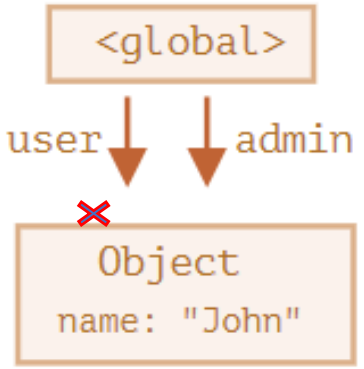



```
1 // object has no reference, garbage coll
  will junk the data and free the memory
2 user = null;
```

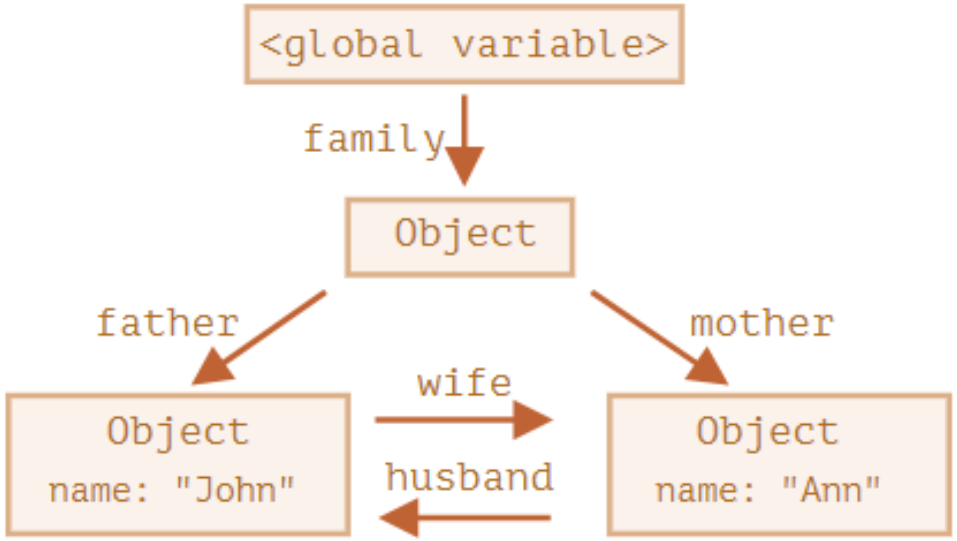


Garbage collection

```
let user = {
  name: "John"
};

let admin = user;
user = null; // object is still reachable via admin variable, so it must stay in memory
```

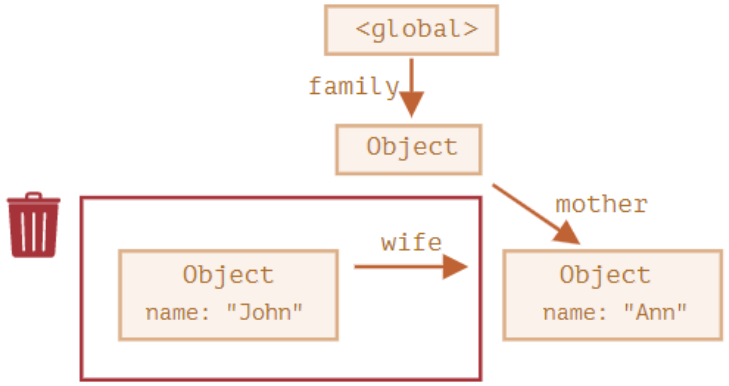



```
function marry(man, woman) {
  woman.husband = man
  man.wife = woman

  return {
    father: man,
    mother: woman,
  }
}

let family = marry(
  {name: 'John'},
  {name: 'Ann'}
)
```

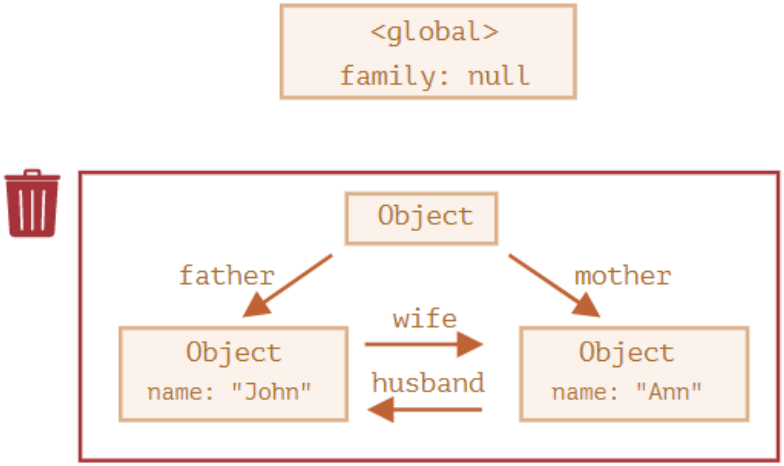


Garbage collection

```
1 delete family.father;
2 delete family.mother.husband;
3 // if we remove only one reference, all objects
  would still be reachable. But if we delete both,
  John has no incoming reference and will be junk.
```




```
1 family = null; // family object has been unlinked
  from root, so the whole island becomes unreachable
  and will be removed
```



String Object

- JavaScript supports both **primitive strings** and **String objects**.

```
// Primitive string
let s1 = "Hello";

// String object
let s2 = new String("Hello");
```

```
let a = "Hello";
let b = new String("Hello");

console.log(a == b);  // true
console.log(a === b); // false
```

- JavaScript allow **primitive strings** to call methods. JavaScript temporarily wraps a primitive string in a String object (Auto-Boxing or Temporary object)

```
let name = "Alice";
console.log(name.toUpperCase()); // "ALICE"
```

`name` → temporarily converted to `new String("Alice")` → `.toUpperCase()` → result → object discarded.

Class

- A template for creating objects

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age() {
    const date = new Date();
    return date.getFullYear() - this.year;
  }
}

const myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML =
"My car is " + myCar.age() + " years old.";
```

JSON

- JSON (JavaScript Object Notation) is a lightweight format for data exchange.
- Send data to server: convert a JavaScript object → string format

```
let user = { name: "Alice", age: 25, active: true };
let jsonString = JSON.stringify(user);

console.log(jsonString);
// '{"name":"Alice","age":25,"active":true}'
```

- Receive data from server: convert that string → object.

```
let parsed = JSON.parse(jsonString);
console.log(parsed.name); // Alice
```

HUST

hust.edu.vn fb.com/dhbkhn

5. Basic DOM and Event

DOM

- **DOM (Document Object Model)** is a tree-like representation of an HTML document.
- Every HTML element is a **node** in the DOM tree.
- It allows JavaScript to access, traverse, and manipulate the structure, content, and style of a webpage.

=> DOM is the bride between HTML and JavaScript

DOM Tree and Node Types

- Each HTML tag, piece of text, or attribute is a **node** in the tree.

```
<html>
  <body>
    <h1 id="title">Hello</h1>
    <p>Welcome to JS!</p>
  </body>
</html>
```

```
Document
└── html (Element Node)
    └── body (Element Node)
        ├── h1 (Element Node)
        │   ├── id="title" (Attribute Node)
        │   └── "Hello" (Text Node)
        └── p (Element Node)
            └── "Welcome to JS!" (Text Node)
```

Node Type	Description	Example
Document Node	Root of the DOM tree	`document`
Element Node	HTML tags	`<h1>, <p>, <div>`
Text Node	Text content	`"Hello"`
Attribute Node	Attributes of elements	`id="title"`
Comment Node	HTML comments	`<!-- note -->`

Searching: document.getElementById(id)

- If an element has the id attribute, we can get the element using document.getElementById(id)

```
1  <div id="elem">
2    <div id="elem-content">Element</div>
3  </div>
4
5  <script>
6    // get the element
7    let elem = document.getElementById('elem');
8
9    // make its background red
10   elem.style.background = 'red';
11 </script>
```

Searching: getElementsBy*

- elem.getElementsByTagName(tag)

  e.g., table.getElementsByTagName("td");

- elem.getElementsByClassName(className)

  e.g., document.getElementsByClassName("example");

- document.getElementsByName(name)

  e.g., <input name="animal" type="checkbox" value="Cats">

  document.getElementsByName("animal");

Searching: querySelectorAll

- The most versatile method: elem.querySelectorAll**(css)**
- Any CSS selector can be used

Selector	Example
.class	.intro
.class1.class2	.name1.name2
.class1 .class2	.name1 .name2
#id	#firstname
*	*
element	p
element.class	p.intro
element,element	div, p
element element	div p
element>element	div > p
element+element	div + p
element1~element2	p ~ ul
[*attribute*]	[target]
[*attribute=value*]	[target=_blank]
[*attribute~=value*]	[title~=flower]

Searching: querySelectorAll

```
1  <ul>
2    <li>The</li>
3    <li>test</li>
4  </ul>
5  <ul>
6    <li>has</li>
7    <li>passed</li>
8  </ul>
9  <script>
10   let elements = document.querySelectorAll('ul > li:last-child');
11
12   for (let elem of elements) {
13     alert(elem.innerHTML); // "test", "passed"
14   }
15 </script>
```

Searching: querySelector

- elem.querySelector(css) returns the first element for the given CSS selector

- The result is the same as elem.querySelectorAll(css)[0]
  - The latter is looking for all elements and picking one
  - elem.querySelector just looks for one. So it's faster and also shorter to write.

Searching: Summary

Method	Searches by...	Can call on an element?
`querySelector`	CSS-selector	✔
`querySelectorAll`	CSS-selector	✔
`getElementById`	`id`	-
`getElementsByName`	`name`	-
`getElementsByTagName`	tag or `'*'`	✔
`getElementsByClassName`	class	✔

The “nodeType” property

- The nodeType property provides one more, “old-fashioned” way to get the “type” of a DOM node.
- It has a numeric value:
  - elem.nodeType == 1 for element nodes,
  - elem.nodeType == 3 for text nodes,
  - elem.nodeType == 8 for comment nodes,
  - elem.nodeType == 9 for the document object

“nodeType” property example

```
<body>
  <script>
  let elem = document.body;

  // let's examine: what type of node is in elem?
  alert(elem.nodeType); // 1 => element

  // and its first child is...
  alert(elem.firstChild.nodeType); // 3 => text

  // for the document object, the type is 9
  alert( document.nodeType ); // 9
  </script>
</body>
```

innerHTML: the contents

- The innerHTML property allows to get the HTML inside the element as a string.
- We can also modify it. So it's one of the most powerful ways to change the page.

```
<body>
  <p>A paragraph</p>
  <div>A div</div>

  <script>
    alert( document.body.innerHTML ); // read the current contents
    document.body.innerHTML = 'The new BODY!'; // replace it
  </script>

</body>
```

outerHTML: full HTML of the element

- The outerHTML property contains the full HTML of the element.
- That's like innerHTML plus the element itself.

```
<div id="elem">Hello <b>World</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>
</script>
```

nodeValue/data: text node content

- The innerHTML property is only valid for element nodes.
- Other node types, such as text nodes, have their counterpart: nodeValue and data properties.
- These two are almost the same for practical use.

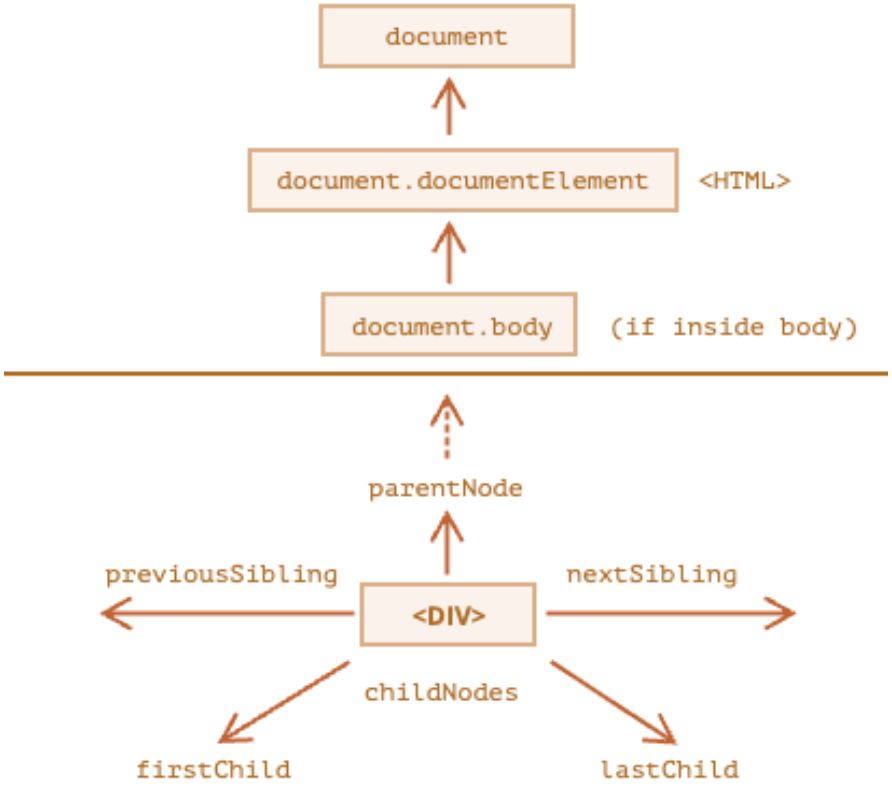
```
1  <body>
2    Hello
3    <!-- Comment -->
4    <script>
5      let text = document.body.firstChild;
6      alert(text.data); // Hello
7
8      let comment = text.nextSibling;
9      alert(comment.data); // Comment
10   </script>
11 </body>
```

Properties used to get value

Property	Applies to	Returns	Includes HTML tags?
innerHTML	Element nodes	The HTML markup inside the element	✅ Yes
outerHTML	Element nodes	The HTML of the element itself + its contents	✅ Yes
textContent	Element / Text nodes	Plain text of all descendants	❌ No
nodeValue	Text or Comment nodes	The raw text of that node	❌ No
data	Text or Comment nodes	Same as nodeValue	❌ No

Walking the DOM

- **Child nodes** (or children): elements that are direct children
- **Descendants** – all elements that are nested in the given one, including children, their children and so on.



Children: childNodes

- The childNodes collection lists all child nodes, including text nodes.

```
<!DOCTYPE html>
<html>
<body>
  <div id="box">
    <h3>Hello</h3>
    <p>Paragraph</p>
  </div>

  <script>
    let box = document.getElementById("box");
    box.childNodes.forEach(n => console.log(n.nodeName));
  </script>
</body>
</html>
```

```
#text
H3
#text
P
#text
```

Children: firstChild, lastChild

- Properties firstChild and lastChild give fast access to the first and last children
- If there exist child nodes, then the following is always true:

```
1  elem.childNodes[0] === elem.firstChild
2  elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

Modifying the document

- DOM modification is the key to creating “live” pages.
- To create DOM nodes, there are two methods:

`document.createElement(tag)`

Creates a new *element node* with the given tag:

```
1  let div = document.createElement('div');
```

`document.createTextNode(text)`

Creates a new *text node* with the given text:

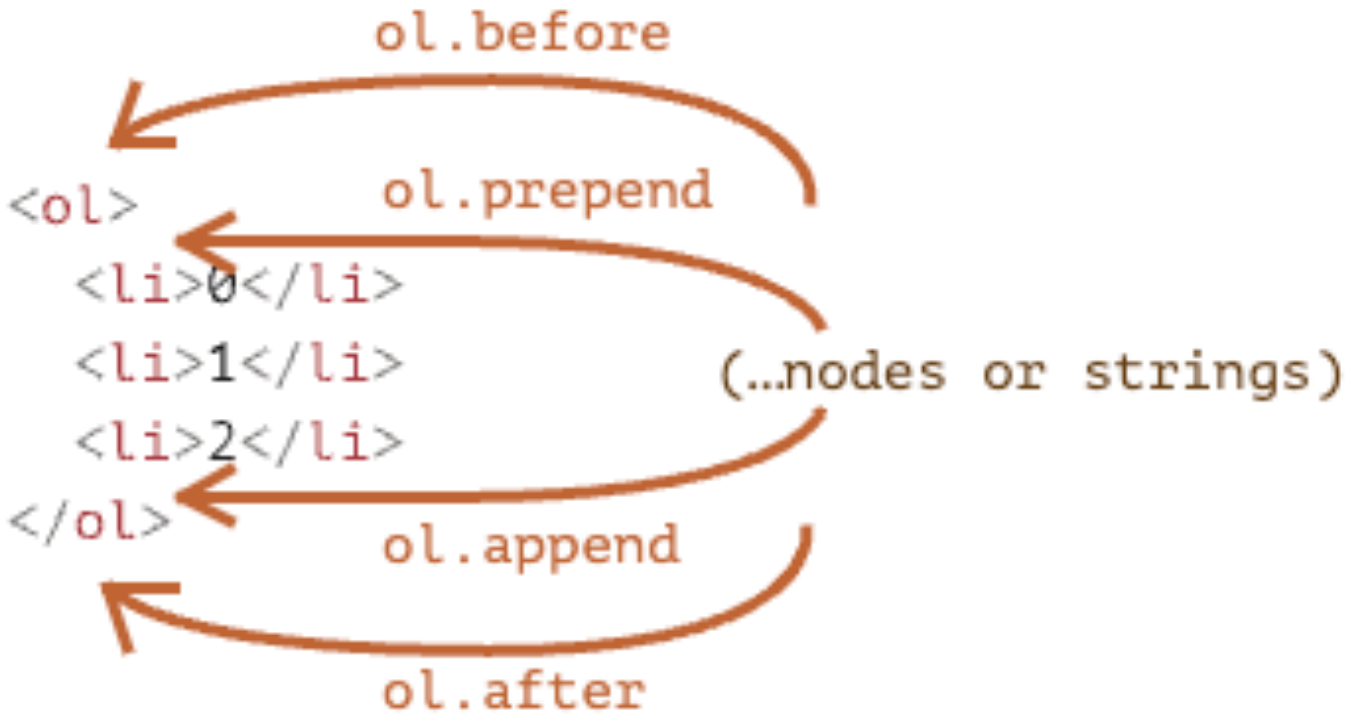
```
1  let textNode = document.createTextNode('Here I am');
```

Insertion methods

- **node.append**(...nodes or strings) – append nodes or strings *at the end* of node,
- **node.prepend**(...nodes or strings) – insert nodes or strings *at the beginning* of node,
- **node.before**(...nodes or strings) – insert nodes or strings *before* node,
- **node.after**(...nodes or strings) – insert nodes or strings *after* node,
- **node.replaceWith**(...nodes or strings) – replace node with the given nodes or strings.

Insertion methods

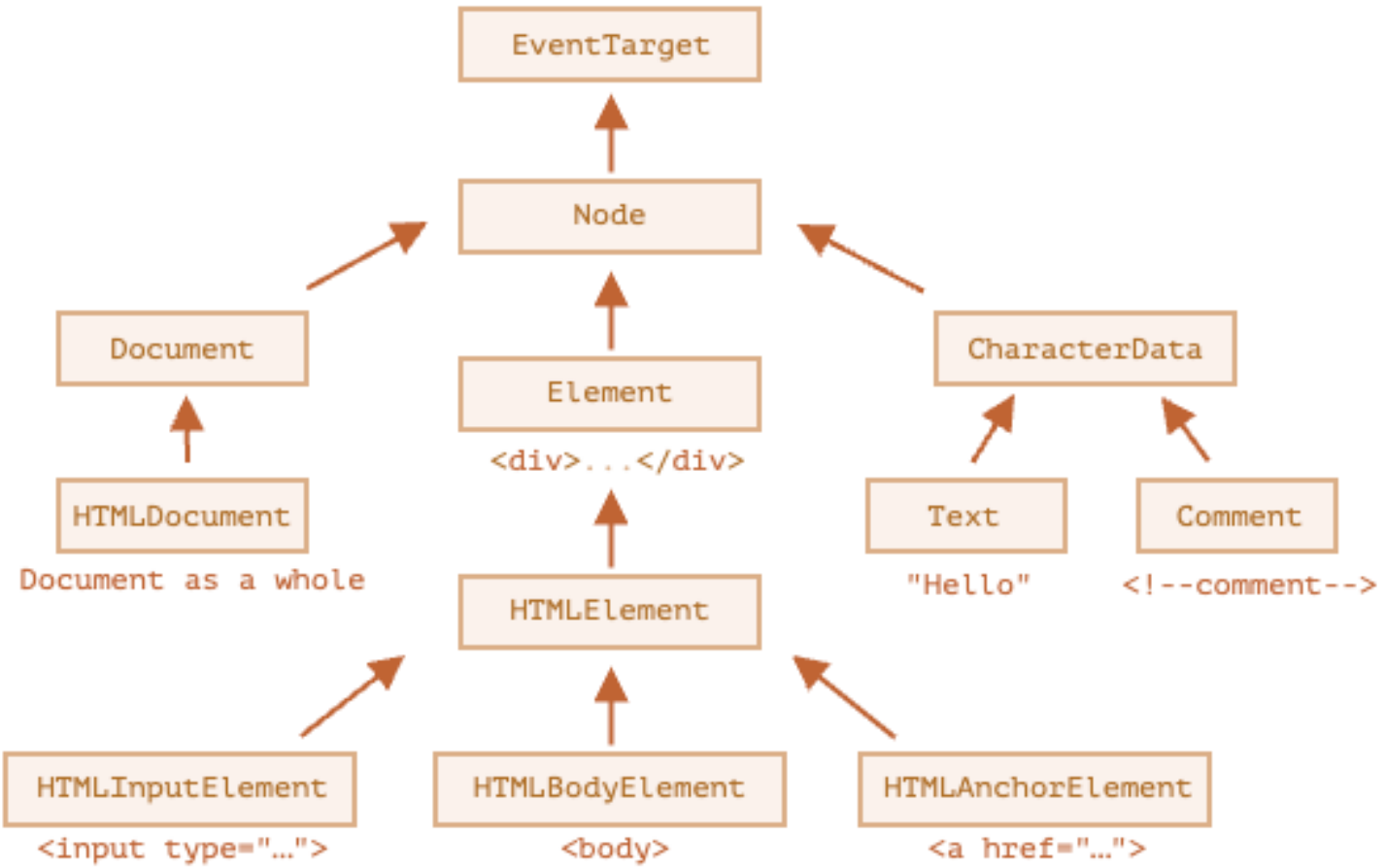
- Here's a visual picture of what the methods do:



Event – The Reaction Mechanism of the Web

- An **event** is **an action or occurrence** that happens in the browser — to which JavaScript can **react**.
  - User clicks a button
  - User types in an input box
  - Page loads or scrolls
- Common Event types
  - Mouse: click, dbclick, mouseover, mouseout
  - Keyboard: keydown, keyup, keypress
  - Form: submit, change, focus
  - Window/Document: load, resize, scroll

Node properties: hierarchy



Every node in the DOM can listen for events (EventListener)

EventTarget

Method	Purpose / Description
addEventListener()	Register a function that will be called whenever the specified event occurs
removeEventListener()	Remove a previously added event listener so it no longer responds to the event.
dispatchEvent()	Manually triggers an event on the target

```
<button id="btn">Click me</button>

<script>
  let btn = document.getElementById("btn");

  // 1️⃣ Gắn listener
  btn.addEventListener("click", () => {
    console.log("Button was clicked!");
  });

  // 2️⃣ Tự kích hoạt sự kiện click
  let event = new Event("click");
  btn.dispatchEvent(event);
</script>
```

```
Button was clicked!
```

Attaching Event Handler

- Inline (HTML level)

```
<button onclick="alert('Clicked!')">Click me</button>
```

- DOM property

```
let btn = document.getElementById("btn");
btn.onclick = function() {
  alert("Button clicked!");
};
```

- addEventListener (prefered)

```
btn.addEventListener("click", () => {
  alert("Hello from event listener!");
});
```

Event Object

- Browser automatically creates an event object when an event happens
- Named e or event that contains all details about that specific event.

```
btn.addEventListener("click", (e) => {
  console.log(e.type); // "click"
  console.log(e.target.id); // element ID
});
```

Properties

Property	Description	Example
`e.type`	Type of event	`"click"`, `"submit"`
`e.target`	Element that triggered the event	`<button>`
`e.currentTarget`	Element that handles the event	Parent element
`e.clientX`, `e.clientY`	Mouse coordinates	250, 120
`e.key`	Key pressed (for keyboard events)	`"Enter"`

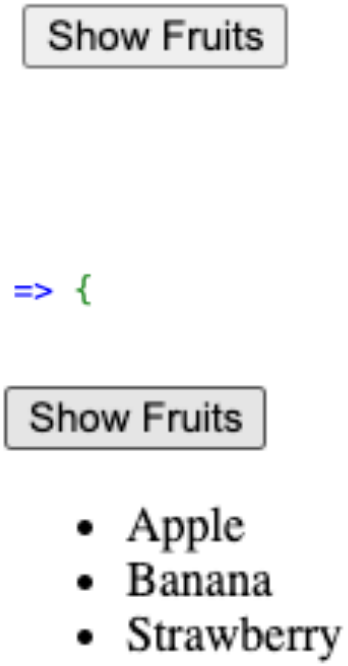
DOM and JavaScript

Aspect	Document Object Model	JavaScript
Nature	A tree-like **object model** created by the browser to represent HTML document.	A **programming language** used to control logic, handle data, and manipulate DOM.
Standardization	W3C / WHATWG.	ECMA (ECMAScript).
Components	Objects: document, window, Node, Event, etc. Methods: getElementById(), createElement(), etc.	Data types (number, string...), Control structures functions, classes, etc.
Created By	Automatically created by the **browser** when loading an HTML page.	Written by developers and executed by the JS engine
Aspect	Access & modify web page content	Control logic & interact with DOM

DOM and JavaScript

- DOM + JavaScript = Dynamic UI
- JavaScript can change HTML, CSS, and behavior in realtime

```
<body>
  <button id="showList">Show Fruits</button>
  <ul id="list"></ul>

  <script>
    const fruits = ["Apple", "Banana", "Strawberry"];
    document.getElementById("showList").addEventListener("click", () => {
      const list = document.getElementById("list");
      list.innerHTML = ""; // clear old list
      fruits.forEach((fruit) => {
        let li = document.createElement("li");
        li.textContent = fruit;
        list.appendChild(li);
      });
    });
  </script>
</body>
```



Exercises

- Ex1. Define factorial fuction
- Ex2. Given an array of objects (e.g., a list of students with name and score), use the array methods (map, filter, reduce) to calculate the average score.
- Ex3. Create a button and a text paragraph. Add an **Event Listener** to the button so that every time it is clicked, a text is shown/hiden by changing its CSS class.
- Ex4. Create an input field and a button. When the button is clicked, take the text from the input, **create a new <li> element** and append it to an unordered list (<ul>).