

# What is JavaScript

**High-level, interpreted language** – runs directly in the browser without compilation.

**Single-threaded** – uses a single **call stack** to execute code.

**Synchronous by default** – but supports **asynchronous programming** using callbacks, promises, and `async/await`.

**Dynamically typed** – variable types are determined at runtime (no need to specify `int`, `string`, etc.).

**Multi-paradigm** – supports **procedural, object-oriented, and functional programming**.

**Event-driven** – uses an event loop to handle asynchronous tasks (`setTimeout`, `fetch`, etc.).

**Prototype-based** – inheritance is implemented using prototypes instead of classical classes (though ES6 classes exist syntactically).

**Interpreted by JS engines** – like V8 (Chrome/Node.js), SpiderMonkey (Firefox).

**Loosely typed** – allows implicit type conversions (type coercion).

## comparison of var, let, and const

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted & initialized as <code>undefined</code>	Hoisted but in <b>Temporal Dead Zone</b> (TDZ) until declaration	Hoisted but in <b>TDZ</b> until declaration
Re-declaration	✓ Allowed	✗ Not allowed in same scope	✗ Not allowed
Re-assignment	✓ Allowed	✓ Allowed	✗ Not allowed (but object properties/array elements <b>can</b> be mutated)
Best Use	✗ Avoid (old way)	✓ When value will change	✓ When value stays constant

# Code Example

## //Scope

```
if (true) {  
  var a = 10; // function  
  scoped  
  let b = 20; // block scoped  
  const c = 30; // block  
  scoped  
}
```

```
console.log(a); // ✓ 10  
// console.log(b); ✗  
ReferenceError  
// console.log(c); ✗  
ReferenceError
```

## Re-declaration

```
var x = 5;  
var x = 10; // ✓ Allowed  
console.log(x); // 10  
  
// let y = 5;  
// let y = 10; ✗ SyntaxError  
(cannot redeclare in same  
scope)
```

**//Same for const as it also  
a block scope variable**

## Re-assignment

```
let z = 100;  
z = 200; // ✓ Allowed
```

```
const PI = 3.14;  
// PI = 3.1415; ✗ TypeError  
(cannot reassign const)
```

//④ Mutating const object/array

```
const user = { name: "Alice" };  
user.name = "Bob"; // ✓ Allowed  
(property can change)  
console.log(user.name); // Bob
```

# JavaScript Data Types

## ◆ 1. Primitive Data Types (Stored by Value)

These are **immutable** (cannot be changed directly).

Data Type	Example	Notes
Number	<code>let x = 42;</code>	Includes integers & floats.
String	<code>let str = "Hello";</code>	Text, enclosed in quotes.
Boolean	<code>let flag = true;</code>	<code>true</code> or <code>false</code> .
Undefined	<code>let a;</code>	Declared but not assigned.
Null	<code>let b = null;</code>	Explicitly no value.
Symbol	<code>let id = Symbol();</code>	Unique and immutable.
BigInt	<code>let big = 123n;</code>	For very large integers.

## ◆ 2. Non-Primitive / Reference Data Types (Stored by Reference)

Data Type	Example
Object	<code>{ name: "John", age: 25 }</code>
Array	<code>["apple", "banana", "cherry"]</code>
Function	<code>function greet() { return "Hi"; }</code>

# Important Interview Points

**Primitive** → copied **by value**

**Objects/Arrays/Functions** → copied **by reference**

`typeof null` → returns "object" (famous tricky question)

`typeof NaN` → returns "number"

JS is **dynamically typed** → variable type can change at runtime.

```
console.log(typeof NaN); // "number"
```

```
console.log(NaN == NaN); // false
```



# Hoisting in JavaScript

- Hoisting is JavaScript's default behavior of **moving declarations to the top of their scope** (memory allocation phase) **before code execution**.
- This applies to **variables** and **functions**, but works differently for each.

## Key Points

- **Function Declarations** are **fully hoisted** (can be called before they are defined).
- **var** variables are **hoisted but initialized with `undefined`**.
- **let** and **const** are hoisted but kept in the **Temporal Dead Zone (TDZ)** until their line of declaration.
- **Function Expressions** and **Arrow Functions** are treated like variables (hoisted but not initialized).

# Code Examples

## // Function Hoisting

sayHello(); // ✅ Works, function is hoisted

```
function sayHello() {  
  console.log("Hello, World!");  
}
```

## //Function Expression & Arrow Function

// greet(); ❌ TypeError (greet is undefined at this point)

```
var greet = function () {  
  console.log("Hi!");  
};  
greet(); // ✅ Works after definition
```

## //Variable Hoisting with var

console.log(a); // ✅ undefined (hoisted but not initialized)  
var a = 10;  
console.log(a); // 10

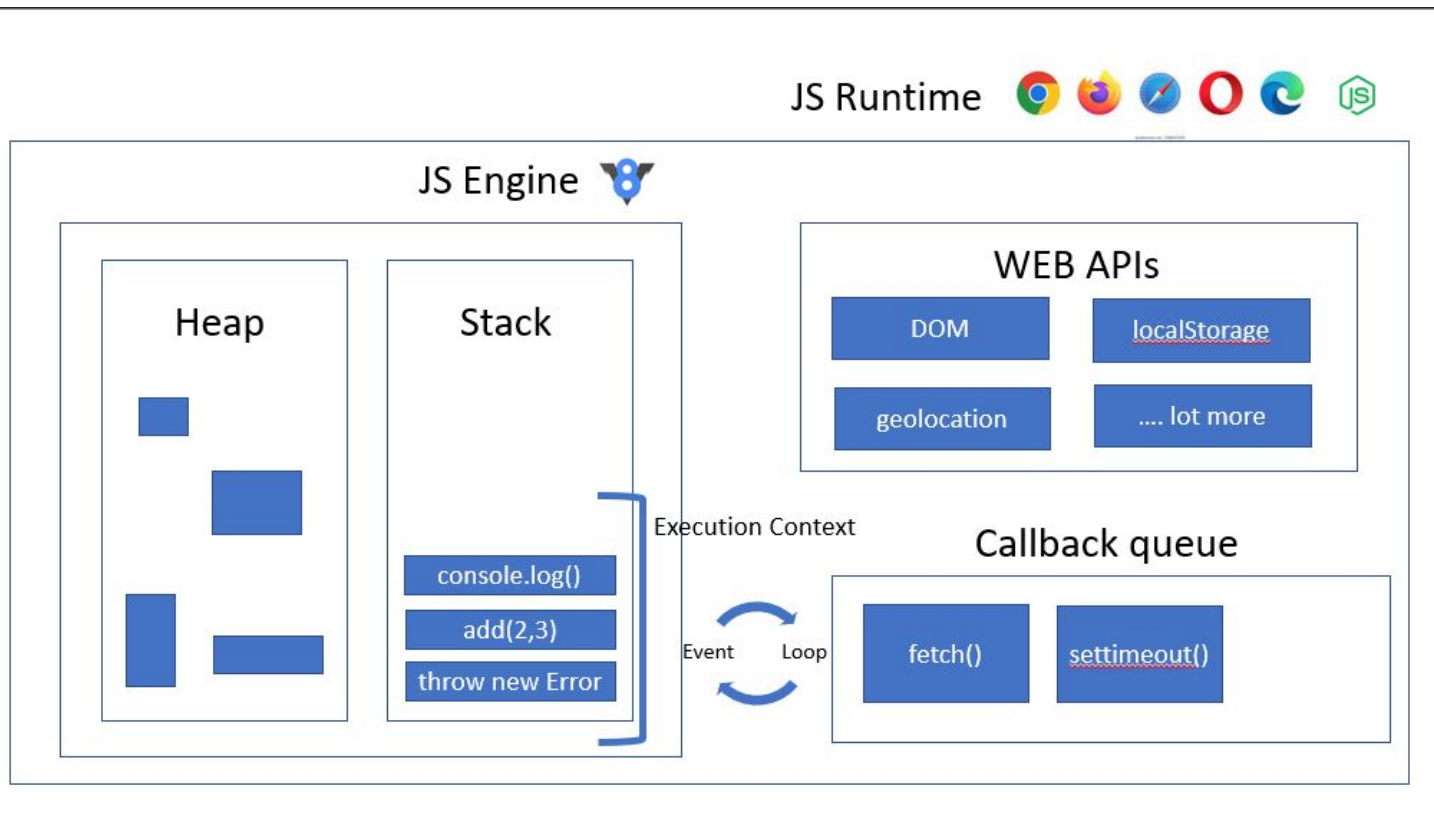
## // Hoisting with let / const (TDZ)

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

// console.log(c); ❌ ReferenceError (TDZ)  
const c = 30;  
console.log(c); // 30

The **Temporal Dead Zone** is the time between **when a variable is hoisted** and **when it is actually declared in the code**, during which you **cannot access it** — or you'll get a **ReferenceError**.

# visual representation of the JavaScript Runtime Environment





# Explanation of the above Image

## Main Parts

### 1. JS Engine (Left Side)

**Heap** – A place where objects and functions are stored in memory.

**Stack (Call Stack)** – Keeps track of what function is currently running.

-> Functions are added here when they are called.

-> Removed when they finish executing.

-> Works like a stack of plates (LIFO – last in, first out).

### 2. Web APIs (Right Side)

- Provided by the browser (or Node.js), **not by JS itself**.
- Examples: DOM, setTimeout, fetch, localStorage, geolocation.

### 3. Callback Queue (Bottom Right)

- Stores callbacks (functions) that are ready to run **after Web APIs finish their work**.
- Example: After `setTimeout()` finishes waiting, its callback goes here.

### 4. Event Loop (Middle)

- The event loop **keeps checking**:
  - “Is the call stack empty?”
  - If yes, it takes the first callback from the **Callback Queue** and puts it on the **Call Stack** to execute.
- This is what makes JS **asynchronous but non-blocking**.

# Step-by-Step Example

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

```
setTimeout(() => console.log("Timer Done"), 2000);
```

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

## How it runs:

1. `console.log("Start")` → goes to Call Stack → runs → removed.
2. `setTimeout()` → sent to Web APIs → waits 2 sec → callback moves to Callback Queue.
3. `console.log("End")` → runs immediately.
4. After Call Stack is empty, Event Loop takes callback from Queue → runs → prints **"Timer Done"**.

## Easy Analogy

Think of:

- **Call Stack** = Kitchen counter where dishes (functions) are cooked one by one.
- **Web APIs** = Extra helpers doing background tasks (boiling water, baking).
- **Callback Queue** = Wait line for dishes that are ready to be served.
- **Event Loop** = Waiter who serves dishes only when counter is free.

# Objects in JavaScript

**Definition:** An object is a collection of **key-value pairs** (properties + methods).

**Data Type:** Objects are **non-primitive** and stored in **heap memory** (reference type).

**Key-Value Structure:**

- Keys are always **strings** (or Symbols).
- Values can be anything: number, string, function, array, or even another object.

**Dynamic Nature:** You can **add, update, or delete** properties at runtime.

**Methods:** Functions inside objects are called **methods**.

**Access:** Properties can be accessed using **dot notation** (obj.key) or **bracket notation** (obj["key"]).

**Iteration:** Can be iterated with for...in, Object.keys(), Object.values(), Object.entries().

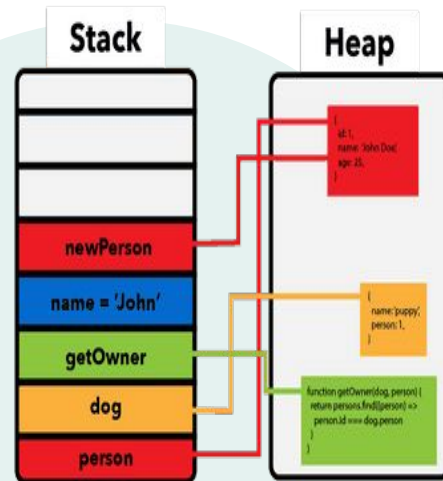
**Shallow vs Deep Copy:** Objects are copied by **reference**, not by value.

# Code Examples on Objects

## // Creating & Accessing an Object

```
const person = {  
  name: "Alice",  
  age: 25,  
  greet: function () {  
    console.log(`Hello, my name is ${this.name}`);  
  },  
};  
  
console.log(person.name);    // Dot notation → Alice  
console.log(person["age"]);  // Bracket notation → 25  
person.greet();              // Method → "Hello, my name is Alice"
```

```
const person = {  
  id: 1,  
  name: 'John',  
  age: 25,  
}  
  
const dog = {  
  name: 'puppy',  
  personId: 1,  
}  
  
function getOwner(dog, persons) {  
  return persons.find((person) =>  
    person.id === dog.person  
  )  
}  
  
const name = 'John';  
  
const newPerson = person;
```



# Code Examples on Objects

## // Adding / Updating / Deleting Properties

```
person.city = "Delhi";    // Add new property
person.age = 26;          // Update property
delete person.city;       // Delete property
```

```
console.log(person);
```

## // Iterating Over an Object

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

## // Copying Objects

```
const obj1 = { a: 1 };
const obj2 = obj1; // Reference copy (both point to same memory)
```

```
obj2.a = 10;
console.log(obj1.a); // 10 (changes reflect in both)
```

```
const clone = { ...obj1 }; // Spread operator → creates a shallow copy
```

# Arrays in JavaScript

- ❖ **Definition:** Arrays are **ordered collections** of values (elements).
- ❖ **Data Type:** Non-primitive, stored in **heap memory**.
- ❖ **Indexing:** Zero-based (first element is at index `0`).
- ❖ **Heterogeneous:** Can store **mixed data types** in a single array.
- ❖ **Dynamic:** Size can grow/shrink dynamically.
- ❖ **Access:** Elements are accessed using their index (`arr[0]`).
- ❖



# Code Example on Arrays

## // Creating Arrays

-> Creating an array using array literal syntax

```
const numbers = [1, 2, 3, 4];
```

-> Creating an array using the Array constructor

```
const fruits = new Array("Apple", "Mango", "Banana");
```

-> 'numbers' holds numeric values, while 'fruits' holds strings (fruit names)

## // Iterating

```
numbers.forEach(num => console.log(num)); // prints each element
```

## // Adding / Removing Elements

```
const arr = [1, 2, 3];  
arr.push(4);    // + Adds at end → [1, 2, 3, 4]  
arr.pop();      // - Removes from end → [1, 2, 3]  
arr.unshift(0); // + Adds at start → [0, 1, 2, 3]  
arr.shift();    // - Removes from start → [1, 2, 3]
```

## // Searching

```
const fruits = ["Apple", "Mango", "Banana"];  
console.log(fruits.indexOf("Mango")); // 1 (returns index)  
console.log(fruits.includes("Banana")); // true (boolean check)
```

# Important Methods of Arrays

```
const numbers = [1, 2, 3, 4, 5];
```

```
/*
```

**map()** → Used to transform each element of the array without changing the original array.

It returns a NEW array of the same length.

```
*/
```

```
const double = numbers.map(num => num * 2); // [2, 4, 6, 8, 10]
```

```
const numbers = [1, 2, 3, 4, 5];
```

```
/*
```

**filter()** → Used to filter elements based on a condition.

It returns a NEW array with only those elements that satisfy the condition.

```
*/
```

```
const even = numbers.filter(num => num % 2 === 0);
```

```
// [2, 4]
```

```
/*
```

**reduce()** → Used to "reduce" the entire array to a single value.

It takes an accumulator and a current value and keeps combining them.

```
*/
```

```
const sum = numbers.reduce((acc, num) => acc + num, 0); // 15
```



# Destructuring

Destructuring lets you **unpack values** from arrays or objects into separate variables.

Think of it as "unboxing" values quickly.

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

## // Traditional way

```
const first = numbers[0];
```

```
const second = numbers[1];
```

## // Using destructuring (short & clean)

```
const [a, b, c] = numbers;
```

```
console.log(a); // 10
```

```
console.log(b); // 20
```

```
console.log(c); // 30
```

# Spread Operator (...)

The spread operator is used to **expand or copy** values from arrays/objects.

```
const arr1 = [1, 2, 3];
```

```
const arr2 = [4, 5];
```

```
// Combine arrays
```

```
const combined = [...arr1, ...arr2];
```

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

## // Copying Arrays

```
const original = [1, 2, 3];
```

```
const copy = [...original]; // Creates a new copy
```

```
copy.push(4);
```

```
console.log(original); // [1, 2, 3] (unchanged)
```

```
console.log(copy); // [1, 2, 3, 4]
```

## // Spreading Objects

```
const user = { name: "Bob", age: 30 };
```

```
const updatedUser = { ...user, age: 31, city: "Delhi" };
```

```
console.log(updatedUser); // { name: "Bob", age: 31, city: "Delhi" }
```

# Shallow Copy

A **shallow copy** only copies the **top-level values**.

If the array/object has nested objects, it still points to the **same memory reference**.

So, changing nested data in the copy also changes it in the original.

```
const original = { name: "John", address: { city: "Delhi" } };
```

// Shallow copy using spread operator

```
const shallowCopy = { ...original };
```

```
shallowCopy.name = "Mike";    // ✅ Only changes copy  
(safe)
```

```
shallowCopy.address.city = "Pune"; // ⚠️ Changes original  
too (not safe)
```

```
console.log(original.address.city); // "Pune" (affected original)
```

# Deep Copy

A **deep copy** copies **everything**, including nested objects. It creates **completely new references**, so changing the copy does **not affect** the original.

```
const original = { name: "John", address: { city: "Delhi" } };
```

// Deep copy using JSON methods

```
const deepCopy = JSON.parse(JSON.stringify(original));
```

```
deepCopy.address.city = "Pune"; // ✅ Only changes copy
```

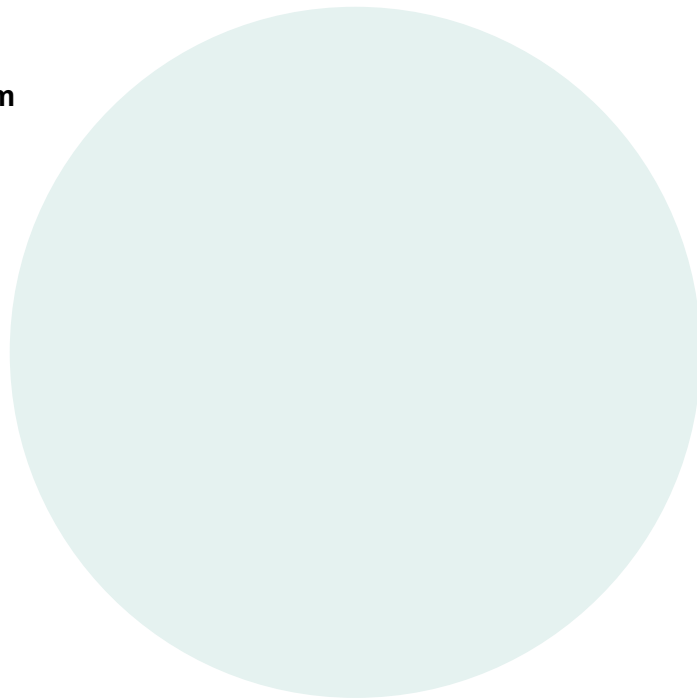
```
console.log(original.address.city); // "Delhi" (safe)
```

# Functions in JavaScript

-> A **function** is a reusable block of code designed to perform a specific task.

-> JavaScript treats functions as **first-class citizens** — meaning you can **store them in variables, pass them as arguments, and return them from other functions.**

- ❖ Functions create their own execution context and scope.
- ❖ Function declarations are hoisted, but function expressions are not.
- ❖ Arrow functions do not have their own this, arguments, or prototype.
- ❖ Functions can be pure (no side effects) or impure.



# Functions create their own Execution Context and Scope

Whenever a function is called, JavaScript creates a **new Execution Context** for it.

This context has its own **memory space (variables)** and **scope chain**.


```
function greet() {  
  const message = "Hello"; // This variable is only available inside greet()  
  console.log(message);  
}
```

```
greet();    // "Hello"  
console.log(message); // ❌ Error: message is not defined
```




# Function Declarations are Hoisted, but Function Expressions are Not

- **Hoisting** means moving declarations to the top of their scope **before code executes**.

sayHello(); //  Works, because function declarations are hoisted

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

- ❖ But **function expressions** are **not hoisted** (or are hoisted as undefined):

sayHi(); //  TypeError: sayHi is not a function

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

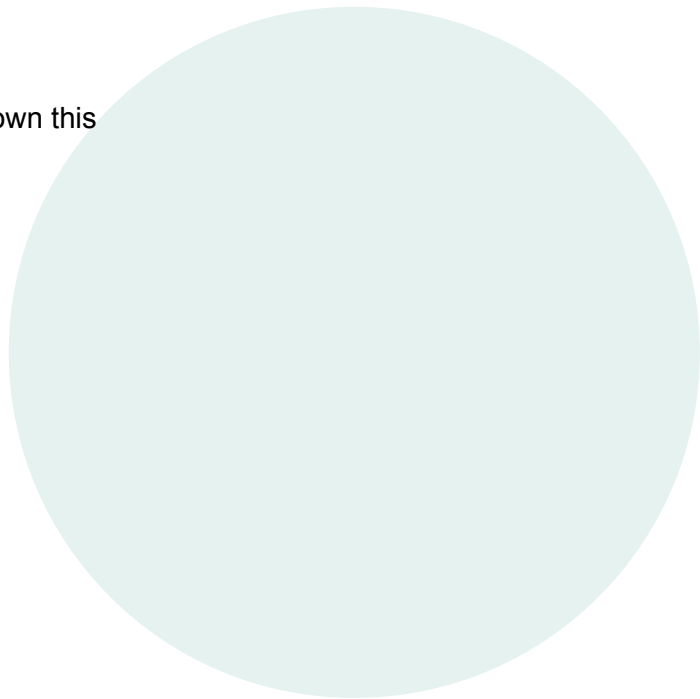
# Arrow Functions Do Not Have Their Own `this`, arguments, or prototype

Arrow functions are **lexically bound** — they use `this` from the surrounding scope.

```
const person = {  
  name: "Alice",  
  greet: () => {  
    console.log(this.name); // ❌ undefined, arrow function does not have its own this  
  }  
};  
person.greet();
```

## Compare with a regular function:

```
const person2 = {  
  name: "Bob",  
  greet() {  
    console.log(this.name); // ✅ "Bob"  
  }  
};  
person2.greet();
```



# Functions can be Pure (No Side Effects) or Impure

**Pure Function:** Always returns the same output for the same input and does not change anything outside itself.

**Impure Function:** May depend on external data or modify it.

```
// Pure Function
function add(a, b) {
  return a + b; // No side effects
}
```

```
// Impure Function
let total = 0;
function addToTotal(value) {
  total += value; // Changes outer variable (side effect)
}
```



# Functions Can Be Higher-Order

Functions in JS can take other functions as arguments or return them. These are called **higher-order functions**.

```
function greet(name) {  
  return function(message) {  
    console.log(`${message}, ${name}!`);  
  };  
}
```

```
const greetJohn = greet("John");  
greetJohn("Hello"); // "Hello, John!"
```





# Types of Functions in JavaScript

- **Function Declaration (Named Function)**

Defined using the function keyword with a name. **Hoisted** (can be called before it's defined).

```
function greet() {  
  console.log("Hello!");  
}  
greet(); // ✅ Works
```

- **Function Expression**

Function stored in a variable. **Not hoisted** (must be defined before calling).

```
const greet = function () {  
  console.log("Hello!");  
};  
greet(); // ✅ Works (after definition)
```

- **Arrow Function**

Shorter syntax for writing functions.

Does **not** have its own this, arguments, or prototype.

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

- **Anonymous Function**

A function **without a name**, usually used as a callback.

```
setTimeout(function () {  
  console.log("Executed after 2 seconds");  
}, 2000);
```

# Closure in JavaScript

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

In simple words: **A function bundled together with its surrounding state (lexical environment).**

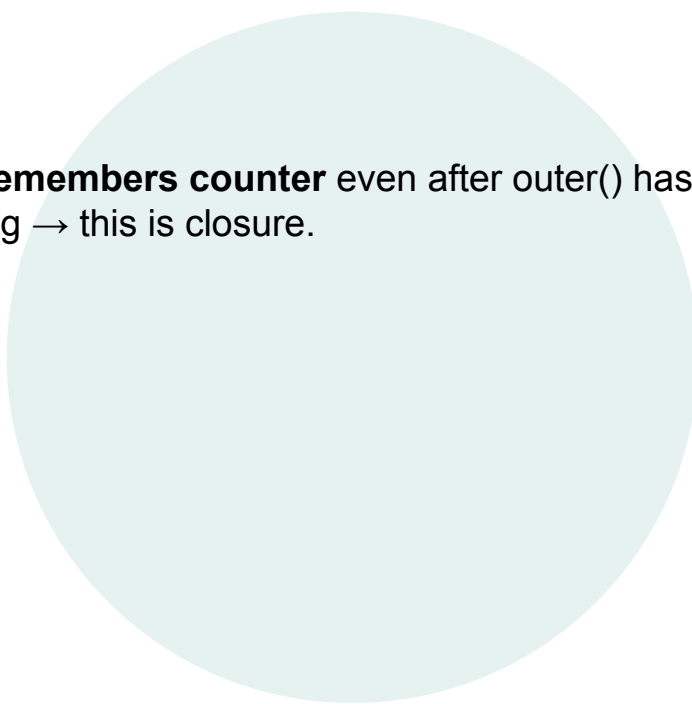
## Key Points

- Created **every time a function is defined**.
- Inner function can **access variables of outer function** even after outer function has returned.
- Used for **data privacy, encapsulation, and function factories**.
- Commonly appears in **interview questions** (like counter example).

# Basic Example

```
function outer() {  
  let counter = 0;  
  
  function inner() {  
    counter++;  
    console.log(counter);  
  }  
  
  return inner;  
}  
  
const increment = outer(); // outer is executed once  
increment(); // 1  
increment(); // 2  
increment(); // 3
```

Here, inner() **remembers counter** even after outer() has finished running → this is closure.



# What is DOM?

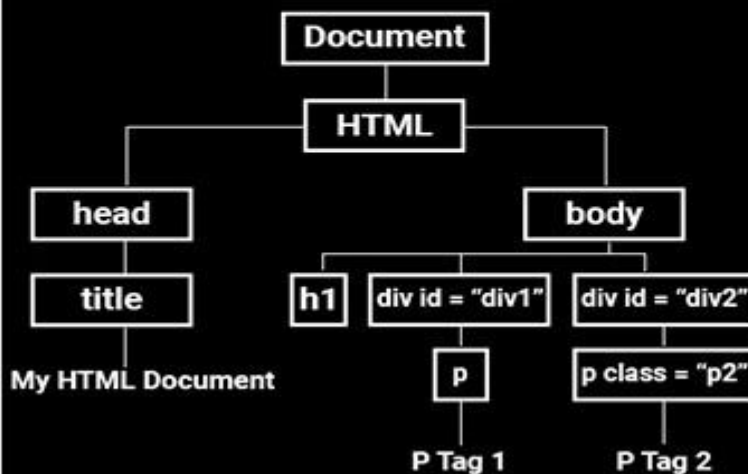
- **DOM (Document Object Model)** is a **tree-like representation** of your HTML page.
- JavaScript can use DOM to **read, change, add, or remove** elements dynamically.

## What is Document Object Model ?

### HTML Document

```
index.html x
1 <html>
2   <head>
3     <title>My HTML Document</title>
4   </head>
5
6   <body>
7     <h1>Heading</h1>
8     <div id="div1">
9       <p>P Tag 1</p>
10    </div>
11    <div id="div2">
12      <p class="p2">P Tag 2</p>
13    </div>
14  </body>
15 </html>
```

### Document Object Model (DOM)



# DOM Continued.....

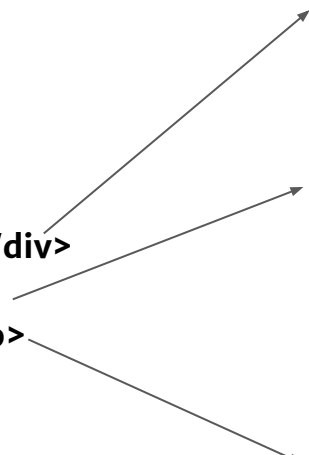
## Key Points

- DOM treats **HTML elements as objects**.
- Browser automatically creates DOM when a page loads.
- DOM can be **read & manipulated** using JavaScript.
- DOM has **methods & properties** to interact with elements.



# Accessing Elements in DOM

```
<div id="myDiv" class="box">Hello</div>  
<p class="text">First Paragraph</p>  
<p class="text">Second Paragraph</p>
```



// 1. By ID (returns single element)  
const div = document.getElementById("myDiv");

// 2. By Class Name (returns HTMLCollection)  
const texts =  
document.getElementsByClassName("text");

// 3. By Tag Name (returns HTMLCollection)  
const paragraphs =  
document.getElementsByTagName("p");

// 4. Using Query Selector (returns first match)  
const firstText = document.querySelector(".text");

// 5. Using Query Selector All (returns NodeList of all matches)  
const allTexts = document.querySelectorAll(".text");

# Manipulating the DOM

## Changing Content

```
div.textContent = "Hi there!"; // Changes text inside element  
div.innerHTML = "<b>Bold Text</b>"; // Can add HTML
```

## Changing Styles

```
div.style.color = "red";  
div.style.backgroundColor = "yellow";  
div.style.fontSize = "20px";
```

## Adding / Removing Classes

```
div.classList.add("active");  
div.classList.remove("box");  
div.classList.toggle("highlight"); // Adds if not present  
// , removes if present
```

## Creating & Appending Elements

```
const newPara = document.createElement("p");  
newPara.textContent = "This is a new paragraph!";  
document.body.appendChild(newPara); // Adds to the end of <body>
```

## Removing Elements

```
div.remove(); // Completely removes the element from DOM
```

## Handling Events

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

# alert, prompt, and confirm in JavaScript

## alert()

- Shows a **popup message** to the user.
- Only **displays information**, cannot get input.
- Stops code execution until user clicks "OK".

```
alert("Hello, Welcome to our site!");
```

**Use Case:** Notify users about something important.

Example: "Your form has been submitted successfully."

## prompt()

- Shows a **popup input box** where the user can type something.
- Returns the **user's input as a string**.
- If the user clicks "Cancel", it returns null.

```
const name = prompt("Enter your name:");  
console.log("Hello " + name);
```

**Use Case:** Ask the user for simple input quickly.

Example: Getting a username, age, or color preference.



## confirm()

- Shows a **popup with “OK” and “Cancel” buttons**.
- Returns **true** if user clicks OK, **false** if Cancel.

```
const isSure = confirm("Do you want to delete this item?");  
if (isSure) {  
  console.log("Item deleted!");  
} else {  
  console.log("Deletion canceled!");  
}
```

**Use Case:** Ask the user to **confirm an action** before proceeding.  
Example: Deleting a file, logging out, or submitting a form.



# Event Handling

**JavaScript Events** are **actions or occurrences** that happen in the browser. They can be triggered by various user interactions or by the browser itself.

## Event Types

JavaScript supports a variety of event types. Common categories include:

- **Mouse Events:** click, dblclick, mousemove, mouseover, mouseout
- **Keyboard Events:** keydown, keypress, keyup
- **Form Events:** submit, change, focus, blur
- **Window Events:** load, resize, scroll

```
<html>
<script>
  function myFun() {
    document.getElementById(
      "web").innerHTML =
      "Welcome to WebPage";
  }
</script>

<body>
  <button
    onclick="myFun()">Click
    me</button>
  <p id="web"></p>
</body>
</html>
```

## Event Handling Methods

### Inline HTML Handlers

```
<button onclick="alert('Button clicked!')">Click Me</button>
```

### DOM Property Handlers

```
let btn = document.getElementById("myButton");  
btn.onclick = () => {  
    alert("Button clicked!");  
};  
  
btn.addEventListener("click", () => {  
    alert("Button clicked using addEventListener!");  
});
```

# Event Propagation

When something happens on a webpage (like clicking a button), the browser checks **where** it happened and tells all the elements about it.

It happens in **two steps**:

1. **Capturing Phase (Going Down):**  
The event starts from the **top** of the page (document) and goes **down** through each parent element until it reaches the exact element you clicked.
2. **Bubbling Phase (Coming Up):**  
After reaching the clicked element, the event travels **back up** through the parent elements toward the top again.

## Preventing Default Behavior

Certain elements have default actions (e.g., links navigating to URLs). Use `preventDefault()` to override them.

```
document.querySelector("a").addEventListener("click", (e) => {  
  e.preventDefault();  
  console.log("Link click prevented");  
});
```

`preventDefault()` stops the link from navigating.

```
<html>
<body>
  <h2>Form Validation</h2>
  <form id="example">
    <input type="text" placeholder="Enter something" id="formInput" />
    <button type="submit">Submit</button>
  </form>
  <script>
    document.querySelector("#example").addEventListener("submit", (e) => {
      let input = document.querySelector("#formInput");
      if (!input.value) {
        e.preventDefault();
        alert("Input cannot be empty");
      }
    });
  </script>

</body>

</html>
```

# Closure in JavaScript

A **closure** is a function that retains access to its outer function's variables, even after the outer function has finished executing.

It "remembers" the environment in which it was created, allowing it to access variables outside its immediate scope.

```
function outer() {  
  let outerVar = "I'm in the outer scope!";  
  function inner() {  
    console.log(outerVar);  
    outerVar = "Updated"  
  }  
  return inner;  
}  
  
const closure = outer();  
closure();  
closure();
```

## Lexical Scoping

Closures rely on lexical scoping, meaning that a function's scope is determined by where the function is defined, not where it is executed. This allows inner functions to access variables from their outer function.

## Private Variables

Closures allow a function to keep variables hidden and only accessible within that function.

```
function counter() {  
    // Private variable  
    let count = 0;  
    return function () {  
        // Access and modify the private variable  
        count++;  
        return count;  
    };  
}  
  
const increment = counter();  
console.log(increment());
```



# Callbacks and Callback Hell in JavaScript

In JavaScript, callbacks are functions that are passed as arguments from one function to another and are executed after the completion of a certain task. They are commonly used in asynchronous operations, such as reading files, making HTTP requests, or handling user input.

- A function can accept another function as a parameter.
- Callbacks allow one function to call another at a later time.
- A callback function can execute after another function has finished.

JavaScript executes code line by line (synchronously), but sometimes we need to delay execution or wait for a task to complete before running the next function. Callbacks help achieve this by passing a function that is executed later.

```
function greet(name, callback) {  
  console.log(`Hello, ${name}!`);  
  callback(); // calling the callback function  
}
```

```
function afterGreet() {  
  console.log('Greeting is complete!');  
}
```

```
greet('Anjali', afterGreet);
```

## Callback Hell (Pyramid of Doom)

When multiple asynchronous operations depend on each other, callbacks get deeply nested, making the code hard to read and maintain.

```
function task1(callback) {  
  setTimeout(() => {  
    console.log("Task One completed");  
    callback();  
  },);  
}  
  
function task2(callback) {  
  setTimeout(() => {  
    console.log("Task Two completed");  
    callback();  
  },);  
}  
  
task1(function () {  
  task2(function () {  
    console.log("Both tasks completed");  
  });  
});
```

# Solution to Callback Hell

## Promises

Promises can help in avoiding the callback hell by providing the structured way to handle the asynchronous operations using the **.then() method**. Due to which the code becomes more readable by avoiding the deeply nested callbacks.

## Async/await

Async/await can help in avoiding the callback hell by writing the asynchronous code that looks like the synchronous code due to which the code becomes more cleaner and readable and reduces the complexity.

# JavaScript Promise

JavaScript Promises make handling asynchronous operations like API calls, file loading, or time delays easier. Think of a Promise as a placeholder for a value that will be available in the future. It can be in one of three states

- **Pending:** The task is in the initial state.
- **Fulfilled:** The task was completed successfully, and the result is available.
- **Rejected:** The task failed, and an error is provided.

```
let promise = new Promise((resolve, reject) => {
```

```
  // Perform async operation
```

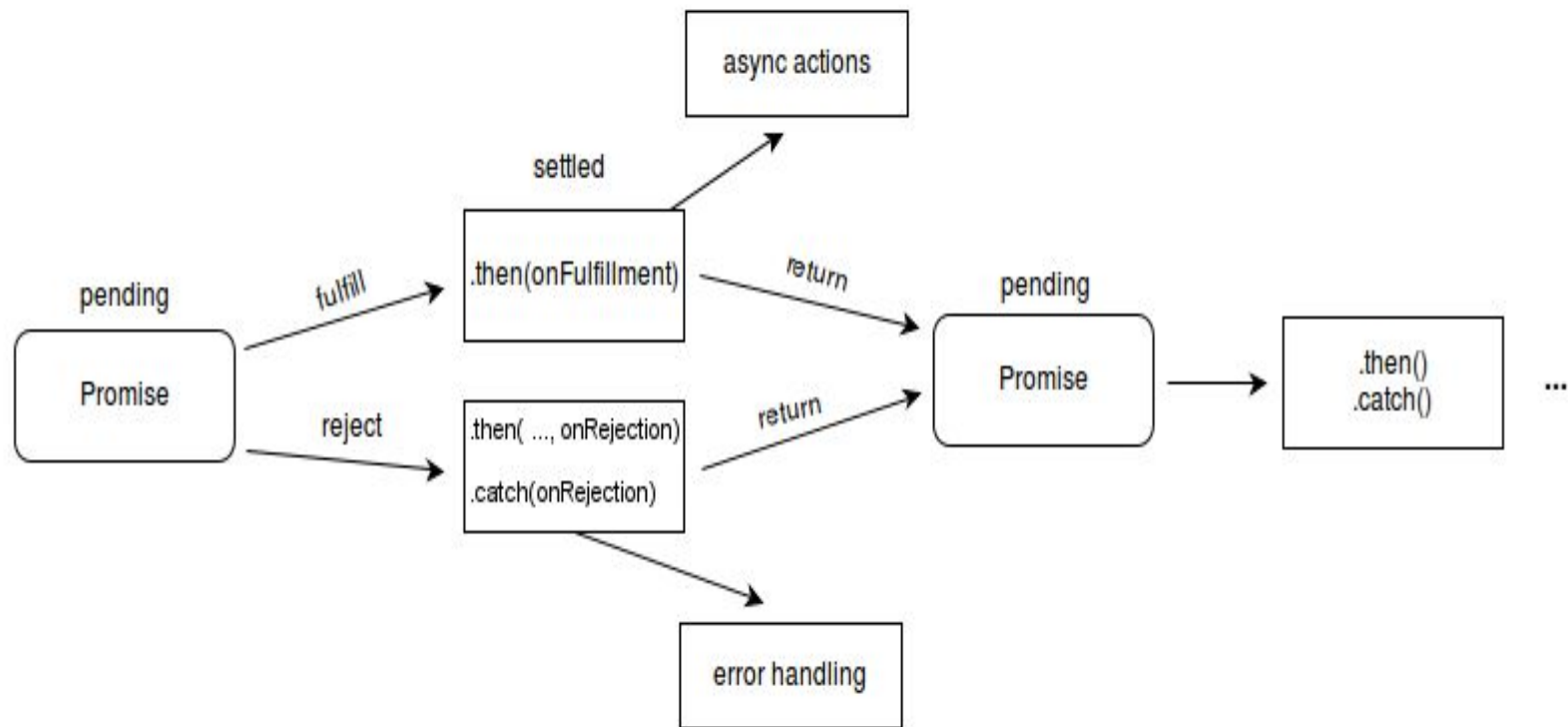
```
  if (operationSuccessful) {  
    resolve("Task successful");
```

```
  } else {  
    reject("Task failed");
```

```
  }
```

```
});
```

- **resolve(value)**: Marks the promise as fulfilled and provides a result.
- **reject(error)**: Marks the promise as rejected with an error.



## Chained Promises (JavaScript)

**then():** Runs when promise is fulfilled (can also take a rejection handler).

**catch():** Handles errors (same as `then(null, errorHandler)` but cleaner).

**finally():** Runs code after promise settles (success or error).

**Promises** can be chained: each `then()` passes its result to the next.



```
new Promise((resolve) => setTimeout(() => resolve("foo"),  
300))
```

```
.then(r => (console.log("A:", r), r + " -> A"))
```

```
.then(r => (console.log("B:", r), r + " -> B"))
```

```
.then(r => console.log("C:", r))
```

```
.catch(e => console.error("Error:", e))
```

```
.finally(() => console.log("Done!"));
```

## Promise Utility Methods (JavaScript)

- **Promise.all()** → Waits for **all promises to succeed**.
  - Returns an array of results if all succeed.
  - If **any fails**, the whole thing rejects.
- **Promise.race()** → Returns result of the **first promise to finish**.
  - Can be success or error — whichever settles first.
- **Promise.allSettled()** → Waits for **all promises to finish** (success or fail).
  - Returns an array with **status + value/reason** for each promise.
- **Promise.any()** → Waits for the **first successful promise**.
  - Ignores rejections (fails only if **all reject**).

# Async and Await in JavaScript

**Async and Await in JavaScript** are used to simplify handling asynchronous operations using promises. By enabling asynchronous code to appear synchronous, they enhance code readability and make it easier to manage complex asynchronous flows.

```
async function functionName() {  
  try {  
    const result = await someAsyncFunction();  
    console.log(result);  
  } catch (error) {  
    console.error("Error:", error.message);  
  }  
}
```

```
async function fetchData() {
```

```
  const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
```

```
  const data = await response.json();
```

```
  console.log(data);
```

```
}
```



## Async Function

The `async` function allows us to write promise-based code as if it were synchronous. This ensures that the execution thread is not blocked. Async functions always return a promise. If a value is returned that is not a promise, JavaScript automatically wraps it in a resolved promise.

```
async function myFunction() {  
  return "Hello";  
}
```

```
const getData = async () => {  
  let data = "Hello World";  
  return data;  
}
```

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

## await Keyword (JavaScript)

1. Used to wait for a Promise to resolve or reject.
2. Can only be used inside an async function.
3. Pauses execution until the Promise settles.
4. Makes async code look synchronous and cleaner.
5. Helps avoid complex .then() chaining.

```
const getData = async () => {  
  let y = await "Hello World";  
  console.log(y);  
}
```

```
console.log(1);  
getData();  
console.log(2);
```

## Error Handling in Async/Await

JavaScript provides predefined arguments for handling promises: resolve and reject.

- **resolve:** Used when an asynchronous task is completed successfully.
- **reject:** Used when an asynchronous task fails, providing the reason for failure.

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

# JavaScript Prototype

**Everything is an object** → functions, arrays, strings, etc.

JavaScript uses a **prototype-based system** (not class-based).

**Prototype = container** for shared methods & properties.

When you add something to a prototype, **all instances** get access to it.

Acts like a **mould** – instances are objects made from it.

Used for **inheritance, code reuse, and object relationships**.



**Every object has a prototype** – a template containing shared methods & properties.

**Prototype = common storage** → all instances can use what's defined here.

Adding a method/property to a prototype makes it **available to all objects**.

Helps **avoid code repetition** – define once, reuse everywhere.

Makes code **cleaner, efficient, and memory-friendly**.

```
Object.prototype.print = function () {  
  console.log('I am from object prototype')  
}  
let b = {  
  name: 'Pranjal',  
  age: 21  
}  
b.print()
```

# How Prototype Works in JavaScript?

Every object has an internal `[[Prototype]]` that points to another object.

This allows **inheritance** → objects can use properties & methods from their prototype.

## Property Lookup:

1. JavaScript first checks the object itself.
2. If not found, it checks the prototype.
3. Continues up the **prototype chain** until found or `null`.

**Functions have a `prototype` property** → used to define shared methods/properties.

**Objects created using a constructor function** inherit from that constructor's prototype.

You can **add new methods or properties** to a prototype at any time → all existing and future instances get access.

**Benefit:** Enables **code reuse**, avoids duplication, and makes objects share behavior efficiently.

Commonly used to **extend built-in objects** like `Array`, `Object`, etc.

# Creating Constructor Functions

A **special function** used to **create and initialize objects**.

Typically called with the **new keyword**.

Sets up **properties & methods** on the new object.

Allows creation of **multiple instances** with the same structure.

Instances can **share behavior** through the constructor's prototype.

```
function Person(name) {  
  this.name = name;  
}  
Person.prototype.sayHello = function () {  
  console.log(`Hello, my name is ${this.name}.`);  
};  
const n = new Person("Sheema");  
n.sayHello();
```

## Adding method to prototype

Defines a **sum method** on `Array.prototype`.

**sum** calculates the **total of all array elements**.

Can be used by **any array** (because it's on the prototype).

Calls `sum()` on two arrays (`arr` and `arr2`).

Logs the **calculated results** to the console.

```
let a1 = [1, 2, 3, 4, 5]
let a2 = [5, 6, 7, 8, 9]
Array.prototype.sum = function () {
  let sum = 0
  for (let i = 0; i < this.length; i++) {
    sum += this[i]
  }
  return sum
}
console.log(a1.sum())
console.log(a2.sum())
```

# Prototype Inheritance

**Child constructor** inherits from **Parent constructor**.

Instances of **Child** get access to **Parent's prototype methods**.

This allows **code reuse** and shared behavior.

You can add **custom methods** to the child's prototype (e.g., `caste()`), and all child instances can use them.

Creates a **prototype chain** → lookup goes from child → parent → Object.

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

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

```
function Dog(name) {  
  Animal.call(this, name); // Call the parent  
  constructor  
}
```

```
\  
Dog.prototype =  
Object.create(Animal.prototype); // Set up  
inheritance
```

```
Dog.prototype.constructor = Dog;  
Dog.prototype.speak = function () {  
  console.log(`${this.name} barks.`);  
};
```

```
const dog = new Dog('Rex');  
dog.speak();
```

# Regular expressions

**Regular Expressions (Regex):** Patterns used to find, match, or manipulate text in strings.

In JavaScript, RegEx are **objects** that define these patterns.

Commonly used with:

- **RegExp methods:** `exec()`, `test()`
- **String methods:** `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`, `split()`

Help with **searching, validation, and text manipulation.**

Provide a **powerful syntax** for complex pattern matching.

# What is React?

**React** – a fast, scalable JavaScript library for front-end development.

**Built by Facebook** to create interactive UIs.

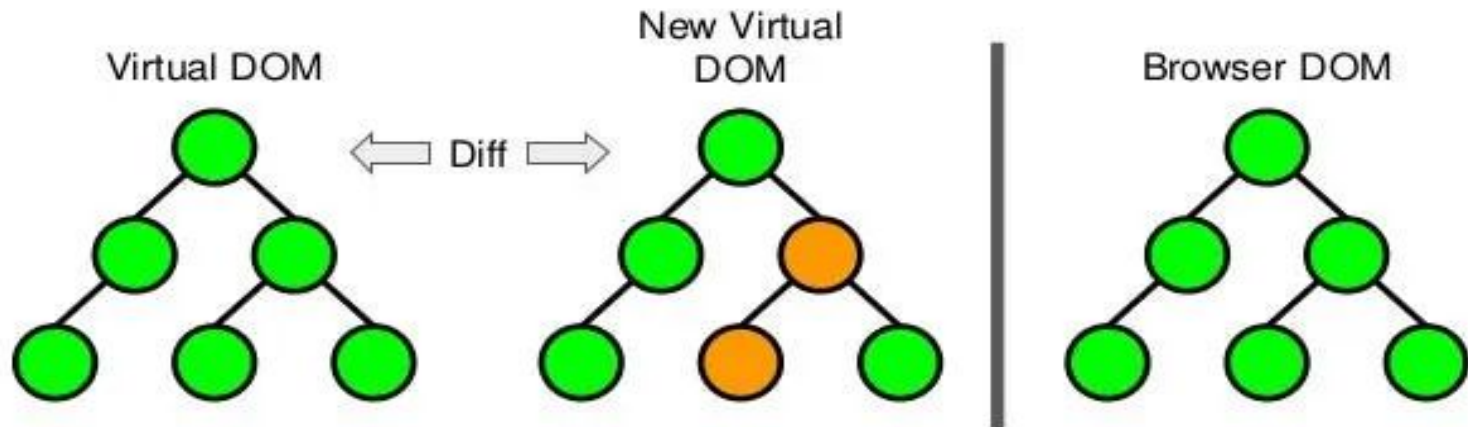
**Component-based** – encourages reusability and modular design.

**SPA support** – smooth navigation without page reloads.

**Virtual DOM** – enables efficient, optimized UI updates.

**Faster rendering** – delivers a seamless user experience.

# Reconciliation





# Introduction to Props

Props = "**Properties**", used to pass data from **parent** → **child** component

**Read-only** (immutable) – cannot be changed by child component

Make components **reusable & dynamic**

```
const Welcome = ({ name }) => <h1>Hello, {name}</h1>;  
<Welcome name="John" />
```

# Prop Validation & Defaults

**Warns in console if wrong type/missing required prop.**

```
Child.propTypes = {  
  name: PropTypes.string.isRequired,  
  age: PropTypes.number,  
};
```

**Default Props (fallback values):**

Default Props (fallback values):

```
MyComp.defaultProps = {  
  name: "Guest",  
  age: 18,  
};
```

# React Props

## Drilling

```
const App = () => {  
  const userName = "John Doe"; // data in the top-level component  
  
  return <Parent userName={userName} />;  
};  
  
const Parent = ({ userName }) => {  
  return <Child userName={userName} />;  
};  
  
const Child = ({ userName }) => {  
  return <GrandChild userName={userName} />;  
};  
  
const GrandChild = ({ userName }) => {  
  return <h1>Hello, {userName}</h1>;  
};  
  
export default App;
```









