💡 JavaScript Lecture 1: Values, Operations, and Variables

✅ 1️⃣ What are Values?

In JavaScript, a **value** is any piece of data.
It can be a:

Type	Example	Description
Number	`5`, `100`, `3.14`	Numeric values (integer or float)
String	`"Hello"` `'JS'`	Text, written inside quotes
Boolean	`true`, `false`	Logic values (Yes/No)
Undefined	`undefined`	Means: "No value assigned yet"
Null	`null`	Intentionally empty
Object	`{ name: "John" }`	Key-value pairs
Array	`[1, 2, 3]`	List of values

✅ 2️⃣ What are Operations?

Operations are actions you perform on values.

💪 Mathematical Operations

```
console.log(5 + 3);  // Addition → Output: 8
console.log(10 - 4); // Subtraction → Output: 6
console.log(6 * 2);  // Multiplication → Output: 12
console.log(8 / 2);  // Division → Output: 4
```

💬 String Operations

```
console.log("Hello " + "World!");
// Output: Hello World!
```

This is called **String Concatenation**.

🔥 Comparison Operations

```
console.log(5 > 3);   // true
console.log(5 == 5);  // true
console.log(5 != 4);  // true
```

These return **Boolean (true/false)**.

🧠 Logical Operations

```
console.log(true && false); // false
console.log(true || false); // true
console.log(!true);         // false
```

Used in decision-making (like `if` statements).

✅ 3️⃣ What are Variables?

A **variable** stores a value so you can use it later.

Think of it like a container 📦.

💡 Declaring Variables

In JavaScript, you declare a variable using:

Keyword	Purpose
`var`	Old way (avoid this)
`let`	Modern, changeable
`const`	Fixed (cannot be reassigned)

💡 Example:

```
let name = "John";
const birthYear = 2000;
let age = 2025 - birthYear;

console.log(name);      // Output: John
console.log(age);       // Output: 25
```

⚡ Rules for Naming Variables

1. Can contain letters, numbers, `$`, or `_`.
2. Cannot start with a number.
3. Should be meaningful.
4. Case-sensitive (`age` ≠ `Age`).

💡 Why Use Variables?

- Easy to store values.
- Easy to reuse values.
- Easy to update values.
- Code is clean and readable.

✅ Summary:

- Values are the actual data (numbers, strings, booleans).
- Operations allow you to manipulate these values.
- Variables help you store values for later use.

💡 **Practice Task for You:**

```
let myName = "Your Name";
let year = 2025;
let myAge = year - 2000;

console.log("My name is " + myName + " and I am " + myAge + " years old.");
```

Excellent — great progress! Let's go deep into:

💡 JavaScript Lecture 2: Decisions and Loops

✅ 1️⃣ Decision Making — `if`, `else if`, `else`

In JavaScript, decision-making helps you write code that can **choose different paths** based on conditions.

💡 Syntax:

```
if (condition) {
    // Code runs if condition is true
} else if (another condition) {
    // Code runs if this new condition is true
} else {
    // Code runs if no conditions are true
}
```

💡 Example:

```
let age = 20;

if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

➡️ Output: `You are an adult.`

💡 Multiple Conditions Example:

```
let score = 85;

if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
} else if (score >= 70) {
    console.log("Grade: C");
} else {
    console.log("Grade: F");
}
```

➡️ Output: `Grade: B`

✅ 2️⃣ Logical Operators in Decisions

Operator	Meaning	Example
`&&`	AND — both must be true	`a > 0 && b > 0`
`` ` ``		`` ` ``
`!`	NOT — reverse the value	`!true` is `false`

💡 Example:

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

if (age >= 18 && hasLicense) {
    console.log("You can drive.");
} else {
    console.log("You cannot drive.");
}
```

➡️ Output: `You can drive.`

✅ 3️⃣ Loops — Repeating Code

Loops allow you to **run the same block of code multiple times** until a condition is false.

🔁 For Loop

Syntax:

```
for (initialization; condition; update) {
    // code to repeat
}
```

💡 Example:

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

➡️ Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

🔁 While Loop

Syntax:

```
while (condition) {
    // code to repeat
}
```

💡 Example:

```
let i = 1;
while (i <= 5) {
```

```
    console.log("While Number: " + i);
    i++;
}
```

➡️ Same output as the `for` loop.

🔁 Do-While Loop

This loop runs the code at least **once** before checking the condition.

Syntax:

```
do {
    // code to repeat
} while (condition);
```

💡 **Example:**

```
let i = 1;
do {
    console.log("Do-While Number: " + i);
    i++;
} while (i <= 5);
```

➡️ Same output: 1 to 5.

⚡ When to Use Which Loop?

Loop Type	Best When...
`for`	You know how many times to repeat
`while`	You don't know how many times, stop on condition
`do-while`	You want the code to run **at least once**

✅ **Summary:**

- `if / else` helps your program make decisions.

- Loops repeat tasks until a condition is met.
- Logical operators help you write smart conditions.

Excellent — this is a super important JavaScript concept! Let's go deep:

💡 JavaScript Lecture 3: Arrays, Indexing & Accessing Elements

---

✅ 1️⃣ What is an Array?

An **Array** is a special variable that can hold **multiple values** in a single container.

Imagine it like a **list** or a **collection** — instead of creating separate variables for each value, you can store them all in one place!

💡 Syntax:

let arrayName = [value1, value2, value3, value4];

💡 Example:

let fruits = ["Apple", "Banana", "Mango", "Orange"];

In this example, the `fruits` array contains **4 string values**.

✅ 2️⃣ What is an Index in an Array?

Every value inside an array is stored at a position called its **Index**.
👉 **Index always starts from 0** (zero-based numbering).

Index	Value
0	"Apple"
1	"Banana"
2	"Mango"
3	"Orange"

✅ 3️⃣ Accessing Elements from an Array

You can get the values from an array by using:

arrayName[index]

💡 Example:

let fruits = ["Apple", "Banana", "Mango", "Orange"];

console.log(fruits[0]); // Output: Apple
console.log(fruits[2]); // Output: Mango

⚠️ If you try an invalid index:

console.log(fruits[4]); // Output: undefined (index 4 does not exist)

✅ 4️⃣ Changing Array Elements

You can **update** an element at a specific index:

fruits[1] = "Pineapple";
console.log(fruits);
// Output: ["Apple", "Pineapple", "Mango", "Orange"]

✅ 5️⃣ Array Length

To check how many items an array holds:

```
console.log(fruits.length);
// Output: 4
```

✅ 6️⃣ Looping through an Array

You can use a `for` loop to go through each element.

💡 **Example:**

```
let fruits = ["Apple", "Banana", "Mango", "Orange"];

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

➡️ Output:

```
Apple
Banana
Mango
Orange
```

✅ **Summary:**

- An **Array** holds multiple values.
- Values are stored using **Index Numbers** (starting from 0).
- You can access, update, and loop through elements.

💡 **Practice Task for You:**

```
let numbers = [10, 20, 30, 40, 50];
```

```
// Print the third number
console.log(numbers[2]);

// Update the last number to 99
numbers[4] = 99;

// Print all numbers using a loop
for (let i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);
}
```

Great! Now you're moving into the real power of arrays:

💡 JavaScript Lecture 4: Array Methods

An **Array Method** is a **built-in JavaScript function** that helps you work with arrays more easily — like adding, removing, or transforming items.

Let's go through the most important ones! 🧠💪

✅ 1️⃣ `push()` — Add to End

Adds a new value to the end of the array.

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

fruits.push("Mango");

console.log(fruits);

// Output: ["Apple", "Banana", "Mango"]
```

✅ 2️⃣ `pop()` — Remove from End

Removes the last element from the array.

fruits.pop();

console.log(fruits);

// Output: ["Apple", "Banana"]

✅ 3 `unshift()` — Add to Start

Adds a new value at the beginning of the array.

fruits.unshift("Pineapple");

console.log(fruits);

// Output: ["Pineapple", "Apple", "Banana"]

✅ 4 `shift()` — Remove from Start

Removes the first element.

fruits.shift();

console.log(fruits);

// Output: ["Apple", "Banana"]

✅ 5 `indexOf()` — Find Index

Returns the index number of a value (or `-1` if not found).

let position = fruits.indexOf("Banana");

console.log(position); // Output: 1

✅ 6 `includes()` — Check if Exists

Returns `true` if the value is found, `false` if not.

console.log(fruits.includes("Mango")); // Output: false

console.log(fruits.includes("Apple")); // Output: true

✅ 7 `join()` — Convert to String

Joins all elements into one string.

let result = fruits.join(", ");

console.log(result);

// Output: "Apple, Banana"

✅ 8 `slice()` — Copy Parts of Array

Creates a new array by slicing out a part (does not change original).

let newFruits = fruits.slice(0, 2);

console.log(newFruits); // Output: ["Apple", "Banana"]

✅ 9 `splice()` — Add/Remove Elements

Modifies the original array.

Syntax:

```
array.splice(startIndex, deleteCount, newItem1, newItem2...)
```

fruits.splice(1, 0, "Grapes");

console.log(fruits);

// Output: ["Apple", "Grapes", "Banana"]

✅ 🔁 Bonus: Looping with `forEach()`

Loops over each item in the array.

fruits.forEach(function(fruit) {

  console.log(fruit);

});

✅ **Summary Table:**

Method	Purpose
`push()`	Add to end
`pop()`	Remove from end
`unshift()`	Add to start
`shift()`	Remove from start

`indexOf()`	Find index of a value
`includes()`	Check if value exists
`join()`	Combine into string
`slice()`	Copy part of array
`splice()`	Add/Remove elements
`forEach()`	Loop through elements

💡 **Practice Task for You:**

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

// Add 50 to the end
numbers.push(50);

// Remove the first number
numbers.shift();

// Check if 30 exists
console.log(numbers.includes(30));  // true
```

```
// Slice the first two items into a new array
let newArray = numbers.slice(0, 2);
console.log(newArray);

// Loop through and print all numbers
numbers.forEach(function(num) {
    console.log("Number: " + num);
});
```

Awesome! Functions are one of the most important concepts in JavaScript. Let's dive in:

💡 JavaScript Lecture 5: Functions and Function Declaration

✅ 1️⃣ Why Functions?

Functions allow you to **group code** into a **single unit** that can be reused multiple times.

Why use functions?

- **Reusability:** Write the code once and reuse it.
- **Organization:** Break your code into smaller, manageable chunks.
- **Avoid Repetition:** Don't repeat yourself! Functions can save you from writing the same code again and again.
- **Maintainability:** You can update a function in one place, and it affects the entire program.
- **Clarity:** Functions make the code more readable and easy to understand.

✅ 2️⃣ What is a Function?

A **function** is a **block of reusable code** that performs a specific task.

💡 Function Declaration Syntax:

```
function functionName(parameter1, parameter2, ...) {
    // Code to execute
    return result;  // Optional (returns a value)
}
```

- `functionName` is the name of the function.
- `parameter1, parameter2, ...` are the inputs the function takes.
- `return` is used to send back the result of the function.

💡 Example:

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

- Here, `greet` is a function that takes a `name` as an argument and logs a greeting.

⚡ Function with Return Value:

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

```
}

let result = add(5, 3);

console.log(result);  // Output: 8
```

- This function adds two numbers and **returns** the result.
- We store the returned value in `result`.

✅ 3️⃣ Function Parameters and Arguments

- **Parameters** are the variables listed inside the parentheses in the function declaration.
- **Arguments** are the actual values you pass when calling the function.

```
function multiply(x, y) {  // x and y are parameters
    return x * y;
}


let product = multiply(4, 5);  // 4 and 5 are arguments

console.log(product);  // Output: 20
```

✅ 4️⃣ Function Expressions

Functions can also be created using **function expressions** (anonymous functions), which are assigned to variables.

💡 Syntax:

```
let functionName = function(parameter1, parameter2, ...) {
    // Code to execute
    return result;
};
```

💡 Example:

```
let square = function(number) {
    return number * number;
};
console.log(square(4));  // Output: 16
```

- Here, the `square` function is created as an **anonymous function** and assigned to the variable `square`.

✅ 5️⃣ Why Use Functions?

Let's recap **why functions are important**:

- They **encapsulate logic**, making the code more **modular**.
- They can help you **avoid repetition** in your code.
- They allow you to **return values** that can be used in other parts of your program.

✅ 6️⃣ Function Scopes

Variables inside a function are **local** to that function, meaning they cannot be accessed outside of it.

```
function example() {
    let message = "Inside function";  // local variable
    console.log(message);
}

example();  // Output: Inside function
console.log(message);  // Error: message is not defined
```

Summary Table:

Concept	Description
Function Declaration	Define a function with `function` keyword.
Function Expression	Assign an anonymous function to a variable.
Parameters	Inputs that the function takes.
Arguments	Actual values passed to the function.
Return	A value that a function sends back.
Scope	Variables declared inside a function are local.

💡 Practice Task for You:

```
// 1. Declare a function that subtracts two numbers

function subtract(a, b) {

    return a - b;

}


// 2. Call the function with arguments 10 and 4

let result = subtract(10, 4);
```

```
console.log(result);  // Output: 6


// 3. Declare a function with a return value and use it
function greet(name) {
    return "Hello, " + name + "!";
}


let greetingMessage = greet("Alice");
console.log(greetingMessage);  // Output: Hello, Alice!
```

Perfect! Let’s go step by step — this is a simple but very important piece of JavaScript knowledge.

💡 JavaScript Lecture 6: Functions Without Parameters & Return

---

✅ 1️⃣ What is a Function Without Parameters?

A **function without parameters** is a function that does **not** accept any input when called.

It simply runs the block of code you wrote inside it.

💡 Example:

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


sayHello();  // Output: Hello, World!
```

✅ **Explanation:**

- `sayHello` is the function name.
- It has **no parameters** inside the `()`.
- When called, it simply runs the code inside `{}`.

✅ 2️⃣ What is a Function Without Return?

A function without `return` means the function **does not send back a result** — it just performs an action (like printing something or changing a value).

💡 Example:

```
function greetUser() {
    console.log("Welcome to JavaScript!");
}

greetUser();  // Output: Welcome to JavaScript!
```

✅ **Explanation:**

- This function performs a task (`console.log`) but **does not return any value**.
- If you try to store it in a variable:

```
let result = greetUser();
console.log(result);  // Output: undefined
```

because there is no `return` statement.

✅ 3️⃣ Full Example:

```
function printLine() {
    console.log("------------");
}

function welcomeMessage() {
    printLine();  // calling another function
    console.log("Welcome to my program!");
    printLine();
}

welcomeMessage();
```

➡️ **Output:**

```
------------
Welcome to my program!
```

✅ **Explanation:**

- `printLine` has no parameters and no return.
- `welcomeMessage` calls `printLine()` before and after printing the message.

✅ 4️⃣ Why Use Functions Without Parameters & Return?

- When you want to **repeat the same action** many times without needing input.
- Useful for **logging**, **displaying messages**, **formatting**, or **executing fixed logic**.

✅ Summary Table:

Function Type	Description
No Parameters, No Return	Runs a fixed block of code; no input, no output.
Example	`function greet() { console.log("Hi!"); }`

💡 **Practice Task for You:**

```
function displayWelcome() {
    console.log("=== Welcome to JavaScript World ===");
}

function showEndMessage() {
    console.log("=== End of Program ===");
}

// Call both functions
displayWelcome();
showEndMessage();
```

Excellent! You're learning this in the right order — this is one of the most useful and powerful parts of JavaScript functions.

💡 JavaScript Lecture 7: Function with Parameters (Single & Multiple)

✅ 1️⃣ What is a Parameter?

A **parameter** is like a **placeholder** for a value — it allows you to pass data into a function when you call it.

> 💡 Think of it like:
> 👉 "A function **accepts parameters** and **processes values**."

✅ 2️⃣ Function With a Single Parameter

This means the function takes **one input** when it is called.

💡 Syntax:

```
function functionName(parameter) {
    // code using the parameter
}
```

💡 Example:

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


greet("Alice");   // Output: Hello, Alice!
greet("Bob");     // Output: Hello, Bob!
```

✅ Explanation:

- `name` is the parameter.
- When you call the function, you pass a value like `"Alice"` or `"Bob"` — and the function uses it.

✅ 3️⃣ Function With Multiple Parameters

You can pass **more than one parameter** if the function needs more data.

💡 Syntax:

```
function functionName(parameter1, parameter2) {
    // code using parameter1 and parameter2
}
```

💡 Example:

```
function add(a, b) {
    console.log("The sum is: " + (a + b));
}


add(3, 4);   // Output: The sum is: 7
add(10, 20); // Output: The sum is: 30
```

✅ Explanation:

- `a` and `b` are parameters.
- You pass values like `3, 4` or `10, 20` when calling the function.

✅ 4️⃣ Function With Multiple Parameters + Return

You can also calculate and return the result instead of just printing it:

```
function multiply(x, y) {
    return x * y;
}


let result = multiply(5, 3);
console.log("Multiplication Result: " + result);  // Output: 15
```

✅ 5️⃣ Important Notes:

- The **order** of parameters matters.
- You can pass any **data type** (number, string, boolean, array, object).
- If you skip an argument when calling, it will become `undefined`.

⚡ Example of Missing Argument:

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

✅ 6️⃣ Real-World Example

```
function calculateArea(length, width) {
    return length * width;
}
let area = calculateArea(5, 10);
console.log("Area is: " + area);  // Output: Area is: 50
```

✅ Summary Table:

Function Type	Description	Example
Single Parameter	Accepts one input	`function greet(name)`

Multiple Parameters	Accepts two or more inputs	`function add(a, b)`
Missing Argument	Results in `undefined`	`greet();`

💡 **Practice Task for You:**

```
// 1. Create a function that takes a number and prints its square.
function square(num) {
    console.log("Square is: " + (num * num));
}
square(5);  // Output: Square is: 25

// 2. Create a function that adds three numbers and returns the result.
function addThree(a, b, c) {
    return a + b + c;
}
let sum = addThree(2, 4, 6);
console.log("Sum is: " + sum);  // Output: Sum is: 12
```

Great topic — you're moving forward like a pro!
Let's break it down clearly:

💡 JavaScript Lecture 8: Function with Unlimited Number of Parameters

✅ 1️⃣ Why Use Unlimited Parameters?

Sometimes you don't know how many values will be passed to your function.
For example: a `sum` function could add 2 numbers, 3 numbers, or even 10 numbers!

Instead of writing:

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

function addThree(a, b, c) { return a + b + c; }

You can write **one single flexible function**.

✅ 2️⃣ Using `arguments` Object (Old Way)

All JavaScript functions have a hidden object called `arguments`.
It looks like an array, and holds all the values passed to the function.

💡 Example:

```
function showArguments() {
    console.log(arguments);
}

showArguments(1, 2, 3, 4, 5);
// Output: [1, 2, 3, 4, 5]
```

💡 Sum Example:

```
function sum() {
    let total = 0;
    for(let i = 0; i < arguments.length; i++) {
        total += arguments[i];
```

```
    }
    return total;
}

console.log(sum(1, 2, 3));         // Output: 6
console.log(sum(5, 10, 15, 20));   // Output: 50
```

✅ **Note:**

- `arguments` is an **array-like object**.
- You can loop through it, but cannot use `.map()`, `.forEach()`, etc. (because it's not a true array).

✅ 3️⃣ Using Rest Parameters (Modern & Recommended Way)

In modern JavaScript (ES6+), we use `...` (called "rest parameter").

💡 Syntax:

```
function functionName(...parameterName) {
    // parameterName is an array of all passed arguments
}
```

💡 Example:

```
function sumAll(...numbers) {
    let total = 0;
    for(let number of numbers) {
        total += number;
    }
```

```
    return total;
}

console.log(sumAll(2, 4));          // Output: 6
console.log(sumAll(1, 2, 3, 4, 5, 6));  // Output: 21
```

✅ **Explanation:**

- `...numbers` collects all arguments into an **array**.
- You can loop over it like a normal array.

✅ 4 Difference: `arguments` vs `rest parameters`

Feature	`arguments`	`...rest`
Type	Array-like (not true array)	Real JavaScript Array
Modern Syntax	❌ Old	✅ Modern ES6+
Methods Supported	Cannot use array methods	Can use `.map()`, `.filter()`, etc

---

✅ 5 Real-World Example:

💡 Calculate Average:

```
function calculateAverage(...numbers) {
    let total = 0;
```

```
    for(let num of numbers) {
        total += num;
    }
    return total / numbers.length;
}

console.log(calculateAverage(10, 20, 30));  // Output: 20
console.log(calculateAverage(100, 200));    // Output: 150
```

✅ Summary:

- `arguments` — old way, not an array.
- `...rest` — new way, gives you a real array.
- Allows you to create flexible, reusable functions that work with **any number of arguments**.

💡 **Practice Task for You:**

```
function multiplyAll(...numbers) {
    let result = 1;
    for(let num of numbers) {
        result *= num;
    }
    return result;
}
```

console.log(multiplyAll(1, 2, 3, 4)); // Output: 24

Excellent! You've now reached one of the coolest and most modern parts of JavaScript!
Let's dive in — step by step.

💡 JavaScript Lecture 9: Arrow Functions (=>)

✅ 1️⃣ What is an Arrow Function?

Arrow functions are a **short and clean way** to write functions in JavaScript.
They were introduced in **ES6 (2015)**.

Instead of writing:

function greet(name) {

    return "Hello, " + name;

}

You can write:

const greet = (name) => "Hello, " + name;

👉 **Same result — less code!**

✅ 2️⃣ Syntax Comparison

Traditional Function	Arrow Function
`function add(a, b) { return a + b; }`	`const add = (a, b) => a + b;`

✅ 3️⃣ Arrow Function Examples

💡 Single Parameter:

```
const greet = name => "Hello, " + name;


console.log(greet("Alice"));  // Output: Hello, Alice
```

✅ If there's only **one parameter**, you can skip the `( )`.

💡 Multiple Parameters:

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


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

✅ Must use `( )` when there are **multiple parameters**.

💡 No Parameters:

```
const welcome = () => "Welcome to JavaScript!";


console.log(welcome());  // Output: Welcome to JavaScript!
```

✅ Use `( )` for zero parameters.

💡 Multi-line Block:

If your function does more than one statement, you must use `{}` and `return`:

```
const calculateArea = (length, width) => {
```

```
    let area = length * width;
    return area;
};

console.log(calculateArea(5, 4));  // Output: 20
```

✅ 4️⃣ Differences: Arrow vs Traditional

Feature	`function` keyword	Arrow Function (`=>`)
Syntax	Verbose	Short & Clean
`this` Binding	Dynamic (depends on caller)	Lexical (inherits from parent scope)
Usage	Everywhere	Best for callbacks & simple functions

⚠️ **Important Note:**
Arrow functions do **not** create their own `this`.
They use the `this` value from the surrounding (parent) scope.

💡 Example of `this` Difference:

```
function Person() {
    this.name = "Alex";
    setTimeout(function() {
        console.log("Traditional:", this.name);  // undefined
    }, 1000);
```

```
    setTimeout(() => {
        console.log("Arrow:", this.name);  // Alex
    }, 1000);
}

new Person();
```

✅ Arrow functions automatically use the `this` from the `Person` object.

✅ 5 When to Use Arrow Functions?

✅ Best for:

- Simple one-line logic.
- `map()`, `filter()`, `forEach()`.
- When you want to keep `this` from the parent scope.

❌ Avoid for:

- Functions that require `this` binding, like constructors or object methods.

✅ 6 Summary:

Situation	Traditional Function	Arrow Function
Short expressions	Possible, but verbose	✅ Perfect
`this` handling	Dynamic	Fixed (lexical)
Syntax	Long	Short & modern

💡 **Practice Task for You:**

1. Convert this traditional function into an arrow function:

```
function double(num) {
   return num * 2;
}
```

✅ **Arrow Version:**

```
const double = num => num * 2;
```

2. Write an arrow function that calculates the square of a number.

```
const square = num => num * num;

console.log(square(5));  // Output: 25
```

Excellent — you're covering all the core JavaScript foundations! This is a very useful concept, especially when you want your code to execute immediately without being called later.

💡 JavaScript Lecture 10: Self-Invoking Functions (IIFE)

✅ 1️⃣ What is a Self-Invoking Function?

A **Self-Invoking Function** (also known as an **Immediately Invoked Function Expression** or **IIFE**) is a function that runs **automatically** — immediately when the code is loaded.

👉 **You don't need to call it!**
It calls itself as soon as the browser (or JavaScript engine) reaches it.

✅ 2 Syntax

```
(function(){
  // Code inside
})();
```

✅ Note:

- The function is **wrapped in ( )** — this tells JavaScript:
  "This is an expression, not a declaration."
- The final **( )** **calls** the function immediately.

✅ 3 Example

```
(function() {
  console.log("I am a self-invoking function!");
})();
```

💡 **Output:**

I am a self-invoking function!

✅ 4 With Parameters

You can pass arguments too:

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

💡 **Output:**

Hello, Alice!

✅ 5 Why Use Self-Invoking Functions?

✅ **Main uses:**

Purpose	Description
Data Privacy / Encapsulation	Variables inside IIFE can't be accessed from outside.
Avoid Global Scope Pollution	Keeps your variables and logic private.
Auto-Execution	Runs the code immediately.

💡 Example: Hiding Variables

```
(function() {
    let secret = "This is private!";
    console.log(secret);  // Output: This is private!
})();

console.log(typeof secret);  // Output: undefined (cannot access `secret` outside)
```

✅ 6 IIFE Variations

a) Anonymous IIFE:

```
(function() {
    console.log("Anonymous self-invoking function");
})();
```

b) Named IIFE:

```
(function showMessage() {
    console.log("Named IIFE function");
})();
```

✅ 7 Modern Use in JavaScript

In modern JavaScript, IIFEs are often used in:

- Modules
- Initialization logic
- Data privacy
- Avoiding global variables

✅ 8 Summary Table

Feature	Explanation
Self-Invokes	Runs automatically when defined.
Syntax	`(function() { ... })();`
Scope Isolation	Keeps variables private inside.
Use Case	Prevent global variable pollution.

💡 **Practice Task for You:**

✅ Write a self-invoking function that multiplies two numbers:

```
(function(a, b) {
    console.log("Result:", a * b);
})(4, 5);
```

💡 Output:

Result: 20

Great — this is a super important concept to understand in JavaScript (and in programming in general)! Let's break it down step by step in simple language.

💡 JavaScript Lecture 11: What Was the Need for Objects?

✅ 1️⃣ The Problem Before Objects

Imagine you have a person's data:

```
let name = "John";
let age = 30;
let profession = "Developer";
let city = "New York";
```

If you want to store **10 people's data** like this, you would need 40 separate variables!

💡 Example:

```
let name1 = "John";
```

```
let age1 = 30;

let name2 = "Alice";
let age2 = 25;

// And so on...
```

❌ **Problem:**

- Code becomes **unmanageable**.
- Variables are **scattered**.
- Relationships between data are **lost**.

✅ 2️⃣ The Solution: Objects

JavaScript introduced **Objects** to solve this!

An **object** groups related data into a single structure.

💡 Example: Person Object

```
let person = {
    name: "John",
    age: 30,
    profession: "Developer",
    city: "New York"
};
```

✅ Now all related data is:

- Neatly stored in one place.
- Easy to manage.
- Easy to access.

✅ 3️⃣ Benefits of Using Objects

Problem Without Objects	Solved By Objects
Too many variables	One object can hold multiple values.
Hard to organize related data	Group data using `key: value` pairs.
Difficult to extend	Easily add or remove properties.
Relationship between data unclear	Object keeps related data together.

✅ 4️⃣ How Objects Work

Objects store data in **key-value pairs**:

```
let car = {
    brand: "Toyota",
    model: "Corolla",
    year: 2020
};
```

You can access the data using:

```
console.log(car.brand);  // Output: Toyota
```

Or:

```
console.log(car["model"]);  // Output: Corolla
```

✅ 5 Real-Life Analogy

Imagine your **phone contacts**:
A contact is not just a name — it's:

Property	Value
Name	Alice
Phone Number	9876543210
Email	alice@mail.com

That's an **object** in the real world.

✅ 6 Objects Make Code Clean & Reusable

You can store **multiple people** in an array of objects:

```
let people = [
  { name: "John", age: 30 },
  { name: "Alice", age: 25 },
```

```
    { name: "Bob", age: 35 }
];
```

And loop through them:

```
for(let person of people) {
    console.log(person.name + " is " + person.age + " years old.");
}
```

✅ 7️⃣ Summary

Before Objects	After Objects
Messy and many variables	One single structured unit.
Hard to manage & understand	Easy, clean, and logical.
Not reusable	Highly reusable & scalable.

🎯 In Short:

- Objects allow you to group **related data** in a single unit.
- Make your code **organized, clean, and professional**.
- Essential when handling **real-world data** like users, cars, books, movies, etc.

💡 **Practice Task for You:**

Create an object for a `Book`:

```
let book = {
    title: "JavaScript Basics",
    author: "John Doe",
    pages: 150,
    price: 299
};


console.log("Book:", book.title, "by", book.author);
```

Excellent — you're learning the heart of JavaScript!
Let's go step by step. This is a super useful topic for working with real-world data!

💡 JavaScript Lecture 12: Creating Objects & Manipulating Values

---

✅ 1️⃣ Creating an Object

In JavaScript, you create an object using **curly braces** `{}` with `key: value` pairs.

💡 Example:

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

In this example:

- `name`, `age`, `profession` → **Keys / Properties**
- `"Alice"`, `25`, `"Developer"` → **Values**

✅ 2️⃣ Accessing Values from an Object

You can access data in two ways:

👉 **Dot Notation**:

```
console.log(person.name);  // Output: Alice
```

👉 **Bracket Notation**:

```
console.log(person["age"]);  // Output: 25
```

✅ 3️⃣ Changing (Manipulating) Values

You can **update** a property by directly assigning a new value:

```
person.age = 26;

console.log(person.age);  // Output: 26
```

or:

```
person["profession"] = "Software Engineer";

console.log(person.profession);  // Output: Software Engineer
```

✅ 4️⃣ Adding New Properties

You can add a new key-value pair anytime:

person.city = "New York";

console.log(person.city); // Output: New York

✅ 5️⃣ Deleting Properties

You can remove properties using the `delete` keyword:

delete person.age;

console.log(person.age); // Output: undefined

✅ 6️⃣ Looping Through an Object

You can loop through all keys using `for...in`:

for (let key in person) {

    console.log(key + ": " + person[key]);

}

💡 **Output:**

name: Alice

profession: Software Engineer

city: New York

✅ 7️⃣ Why is Manipulation Useful?

Situation	Solution
Need to update data	Reassign the property.

Need to add new data	Create new property.
Need to remove data	`delete` keyword.

💡 Real-World Example

Imagine you have a student profile:

```
let student = {
    name: "Rahul",
    class: 6,
    grade: "B"
};

// Update grade:
student.grade = "A";

// Add a new property:
student.section = "Blue";

// Delete a property:
delete student.class;

console.log(student);
```

✅ 8️⃣ Summary Table

Action	Syntax Example
Create Object	`let obj = { key: value };`
Access Value	`obj.key` or `obj["key"]`
Update Value	`obj.key = newValue;`
Add New Property	`obj.newKey = value;`
Delete Property	`delete obj.key;`

✅💡 **Practice Task for You:**

1️⃣ Create an object called `car` with keys: `brand`, `model`, `year`.
2️⃣ Update the `year`.
3️⃣ Add a new key: `color`.
4️⃣ Delete the `model` key.
5️⃣ Print the final object.

Perfect — now you're entering the world where JavaScript starts behaving more like real life! Let's explore how **Object Methods** work and why they are so powerful.

💡 JavaScript Lecture 13: Object Methods

✅ 1️⃣ What is an Object Method?

A method is simply a **function stored inside an object**.

➡️ So, **when a function is attached to an object, it becomes a method**.

🔶 Example:

```
let person = {
    name: "Alice",
    greet: function() {
        console.log("Hello, my name is " + this.name);
    }
};
```

Here, `greet` is a **method** of the `person` object.

✅ 2️⃣ Calling a Method

You call it just like a normal function — but through the object:

```
person.greet();  // Output: Hello, my name is Alice
```

✅ 3️⃣ The `this` Keyword

Inside an object method, `this` refers to **the object itself**.

```
let car = {
    brand: "Toyota",
    showBrand: function() {
```

```
        console.log("Brand is: " + this.brand);
    }
};

car.showBrand();  // Output: Brand is: Toyota
```

🧠 `this.brand` = `car.brand` in the example above.

✅ 4️⃣ Shorter Syntax (ES6)

You can write methods in a shorter way:

```
let student = {
    name: "Rahul",
    sayHi() {
        console.log("Hi, I am " + this.name);
    }
};

student.sayHi();  // Output: Hi, I am Rahul
```

✅ 5️⃣ Object with Multiple Methods

```
let calculator = {
    add(a, b) {
        return a + b;
    },
```

```
    subtract(a, b) {
        return a - b;
    }
};

console.log(calculator.add(5, 3));      // Output: 8
console.log(calculator.subtract(5, 3)); // Output: 2
```

> The object `calculator` acts like a **toolbox** full of useful functions!

✅ 6 Use Case in Real World

Objects with methods are perfect for:

- Models (like a `user`, `product`, or `order`)
- Logic encapsulation (a calculator, bank account, game character)
- Code organization

💡 Example: Bank Account

```
let account = {
    name: "John",
    balance: 1000,
    deposit(amount) {
        this.balance += amount;
        console.log("Deposited " + amount);
    },
    withdraw(amount) {
```

```
        if (amount <= this.balance) {
            this.balance -= amount;
            console.log("Withdrew " + amount);
        } else {
            console.log("Insufficient balance");
        }
    },
    checkBalance() {
        console.log("Balance: " + this.balance);
    }
};

account.deposit(500);
account.withdraw(300);
account.checkBalance();
```

✅ 7️⃣ Summary Table

Concept	Example
Create Method	`methodName: function() {}`
Call Method	`object.methodName()`

Use `this`	Refers to the current object
ES6 Syntax	`methodName() {}`

🎯 Practice Task for You:

Create an object `student` with:

- `name`
- `marks`
- A method `greet()` → prints name.
- A method `checkPass()` → prints "Pass" if marks ≥ 40 else "Fail".

Try it out!

Excellent! You're now entering one of the most important parts of web development —

JavaScript + DOM is what gives life to web pages!
Let's go deep into it, nice and clear. 👇

💡 JavaScript Lecture 14: Understanding the DOM — Visualization & Working

---

✅ 1️⃣ What is the DOM?

DOM stands for:
Document **O**bject **M**odel

When a web page loads, the browser converts the HTML into a structured **tree-like model**.
That model is called the **DOM**.

➡️ You can think of the DOM as the **JavaScript-friendly version of your HTML**.

✅ 2️⃣ Visualize the DOM (Mental Image)

Let's say your HTML looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

🧠 The browser turns this into a **Tree Structure**:

```
Document
 └── html
     ├── head
     │    └── title
     └── body
          ├── h1
          └── p
```

Every part of your page:

- tags (`<p>`, `<h1>`)
- text (`"Hello World"`)
- attributes (`class`, `id`)

becomes an **object** in the DOM.

✅ 3 Why is the DOM Important?

Because JavaScript can:

✔ Read
✔ Change
✔ Delete
✔ Create

any part of the webpage by accessing the DOM.

✅ 4 DOM = Live Connection

When you load a web page:

- The browser reads the **HTML**
- Converts it into the **DOM Tree**
- JavaScript interacts with that tree.

Any change you make with JavaScript → immediately updates the visible web page!

✅ 5 DOM in Action (Simple Example)

💡 HTML:

<h1 id="heading">Welcome!</h1>

💡 JavaScript:

```
document.getElementById("heading").innerText = "Hello, JavaScript!";
```

✅ **What happens?**

- JavaScript finds the element `<h1 id="heading">`.
- Changes its text.
- The browser shows: `Hello, JavaScript!`

✅ 6️⃣ DOM — Real World Analogy

Think of your web page like a **House**:

Real World	DOM Equivalent
A window	`<div>` element
A light switch	JavaScript event
Turning ON light	Changing element style

Your JavaScript is the person **controlling the house** (using `document` as the tool) to do things.

✅ 7️⃣ DOM Working (Step-by-Step)

1️⃣ **Browser loads HTML**
2️⃣ Builds **DOM Tree**
3️⃣ JavaScript uses `document` to:

- select elements
- modify elements
- add new elements
- remove elements

Example:

let title = document.querySelector("h1");

title.style.color = "red";

➡️ Changes the `<h1>` text color to red on your web page!

✅ 8️⃣ Recap Table

Concept	Meaning
DOM	JavaScript version of the webpage
Tree Structure	Elements organized like a family tree
Manipulation	JavaScript can edit page in real-time
`document` object	Main entry point for the DOM

✅ **In Simple Words:**
HTML is the page's "blueprint",
DOM is the **live model** of the page,
JavaScript is the **worker** that changes it.

🎯 Practice Task for You:

👉 Open your browser (press `F12` or `Ctrl + Shift + I`)
👉 Go to `Console` tab.

Paste this:

document.body.style.backgroundColor = "lightblue";

Great! You're moving smoothly — this is the part where JavaScript starts to "touch" the webpage directly!
Let's go deep into:

💡 JavaScript Lecture 15: Methods of DOM — Part 1

✅ 1️⃣ What are DOM Methods?

👉 DOM Methods are **pre-built JavaScript functions** that allow you to:

- Find
- Create
- Update
- Delete

elements on a web page.

✅ 2️⃣ Common DOM Methods

Let's focus today on the **most important ones** for selecting elements!

Method	Purpose
`getElementById()`	Selects element by `id`
`getElementsByClassName()`	Selects all elements with a class
`getElementsByTagName()`	Selects all elements of a tag
`querySelector()`	Selects first matching element
`querySelectorAll()`	Selects all matching elements

✅ 3️⃣ `getElementById()`

Selects a single element by its **ID**.

💡 Example:

<h1 id="mainTitle">Welcome!</h1>

let title = document.getElementById("mainTitle");

title.innerText = "Hello, JavaScript!";

✅ 4️⃣ `getElementsByClassName()`

Selects **all elements** that share a **class**.
Returns a **collection** (like an array).

💡 Example:

```
<p class="note">Paragraph 1</p>

<p class="note">Paragraph 2</p>


let notes = document.getElementsByClassName("note");

notes[0].style.color = "blue";
```

✅ 5 `getElementsByTagName()`

Selects **all elements** with a specific tag.
Returns a **collection**.

💡 Example:

```
<h1>Hello</h1>

<h1>World</h1>


let headings = document.getElementsByTagName("h1");

headings[1].innerText = "JavaScript!";
```

✅ 6 `querySelector()`

Selects the **first matching element** using **CSS selector rules**.

💡 Example:

```
<p class="intro">First Paragraph</p>
```

Second Paragraph

let firstPara = document.querySelector(".intro");

firstPara.style.fontSize = "20px";

✅ 7 querySelectorAll()

Selects **all matching elements** using **CSS selector rules**.

let allParas = document.querySelectorAll(".intro");

allParas.forEach(function(para) {

  para.style.color = "green";

});

✅ 💡 8 Summary Table

Method	Return Type	Usage Example
`getElementById`	Single Element	`document.getElementById("id")`
`getElementsByClassName`	HTMLCollection (list)	`document.getElementsByClassName("class")`
`getElementsByTagName`	HTMLCollection (list)	`document.getElementsByTagName("tag")`

`querySelector`	Single Element	`document.querySelector(".class")`
`querySelectorAll`	NodeList (list)	`document.querySelectorAll(".class")`

🎯 Practice Task for You:

Create an HTML file with:

<h1 id="myTitle">Original</h1>

Text 1

Text 2

and write JavaScript to:

1️⃣ Change the `<h1>` text to "Updated by JavaScript".
2️⃣ Change the color of all `<p class="myText">` to blue.

Excellent — welcome to **Part 2** of one of the most powerful tools in web development!
Let's go deeper into DOM Methods beyond just "selecting."

💡 JavaScript Lecture 16: Methods of DOM — Part 2

✅ 1️⃣ Recap: What you've learned so far

- `getElementById`

- `getElementsByClassName`
- `getElementsByTagName`
- `querySelector` / `querySelectorAll`

All of these help you **find elements**.

✅ 2️⃣ Now: Manipulating Elements with DOM Methods

Once you select an element, you can:

Action	Method / Property
Change text	`.innerText` / `.textContent`
Change HTML	`.innerHTML`
Change CSS	`.style`
Create new elements	`document.createElement()`
Add elements to DOM	`appendChild()` / `append()`
Remove elements	`remove()`

✅ 3️⃣ Change Text / HTML Content

🧠 Change Text:

let heading = document.getElementById("myTitle");

heading.innerText = "New Heading!";

🧠 Change HTML:

let para = document.querySelector(".myText");

para.innerHTML = "Bold Text!";

✅ 4️⃣ Change CSS (inline styles)

let heading = document.querySelector("h1");

heading.style.color = "red";

heading.style.backgroundColor = "yellow";

heading.style.padding = "10px";

✅ 5️⃣ Create New Elements

let newElement = document.createElement("p");

newElement.innerText = "This is a new paragraph!";

✅ 6️⃣ Add New Elements to the Page

First select a **parent element** (where you want to insert):

```
let container = document.body;

container.appendChild(newElement);
```

Or:

```
container.append(newElement);  // Modern, can append text and nodes
```

✅ 7️⃣ Remove Elements

If you want to delete an element:

```
let unwanted = document.querySelector("p");

unwanted.remove();
```

✅ 💡 8️⃣ Example: Full Flow

```
<div id="box">
  <p>Old text</p>
</div>


let div = document.getElementById("box");


// Change existing text

div.querySelector("p").innerText = "New text from JavaScript!";


// Create a new element

let newPara = document.createElement("p");
```

```
newPara.innerText = "This is a new paragraph.";

// Add the new element inside the div
div.appendChild(newPara);

// Remove the old paragraph
let oldPara = div.querySelector("p");
oldPara.remove();
```

✅ 9️⃣ Recap Table

Action	Code Example
Change text	`element.innerText = "..."`
Change HTML	`element.innerHTML = "<b>...</b>"`
Change CSS	`element.style.property = "value"`
Create element	`document.createElement("tag")`
Add element to page	`parent.appendChild(newElement)`

Remove element	`element.remove()`

🎯 Practice Task for You:

1️⃣ Create an HTML file with a `<div>` and `<p>` inside.
2️⃣ Using JavaScript:
✔️ Change the `<p>` text.
✔️ Create and append a new `<h2>` with your name.
✔️ Change the color of `<h2>` to green.
✔️ Remove the `<p>`.

Awesome — you're doing great! Let's jump into the advanced side of DOM methods now — this is where your webpage will truly become **dynamic**.

💡 JavaScript Lecture 17: Methods of DOM — Part 3

✅ 1️⃣ More Ways to Add and Remove Elements

In the last lecture you learned about:

- `appendChild()`
- `append()`
- `remove()`

But JavaScript gives you even more control over where and how you place elements!

🧠 2️⃣ `prepend()` — Add at the Beginning

`appendChild()` and `append()` add at the end, but `prepend()` adds the element as the **first child**.

💡 Example:

  Old Paragraph

let box = document.getElementById("box");

let newPara = document.createElement("p");

newPara.innerText = "I am the first paragraph!";

box.prepend(newPara);

🧠 3️⃣ `insertBefore()` — Insert at Specific Position

You can insert a new element **before a specific existing one**.

let parent = document.getElementById("box");

let newElement = document.createElement("p");

newElement.innerText = "Inserted Paragraph";

let reference = parent.querySelector("p");

parent.insertBefore(newElement, reference);

🧠 4️⃣ `replaceChild()` — Replace Existing Element

Replace one child element with another:

```
let parent = document.getElementById("box");
let newElement = document.createElement("p");
newElement.innerText = "I am the new one!";

let oldElement = parent.querySelector("p");
parent.replaceChild(newElement, oldElement);
```

🧠 5️⃣ `cloneNode()` — Clone Elements

Duplicate an existing node:

```
let original = document.querySelector("p");
let copy = original.cloneNode(true);  // true = deep clone (copy children too)
document.body.appendChild(copy);
```

✅ 6️⃣ Changing Attributes with DOM

You can also change attributes like `src`, `href`, `alt`, `id`, `class` etc.

```
let img = document.querySelector("img");
img.setAttribute("src", "new-image.jpg");
img.setAttribute("alt", "New Description");

Or remove an attribute:
img.removeAttribute("alt");

Or check if an attribute exists:
```

```
if (img.hasAttribute("src")) {
    console.log("Image has a source!");
}
```

✅ 7 Class List Methods

Another cool way to control `class` values.

```
let heading = document.querySelector("h1");

heading.classList.add("highlight");    // Add class
heading.classList.remove("highlight"); // Remove class
heading.classList.toggle("highlight"); // Add/remove based on state
```

This is often used with CSS to control visibility, color, animation.

✅ 8 Recap Table

Method	Purpose
`appendChild()`	Adds at the end
`prepend()`	Adds at the beginning
`insertBefore()`	Inserts before a specific element

`replaceChild()`	Replaces one child with another
`cloneNode(true/false)`	Clones an element (deep or shallow)
`setAttribute()`	Sets/Updates an attribute
`removeAttribute()`	Removes an attribute
`classList.add()`	Adds a class
`classList.remove()`	Removes a class
`classList.toggle()`	Toggles a class on/off

🎯 Practice Task for You:

Create this mini challenge:

1️⃣ Create a `<div>` with one `<p>` inside it.
2️⃣ Using JavaScript:

- Insert a new paragraph at the top using `prepend()`.
- Replace the old paragraph with a new one using `replaceChild()`.
- Clone the new paragraph and add it at the bottom.
- Add a CSS class to the cloned one (`.highlight`).

Excellent! Now you're moving toward making **interactive web pages** — the modern way.

Let's build this lecture cleanly!

💡 JavaScript Lecture 18: Color Changer in DOM (Modern Way)

✅ 1️⃣ Old-School Way — Why Avoid `onclick`?

In older tutorials, you often see this:

<button onclick="changeColor()">Click Me</button>

👉 But this mixes **HTML** and **JavaScript**, which is:

- ❌ Hard to maintain
- ❌ Hard to debug
- ❌ Breaks separation of concerns (structure & logic mixed)

✅ 2️⃣ Modern Way — `addEventListener()`

The clean way to handle interaction is by using:

element.addEventListener("event", function);

This keeps your **JavaScript separate** from your HTML.

🧠 3️⃣ Example: Build a Color Changer

💻 HTML:

<button id="colorBtn">Change Color</button>

<div id="colorBox"
style="width:200px;height:200px;background-color:lightgray;margin-top:20px;">

💻 JavaScript:

let button = document.getElementById("colorBtn");

let box = document.getElementById("colorBox");

button.addEventListener("click", function() {

  let randomColor = "#" + Math.floor(Math.random()*16777215).toString(16);

  box.style.backgroundColor = randomColor;

});

✅ **Explanation:**

- `getElementById` grabs the button and box.
- `addEventListener("click",...)` listens for the click.
- `Math.random()` generates a random color.
- The background color of the box changes dynamically when you click.

💡 4️⃣ Bonus: Clean Function Version

If you want even cleaner code:

function generateRandomColor() {

```
    return "#" + Math.floor(Math.random() * 16777215).toString(16);
}

function changeBoxColor() {
    box.style.backgroundColor = generateRandomColor();
}

button.addEventListener("click", changeBoxColor);
```

✅ **This is modern, reusable, and readable!**

💡 5️⃣ Summary:

Method	Purpose
`onclick` (old)	Direct event binding in HTML (avoid)
`addEventListener()`	Modern event handling — clean and flexible

🎯 Practice Task for You:

1️⃣ Create an HTML page with a button and a square `<div>`.
2️⃣ Write JavaScript that:

- Changes the square's color to a **random color** every time the button is clicked.

- Uses `addEventListener()`, not `onclick`.

Excellent — this is one of the most important and interesting parts of JavaScript!
Let's break this down step by step, nice and clear.

💡 JavaScript Lecture 19: Higher Order Functions (HOF), Callback, Returning Functions, setInterval & setTimeout

---

✅ 1️⃣ What is a Higher Order Function (HOF)?

A **Higher Order Function** is:

> 💡 "A function that either takes another function as an argument, or returns another function."

🧠 Why use it?

- It makes your code reusable.
- It makes your code clean and dynamic.
- It's the heart of JavaScript's flexibility (especially in frameworks like React).

💻 Example: Passing Function as Argument

```
function greet(name) {
   return `Hello, ${name}!`;
}

function processUserInput(callback) {
```

```
    let name = "John";
    console.log(callback(name));
}

processUserInput(greet);
```

✅ **Explanation:**

- `greet` is passed as a parameter to `processUserInput`.
- `processUserInput` executes the `greet` function.
- So here, `processUserInput` is a **Higher Order Function**.

✅ 2️⃣ What is a Callback Function?

> 💡 "A callback is a function passed into another function as an argument, which will be executed later."

Every callback is a function, but not every function is a callback (only if it's passed to another function to run).

💻 Simple Callback Example:

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

function execute(callback) {
    callback();  // calling the function passed as argument
}
```

```
execute(sayHello);
```

✅ **Explanation:**

- `sayHello` is a callback.
- `execute` is a higher-order function.

✅ 3️⃣ Returning a Function from Another Function

In JavaScript, functions are **first-class citizens** — meaning you can return them!

💻 Example:

```
function outerFunction() {
    return function innerFunction() {
        console.log("I am the inner function!");
    }
}

let result = outerFunction(); // returns the inner function
result(); // calling the returned function
```

✅ **Explanation:**

- `outerFunction()` returns another function.
- `result` now holds the inner function.
- `result()` runs the returned function.

✅ 4️⃣ Real-life use: `setTimeout` and `setInterval`

⏰ `setTimeout` — Run something once after a delay

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

✅ **Explanation:**

- The function will execute **once** after 2000ms (2 seconds).
- `setTimeout` takes a callback function and a time.

⏰ `setInterval` — Run something repeatedly at intervals

```
setInterval(function() {
    console.log("This runs every 1 second!");
}, 1000);
```

✅ **Explanation:**

- The function runs every 1000ms (1 second) — again and again.
- Until you stop it manually with `clearInterval()`.

💡 Using Named Functions with setTimeout / setInterval

```
function showMessage() {
    console.log("Hello, I am a delayed message!");
```

```
}

setTimeout(showMessage, 3000);  // runs after 3 seconds
setInterval(showMessage, 5000); // runs every 5 seconds
```

✅ 💡 5️⃣ Real-World Flow: HOF + Callback + setTimeout

```
function delayedGreeting(name, delay) {
    setTimeout(function() {
        console.log(`Hi, ${name}!`);
    }, delay);
}

delayedGreeting("Alice", 3000);  // prints after 3 seconds
```

✅ **Explanation:**

- `setTimeout` expects a function as a callback.
- `delayedGreeting` is a higher-order function (because it passes a function to `setTimeout`).

🧠 6️⃣ Summary Table

Concept	Meaning	Example

Higher Order Function (HOF)	Accepts or returns another function	`processUserInput(callback)`
Callback Function	Passed as argument and invoked later	`sayHello` in `execute(sayHello)`
Returning Functions	A function returns another function	`let f = outerFunction(); f();`
`setTimeout`	Executes once after delay	`setTimeout(fn, 2000);`
`setInterval`	Executes repeatedly at fixed time intervals	`setInterval(fn, 1000);`

🎯 Practice Task for You:

1️⃣ Write a function `countdown()` that prints numbers 5 to 1 with a 1-second gap using `setTimeout`.
2️⃣ Write a function that uses `setInterval` to print "Working..." every 2 seconds.
3️⃣ Pass a callback to a function that logs `"Task Completed!"` after the main task runs.

Excellent — this is one of the most powerful and important lectures for writing clean, short, and professional JavaScript code!

Let’s dive deep, nice and easy:

💡 JavaScript Lecture 20: forEach, map, filter, reduce, every, find, sort

---

✅ 1️⃣ `forEach()` — Loop through an Array

💡 Executes a function once for each array element.

💻 Example:

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


numbers.forEach(function(number) {
    console.log(number);
});
```

✅ **Explanation:**

- It runs your function for each item.
- Useful for printing, applying side effects.
- It does NOT return a new array.

✅ 2️⃣ `map()` — Create a New Transformed Array

💡 Returns a new array after applying a function to each item.

💻 Example:

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


let doubled = numbers.map(function(num) {
    return num * 2;
});


console.log(doubled);  // [2, 4, 6, 8]
```

✅ **Explanation:**

- Creates a new array.
- Original `numbers` array is unchanged.
- `map` is great for transformation.

✅ 3️⃣ `filter()` — Return Items that Pass a Test

💡 Returns a new array with items that pass the condition.

💻 Example:

```
let ages = [12, 17, 18, 24, 16];


let adults = ages.filter(function(age) {
   return age >= 18;
});


console.log(adults);  // [18, 24]
```

✅ **Explanation:**

- Only elements that pass `return true` are kept.
- Often used for searching or sorting subsets.

✅ 4️⃣ `reduce()` — Condense Array to a Single Value

💡 Reduces the whole array to a single output (sum, total, product, etc).

💻 Example:

```
let numbers = [5, 10, 15];


let sum = numbers.reduce(function(accumulator, currentValue) {
    return accumulator + currentValue;
}, 0);


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

✅ Explanation:

- `accumulator` is the running result.
- `currentValue` is the item being processed.
- `reduce()` is great for totals, averages, or combining data.

✅ 5️⃣ `every()` — Check if All Elements Meet Condition

> 💡 Returns `true` if **all** elements pass the test.

💻 Example:

```
let marks = [60, 70, 80, 90];


let allPassed = marks.every(function(mark) {
    return mark >= 50;
});
```

```
console.log(allPassed);  // true
```

✅ **Explanation:**

- As soon as one item fails the test, it returns `false`.

✅ 6 `find()` — Return First Matching Element

💡 Returns the first element that passes the test.

💻 Example:

```
let numbers = [3, 8, 12, 5];

let found = numbers.find(function(num) {
   return num > 10;
});

console.log(found);  // 12
```

✅ **Explanation:**

- Stops at the first matching element.
- If none found, returns `undefined`.

✅ 7 `sort()` — Sort Elements

💡 Sorts elements, but ⚠️ be careful — it sorts as strings by default.

💻 Example: Default (String Sort)

```
let numbers = [100, 20, 3];

numbers.sort();

console.log(numbers);  // [100, 20, 3] (unexpected!)
```

💻 Example: Numeric Sort

```
numbers.sort(function(a, b) {
   return a - b;
});

console.log(numbers);  // [3, 20, 100]
```

✅ **Explanation:**

- `a - b` sorts ascending.
- `b - a` sorts descending.

🧠💡 Summary Table

Method	Purpose	Return Type
`forEach()`	Loops through each item	undefined (no return)
`map()`	Transforms array items	New array
`filter()`	Filters elements based on condition	New array

`reduce()`	Reduces to a single value	Single value
`every()`	Checks if all elements match condition	Boolean (`true/false`)
`find()`	Finds first matching element	Element / `undefined`
`sort()`	Sorts elements (strings/numbers)	Sorted same array

🎯 Practice Task for You:

1️⃣ Create an array `[2, 5, 8, 11, 14]` and:

- Use `map()` to double the numbers.
- Use `filter()` to select numbers greater than 10.
- Use `reduce()` to sum all numbers.
- Use `every()` to check if all numbers are positive.
- Use `find()` to locate the first number larger than 10.
- Use `sort()` to sort the numbers in ascending order.

Awesome — this is a very powerful topic! Understanding **Regex (Regular Expressions)** will make you a smarter developer, especially when you deal with:

✅ form validation,
✅ searching text,
✅ replacing patterns,
✅ filtering input,
✅ web scraping, etc.

Let's break this down clearly!

💡 JavaScript Lecture 21: Regex — Meaning, Characters & Importance

---

✅ 1️⃣ What is Regex?

> 💡 **Regex (Regular Expression)** is a powerful tool to search, match, and manipulate text using patterns.

Think of it as a super-smart search tool.
It doesn't look for exact words, it looks for **patterns**.

💻 Simple Example:

```
let text = "I love JavaScript!";

let pattern = /JavaScript/;


console.log(pattern.test(text));  // true
```

✅ `test()` checks: "Does this string match the pattern?"
→ Here it's `true` because `JavaScript` is inside the string.

✅ 2️⃣ Why Regex is Important?

- 🔎 **Searching** in text.
- ✅ **Validating** email, passwords, names, etc.
- 💡 **Extracting data** from files, APIs.
- 🧹 **Replacing patterns** in text.

✅ 3 Syntax of Regex

Symbol	Meaning
`/`	Start and end of a regex pattern
`[ ]`	Character set (match any character inside)
`.`	Matches any one character except newline
`^`	Matches the start of a string
`$`	Matches the end of a string
`*`	Zero or more of the previous character
`+`	One or more of the previous character
`{}`	Exact number or range of repetitions
`( )`	Grouping part of the expression
`` ` ``	`` ` ``
`\`	Escape special characters

✅ 4️⃣ Character Examples

🔥 [ ] — Character Set

```
let pattern = /[aeiou]/;  // any vowel

console.log(pattern.test("Hello"));  // true
```

🔥 . — Any Character

```
let pattern = /h.t/;

console.log(pattern.test("hat"));  // true

console.log(pattern.test("hot"));  // true
```

→ Matches `h` + any character + `t`.

🔥 ^ and $ — Start and End

```
let pattern = /^Hello/;

console.log(pattern.test("Hello World"));  // true

let pattern2 = /World$/;

console.log(pattern2.test("Hello World"));  // true
```

🔥 * and + — Repeats

```
let pattern = /go*/;
```

```
console.log(pattern.test("gooo"));  // true
```

`o*` → zero or more `o`s.

🔥 `{}` — Exact Counts

```
let pattern = /\d{3}/;  // exactly 3 digits

console.log(pattern.test("abc123"));  // true
```

🔥 `|` — OR

```
let pattern = /apple|banana/;

console.log(pattern.test("I like banana"));  // true
```

✅ 5️⃣ Real-World Example: Email Validation

```
let email = "user@example.com";

let pattern = /^[a-zA-Z0-9._]+@[a-zA-Z]+\.[a-zA-Z]{2,}$/;

console.log(pattern.test(email));  // true
```

✅ **Explanation:**

- `^[a-zA-Z0-9._]+` → start with letters, numbers, dot or underscore.
- `@` → must contain `@`.
- `[a-zA-Z]+` → domain name.

- `\.` → a literal dot.
- `[a-zA-Z]{2,}$` → domain ending must be at least 2 letters.

💡 6️⃣ Useful Regex Functions in JavaScript

Function	Purpose
`test()`	Checks if pattern exists (`true/false`)
`exec()`	Returns detailed match result or `null`
`match()`	Returns an array of all matches
`replace()`	Replaces matched content
`split()`	Splits string based on regex

💻 Example:

```
let text = "My number is 12345.";

let pattern = /\d+/;  // one or more digits


console.log(pattern.test(text));  // true

console.log(text.match(pattern)); // ["12345"]
```

🎯 7️⃣ Practice Task for You:

1️⃣ Write a regex that checks if a string contains **only digits**.
2️⃣ Write a regex to match valid phone numbers like `+91-9876543210`.
3️⃣ Write a regex that finds all the words starting with `a` in a sentence.

Excellent choice! This is one of the most useful topics in modern JavaScript — makes your code shorter, cleaner, and easier to understand!
Let's go step by step:

💡 JavaScript Lecture 22: Destructuring, Spread & Rest

✅ 1️⃣ Destructuring

Destructuring lets you unpack values from arrays or properties from objects into separate variables — in a clean and readable way.

💻 Array Destructuring

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


const [a, b, c] = numbers;


console.log(a);  // 10

console.log(b);  // 20

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

✅ **Explanation:**

- `[a, b, c] = numbers` → assigns each array item to a variable.

💻 Object Destructuring

const person = { name: "John", age: 25 };

const { name, age } = person;

console.log(name); // John

console.log(age); // 25

✅ **Explanation:**

- `{ name, age } = person` → assigns object properties to variables.

💡 **Renaming While Destructuring**

const person = { name: "Alice", age: 30 };

const { name: userName, age: userAge } = person;

console.log(userName); // Alice

console.log(userAge);   // 30

💡 **Default Values**

const user = { name: "Bob" };

const { name, age = 18 } = user;

```
console.log(age);  // 18 (default value)
```

✅ 2️⃣ Spread Operator (`...`)

The **spread operator** allows you to copy, merge or expand values.

💻 Copying Arrays

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

const arr2 = [...arr1];


console.log(arr2);  // [1, 2, 3]
```

💻 Merging Arrays

```
const fruits = ["apple", "banana"];

const moreFruits = ["mango", "pineapple"];


const allFruits = [...fruits, ...moreFruits];

console.log(allFruits);  // ["apple", "banana", "mango", "pineapple"]
```

💻 Copying Objects

```
const person = { name: "Lily", age: 22 };

const clone = { ...person };
```

```
console.log(clone);  // { name: "Lily", age: 22 }
```

💻 **Merging Objects**

```
const a = { x: 1 };

const b = { y: 2 };


const merged = { ...a, ...b };

console.log(merged);  // { x: 1, y: 2 }
```

✅ 3️⃣ Rest Parameter (`...`)

The **Rest parameter** collects remaining items into an array.

💻 **Function with Rest**

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


console.log(sum(1, 2, 3, 4));  // 10
```

✅ **Explanation:**

- `...numbers` gathers all arguments into one array.
- You can pass any number of values.

💻 Destructuring with Rest

```
const [first, second, ...rest] = [1, 2, 3, 4, 5];


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

🧠 💡 Summary Table

Feature	Purpose
Destructuring	Extract values from arrays/objects
Spread (`...`)	Expand array/object into individual elements
Rest (`...`)	Collect multiple values into a single variable

🎯 💪 Practice Task for You:

1️⃣ Destructure this object:

```
const student = { id: 101, name: "Rahul", grade: "A" };
```

2️⃣ Use spread to merge:

```
const colors1 = ["red", "blue"];
const colors2 = ["green", "yellow"];
```

3️⃣ Create a function that takes unlimited numbers as arguments and returns their multiplication using rest `...`.

Perfect — this is a super important topic, especially when your code needs to handle unexpected errors without crashing!

Let's make it very clear and beginner-friendly:

💡 JavaScript Lecture 23: Try & Catch — Error Handling

✅ 1️⃣ Why do we need `try` & `catch`?

When JavaScript runs your code, if it hits an error — your program will usually stop (break). But in real-world apps, we don't want that! We want to:

- Detect the error.
- Handle the error without stopping the whole program.
- Show the user a helpful message instead of "Page crashed!"

👉 **That's where `try...catch` comes in!**

✅ 2️⃣ Basic Syntax

```
try {
  // Code that might throw an error
} catch (error) {
  // Code to handle the error
}
```

💻 Example:

```
try {
  let result = 10 / 0;
  console.log("Result is:", result);

  // simulate error
  let user = undefined;
  console.log(user.name);  // 💥 This will cause an error
}
catch (error) {
  console.log("Oops! Something went wrong.");
  console.log("Error Details:", error.message);
}
```

✅ Explanation:

- JS tries to run everything in `try {}`.
- If there's an error, it jumps to `catch {}`.
- Your program doesn't crash — the error is handled.

✅ 3️⃣ Real-World Example — Form Validation

```
function processUserInput(input) {
  try {
    if(input.trim() === '') {
```

```
    throw new Error("Input cannot be empty.");
  }
  console.log("Processing:", input);
 }
 catch(error) {
  console.log("Error:", error.message);
 }
}

processUserInput("");        // ➡️ Error: Input cannot be empty.
processUserInput("Rahul");   // ➡️ Processing: Rahul
```

✅ 4️⃣ `finally` Block

`finally` is a part of `try...catch` that runs no matter what — error or not.

```
try {
 console.log("Trying something...");
 let x = 5 / 0;
}
catch(error) {
 console.log("Error caught:", error.message);
}
finally {
 console.log("I will always run, error or no error.");
}
```

✅ 5️⃣ Throwing Your Own Errors

You can manually create errors using `throw`. This is useful for validation.

```
let age = -3;


try {
  if(age < 0) {
    throw new Error("Age cannot be negative!");
  }
  console.log("Age is valid.");
}
catch(error) {
  console.log("Custom Error:", error.message);
}
```

🎯 💡 Summary

Concept	Purpose
`try {}`	Runs risky code
`catch {}`	Handles the error (if it happens)
`finally {}`	Always runs, no matter what

`throw`	Lets you trigger your own errors manually

🧠💪 Practice Task for You:

1️⃣ Create a function `divide(a, b)` that:

- returns the result if `b` is not zero.
- throws an error: `"Cannot divide by zero!"` if `b` is `0`.
- handles it using `try...catch`.

2️⃣ Add a `finally` block to the same function that logs:
`"Calculation finished."`

Excellent — this is an important step in learning modern JavaScript!
Classes are the heart of "Object Oriented Programming" (OOP). Once you understand this, you'll be able to write powerful, clean, and scalable code.

Let's start!

💡 JavaScript Lecture 24: Defining Classes & Class Instantiation

---

✅ 1️⃣ What is a Class?

A **class** is like a blueprint or template for creating objects.

For example:
A **Car** class can define:

- brand
- model

- color

But the real cars (**objects**) will have different values:

Blueprint	Real Object
Car (class)	Honda City (object)
	Tesla Model 3 (object)

✅ 2️⃣ Defining a Class in JavaScript

👉 Basic Syntax:

```
class ClassName {
  constructor(parameters) {
    // property initialization
  }

  methodName() {
    // method logic
  }
}
```

💻 Example: Defining a Class

```
class Car {
  constructor(brand, model) {
```

```
    this.brand = brand;
    this.model = model;
  }

  startEngine() {
    console.log(`${this.brand} ${this.model} engine started.`);
  }
}
```

✅ **Explanation:**

- `constructor()` is a special method called when you create a new object.
- `this` refers to the current object.
- `startEngine()` is a method you can call on any object of this class.

✅ 3️⃣ Instantiating a Class (Creating Objects)

👉 **Instantiation** = Creating an object from a class.

```
let myCar = new Car("Toyota", "Corolla");
myCar.startEngine();  // Output: Toyota Corolla engine started.
```

✅ `new Car()` → Creates a new object based on the class blueprint.

💡 You can create as many objects as you want:

```
let car1 = new Car("Honda", "Civic");
let car2 = new Car("Tesla", "Model S");
```

```
car1.startEngine();  // Honda Civic engine started.
car2.startEngine();  // Tesla Model S engine started.
```

✅ 4️⃣ Why Classes?

- Organizes code into reusable "blueprints."
- Makes it easier to manage multiple similar objects.
- Perfect for "real-world modeling" in code.

💡 🎯 Summary Table

Concept	Purpose
`class`	Defines a blueprint for creating objects
`constructor()`	Initializes properties during creation
`new` keyword	Instantiates (creates) a new object
`this`	Refers to the current object

🧠 💪 Practice Task for You:

1️⃣ Define a `Person` class with:

- name, age (as properties).
- method `introduce()` that logs:
  `Hi, my name is [name] and I am [age] years old.`

2️⃣ Create 2 objects from the class and call their `introduce()` methods.

Great! You're moving like a real developer — this is the perfect follow-up after learning about classes.

Let's go deep into:

💡 JavaScript Lecture 25: Class Constructor & Default Values

✅ 1️⃣ What is a Constructor?

A `constructor()` is a special method inside a class.

👉 It runs **automatically** when you create an object using `new`.
Its job is to:

- Initialize object properties.
- Set default values (if needed).
- Prepare the object for use.

💻 **Basic Example:**

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
```

```
  }

  greet() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  }
}

const person1 = new Person("Rahul", 25);
person1.greet();  // Output: Hi, I'm Rahul and I'm 25 years old.
```

✅ **Explanation:**

- When you write `new Person("Rahul", 25)`, the `constructor()` runs.
- `this.name` and `this.age` get values.

✅ 2️⃣ Default Values in Constructor

If no values are passed, JavaScript can use **default values**.

💻 **Example:**

```
class Person {
  constructor(name = "Anonymous", age = 18) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
```

```
  }
}

const p1 = new Person();              // No arguments
const p2 = new Person("Sita", 22);    // Passing arguments

p1.greet();  // Hi, I'm Anonymous and I'm 18 years old.
p2.greet();  // Hi, I'm Sita and I'm 22 years old.
```

✅ **Explanation:**

- If you pass values, `constructor` uses them.
- If you don't pass values, it uses the defaults (`"Anonymous"` and `18`).

💡 You can also mix defaults:

```
class Product {
  constructor(name = "No Name", price = 0) {
    this.name = name;
    this.price = price;
  }
}

const item = new Product();
console.log(item);  // { name: "No Name", price: 0 }
```

💡🧠 Why Use Default Values?

- Avoids `undefined` errors.
- Makes your code safer and easier to debug.
- Useful when sometimes you don't have all the info at the time of object creation.

🎯💪 Summary Table

Concept	Purpose
`constructor()`	Initializes properties of an object
Default Parameters	Assign fallback values when none are given

---

🧠💪 Practice Task for You:

1️⃣ Create a class called `Book`

- Properties: `title`, `author`, `year`
- Default values: `"Unknown Book"`, `"Anonymous"`, `2000`

2️⃣ Create two objects:

- One with values.
- One without any values.

3️⃣ Add a method `getSummary()` that logs:
`"[Title]" was written by [Author] in [Year].`

Excellent! This lecture will strengthen your understanding of how classes behave and prepare you for professional, real-world coding. Let's break it down clearly:

💡 JavaScript Lecture 26: Class Methods & Properties with Initial Values

---

✅ 1️⃣ What are Class Properties?

Properties are the variables (data) attached to an object.
In classes, they are usually initialized inside the `constructor()` method.

💻 Example: Basic Class with Properties

```
class Student {
  constructor(name, age) {
    this.name = name;  // property
    this.age = age;    // property
  }
}

let student1 = new Student("Aman", 20);
console.log(student1.name);  // Output: Aman
```

✅ Explanation:

- `this.name` and `this.age` are properties.

- Their values are set during `object creation` using the constructor.

✅ 2️⃣ Setting Initial Values to Properties (without constructor)

In modern JavaScript (ES2022+), you can directly assign **initial values** when you declare a property, outside of the `constructor`.

💻 Example: Initial Value Properties

```
class User {
  name = "Guest";    // Initial value
  age = 18;          // Initial value

  greet() {
    console.log(`Hi, ${this.name}! You are ${this.age} years old.`);
  }
}

const u1 = new User();
u1.greet();  // Output: Hi, Guest! You are 18 years old.
```

✅ Explanation:

- If you don't pass values, the object uses the predefined values.
- Cleaner and shorter than always using `constructor()` for defaults.

✅ 3️⃣ What are Class Methods?

Methods are functions that belong to a class.
They define the behaviors or actions an object can perform.

💻 **Example: Adding a Method**

```
class Student {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  introduce() {  // Method
    console.log(`Hi, I'm ${this.name}, and I'm ${this.age} years old.`);
  }
}

let student2 = new Student("Sara", 22);
student2.introduce();  // Output: Hi, I'm Sara, and I'm 22 years old.
```

✅ **Explanation:**

- `introduce()` is a method attached to every `Student` object.
- You can call it like: `student2.introduce();`

✅ 4️⃣ Properties vs Methods — Easy Table

Term	Meaning	Example

Property	Data stored in the object	`this.name = "Ali"`
Method	Action or behavior of the object	`introduce()`

🧠💡 Why use Initial Value Properties?

- Saves you from writing a constructor if defaults are enough.
- Simplifies your code when the same value is needed for all objects until overridden.

🎯💪 Practice Task for You:

1️⃣ Create a `Product` class:

- Properties: `name = "Generic Item"`, `price = 100`.
- Method: `displayInfo()` should log:
  `This product is [name] and costs $[price].`

2️⃣ Create an object and call `displayInfo()`.

Awesome! This is one of the most important and elegant features of JavaScript classes — understanding **getter and setter** makes your code professional, clean, and secure.

Let's go deep into it!

💡 JavaScript Lecture 27: Getter & Setter

✅ 1️⃣ Why Getter and Setter?

In real-world coding, you don't always want to allow direct access to an object's property.

💡 **Example:**
Let's say you have:

```
class Person {

  constructor(name) {

    this.name = name;

  }

}


let p = new Person("Rahul");

p.name = "";  // ⚠️ Invalid data!
```

👉 Direct access allows wrong or invalid data!
This is where **getter** and **setter** help.

✅ 2️⃣ What is a Getter?

A `getter` allows you to access a property **as if it's a variable**, but behind the scenes, it's calling a function.

💻 **Example:**

```
class Person {

  constructor(name) {

    this._name = name;  // private-like convention (_underscore)

  }


  get name() {

    return this._name.toUpperCase();  // always return uppercase!

  }
```

```
}

let p = new Person("Rahul");
console.log(p.name);  // Output: RAHUL
```

✅ **Explanation:**

- `get name()` runs like a function but you use it like a property: `p.name`.
- Usually used for:
  - ✔ Data formatting
  - ✔ Logic control
  - ✔ Protection.

✅ 3️⃣ What is a Setter?

A `setter` allows you to control how values are assigned to a property.

💻 **Example:**

```
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name;
  }

  set name(newName) {
    if (newName.length < 3) {
```

```
      console.log("Name too short!");
    } else {
      this._name = newName;
    }
  }
}

let p = new Person("Rahul");
p.name = "Ra";  // Output: Name too short!
console.log(p.name);  // Output: Rahul
```

✅ **Explanation:**

- Setter allows **validation** or any custom logic before changing the property.
- `_name` is used as a private-like variable (by convention).

🧠⚡ Why Use Getters and Setters?

Without Getter/Setter	With Getter/Setter
Direct access (no control)	Full control over read/write
Risk of invalid data	Safe, validated data
No extra logic on read/write	You can apply conditions easily

✅ 4️⃣ Real-Life Analogy:

Imagine an **ATM Machine**:

- 🏦 You can't directly take cash from the bank vault (direct property access ❌).
- You use the **ATM interface** (getter/setter logic ✅).

That's exactly how getter and setter work!

💡 🎯 Practice Task for You:

1️⃣ Create a `Rectangle` class:

- Properties: `_width`, `_height`.

2️⃣ Create:

- `get area()` → returns `width * height`.
- `set width(value)` → sets width only if `value > 0`.
- `set height(value)` → sets height only if `value > 0`.

3️⃣ Create an object, set values, and log the `area`.

Excellent — this is a super important lecture! Once you understand **Prototypes** deeply, JavaScript will make much more sense, especially when you start working with classes, inheritance, and advanced topics.

Let's break it down step by step!

💡 JavaScript Lecture 28: Understanding Prototypes

🧠 What is a Prototype?

In JavaScript, every object has a hidden property called `[[Prototype]]` (or `__proto__` in practice) — which is like a blueprint or a backup source for methods and properties.

When you try to access a property or method on an object:

- JavaScript first looks at the **object itself**.
- If it doesn't find it, it looks up to the `prototype`.
- If still not found, it goes to the prototype's prototype... (called **Prototype Chain**).

💻 Example 1: Default Object Prototype

```
let person = {
  name: "Rahul"
};

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

👉 You never defined `toString()` in `person`.
But it still works! Why?

Because `person` inherits `toString()` from:

person.__proto__ === Object.prototype

So if JavaScript can't find a property in `person`, it checks its prototype.

💡 Prototype Chain Simplified:

person --> Object.prototype --> null

💻 Example 2: Prototypes with Constructor Functions

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

Person.prototype.greet = function() {
  console.log(`Hi, I'm ${this.name}`);
};

const p1 = new Person("Aman");
p1.greet();  // Output: Hi, I'm Aman
```

✅ **Explanation:**

- `greet()` was not inside the object `p1` — it lives on the `Person.prototype`.
- `p1` uses the `Prototype Chain` to find and call `greet()`.

⚡ Why Use Prototypes?

- Avoids copying the same method for every object.
- Saves memory.
- Enables inheritance and shared functionality.

💻 Example 3: Class & Prototype Relationship

When you create a class in JavaScript, it uses `prototype` behind the scenes:

```
class Animal {
  speak() {
    console.log("Animal speaks");
  }
}

let dog = new Animal();
dog.speak();  // Output: Animal speaks
```

💡 Internally, JavaScript does:

```
Animal.prototype.speak = function() { ... }
```

So the concept of `prototype` is still there even when you use modern `class` syntax!

🧠 Prototype Summary Table

Concept	Meaning
`prototype`	An object that provides shared properties/methods.
`__proto__`	Actual reference to prototype (object's hidden link).
Prototype Chain	JavaScript searches properties up the chain until `null`.

🎯 💪 Practice Task for You:

1️⃣ Create a `Car` constructor function.

2️⃣ Add a property: `brand`.

3️⃣ Add a method to the `Car.prototype`: `showBrand() → logs: This car is a [brand]`.

4️⃣ Create two car objects and call `showBrand()` on both.

Excellent — this is one of the most important and most asked interview topics!
Today we'll make **Closure** super simple for you.

💡 JavaScript Lecture 29: First Program in Closure & Understanding Closure

---

✅ 1️⃣ What is a Closure?

A **Closure** is created when:

> 👉 A function is able to "remember" and access its **outer scope** (variables) — even after the outer function has finished executing.

This is a powerful feature of JavaScript!

💡 Simple Formula:

Closure = Function + Outer Lexical Environment

💻 Example 1: Basic Closure

function outer() {

```
  let name = "Rahul";

  function inner() {
    console.log("Hello " + name);
  }

  return inner;
}

const greet = outer();  // outer() has finished
greet();  // Output: Hello Rahul
```

✅ **Explanation:**

- `outer()` defines a variable `name` and returns `inner()`.
- Even though `outer()` is done executing, `inner()` still remembers `name` because of **closure**.

🧠💡 Why is this useful?

Closures allow:

- **Data protection (encapsulation)** — variables stay private.
- **Function factories** — functions that create other functions.
- **Asynchronous programming** — keeping access to old data even after a delay.

💻 Example 2: Closure as a Counter

```
function createCounter() {
  let count = 0;

  return function() {
    count++;
    console.log("Count is: " + count);
  };
}

const counter = createCounter();

counter();  // Output: Count is: 1
counter();  // Output: Count is: 2
counter();  // Output: Count is: 3
```

✅ **Explanation:**

- `count` is hidden inside the closure.
- No one can directly change `count`, only the returned function can.

⚡ Common Interview Question:

> Q: What is a Closure in JavaScript?
> A: A closure is a function that remembers the variables from its outer scope even after the outer function has finished.

🔥 Why is Closure Important?

- Used in real-world programming: event handlers, timers, iterators, module patterns.
- Helps avoid polluting the global scope.
- Essential for understanding JavaScript's **lexical scoping**.

🎯 💪 Practice Task for You:

1️⃣ Write a function `greetGenerator` which takes a name as input and returns a function. When that returned function is called, it should log:
`Hello [name]!`

✅ Example:

```
const greetRahul = greetGenerator("Rahul");

greetRahul();  // Output: Hello Rahul!
```

Awesome — this is another must-know concept for any JavaScript developer!
Let's go deep into:

💡 JavaScript Lecture 30: Understanding Callback Functions

✅ 1️⃣ What is a Callback Function?

A **Callback** is:

> 👉 A function that is **passed as an argument to another function**, and that function will **call it back** (execute it) later.

💡 Why Use Callback Functions?

- JavaScript is **single-threaded** (runs one thing at a time).
- To avoid waiting/blocking for things like:
    - Server responses.
    - File reading.
    - Time delays.

Callbacks allow code to continue running and handle tasks **when ready**.

💻 Example 1: Basic Callback

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

function sayBye() {
  console.log("Goodbye!");
}

greet("Rahul", sayBye);
```

✅ **Explanation:**

- `sayBye` is passed into `greet` as a **callback**.
- Once `greet` finishes printing, it "calls back" the `sayBye` function.

💻 Example 2: Callback in setTimeout (Real-World Usage)

```
function welcome() {
  console.log("Welcome to the website!");
}

setTimeout(welcome, 2000);  // Callback will execute after 2 seconds
```

✅ **Explanation:**

- JavaScript doesn't pause for 2 seconds.
- `setTimeout` registers `welcome` as a **callback** to be run later.

⚡ Benefits of Callbacks:

Without Callback	With Callback
Fixed behavior	Custom behavior on demand
Less flexible	Highly reusable functions
Can't handle asynchronous tasks	Perfect for asynchronous code

---

💡 Callback Visualization

Main Function

└── Calls: Callback Function (when ready!)

💡 Example 3: Array Callback Example (`forEach`)

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


numbers.forEach(function(num) {
  console.log(num * 2);
});
```

✅ Here, `function(num) {}` is a **callback** used inside `forEach`.
JavaScript will automatically call your function for every item in the array!

⚡ Callback Summary:

Term	Meaning
Callback Function	A function passed as an argument
When is it used?	When the code must run **later** (async)

🎯 💪 Practice Task for You:

1️⃣ Create a function `doMath` that takes two numbers and a callback.

2️⃣ The callback should do an operation: add, subtract, multiply, or divide.

💻 Example:

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

function doMath(a, b, callback) {
  console.log("Result: " + callback(a, b));
}

doMath(5, 3, add);  // Output: Result: 8
```

Great move! You've now reached one of the most powerful modern JavaScript topics:

💡 JavaScript Lecture 31: Promise Constructor

🧠 What is a Promise?

A **Promise** in JavaScript is:

> 👉 An object that represents the **future result** of an asynchronous operation — it can be successful or failed.

It promises to give you something:

- now ✅ (immediately resolved),
- later 🕐 (after some time), or
- never ❌ (if an error happens).

💡 Promise States

A Promise can be in one of these 3 states:

State	Meaning
`Pending`	Still waiting for the task to finish
`Fulfilled`	Task completed successfully (`resolve`)
`Rejected`	Task failed (`reject`)

💻 Syntax of Promise Constructor

```
let promise = new Promise(function(resolve, reject) {
  // Your async task here
  if (/* success condition */) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

💻 Example 1: Basic Promise

```
let promise = new Promise(function(resolve, reject) {
  let success = true;

  if (success) {
    resolve("✅ Task completed successfully!");
  } else {
    reject("❌ Task failed!");
  }
});

promise
  .then(function(result) {
    console.log(result);
  })
  .catch(function(error) {
    console.log(error);
  });
```

✅ **Explanation:**

- `resolve` means: success! Send the result.
- `reject` means: failed! Send the error.
- `.then()` handles success.
- `.catch()` handles errors.

💡 Example 2: Promise with setTimeout (simulate server delay)

```
let loadingData = new Promise(function(resolve, reject) {
  setTimeout(() => {
    let serverResponse = true;  // change this to false to test rejection

    if (serverResponse) {
      resolve("🎉 Data loaded from server!");
    } else {
      reject("🚨 Failed to load data!");
    }
  }, 2000);
});

loadingData
  .then((message) => console.log(message))
  .catch((err) => console.log(err));
```

✅ **Explanation:**

- Simulates loading data from a server.
- After 2 seconds, `resolve` or `reject` is called.
- The `.then()` will run if successful.
- The `.catch()` will run if failed.

💡 Why Promises?

Before promises, JavaScript used **callback hell**:

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log(finalResult);
    });
  });
});
```

This was hard to read and debug!

Promises solve this by allowing **clean chaining**:

```
doSomething()
  .then(doSomethingElse)
  .then(doThirdThing)
  .then(console.log)
  .catch(console.error);
```

🎯 💪 Practice Task for You:

1️⃣ Create a Promise named `orderPizza`.
Inside the promise:

- If `orderAccepted = true`, call `resolve('🍕 Your pizza is on the way!')`.
- If `orderAccepted = false`, call `reject('❌ Order failed, please try again.')`.

Test both `.then()` and `.catch()`.

Excellent! You're now stepping into one of the most modern and professional parts of JavaScript:

💡 JavaScript Lecture 32: Async / Await & Fetch API

✅ 1️⃣ What is Async / Await?

Async / Await is modern JavaScript syntax to handle **Promises** in a cleaner and more readable way — instead of using `.then()` and `.catch()`.

💡 Syntax:

```
async function myFunction() {
  try {
    let response = await somePromise;
    console.log(response);
  } catch (error) {
    console.log(error);
  }
}
```

⚡ async

- Makes a function return a **Promise**.
- Allows you to use `await` inside it.

⚡ `await`

- Waits for a **Promise** to settle.
- Makes asynchronous code look like **synchronous** code.

💻 Example 1: Basic Async / Await

```
function makeTea() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("☕ Tea is ready!"), 2000);
  });
}

async function serveTea() {
  console.log("Making tea...");
  let tea = await makeTea();
  console.log(tea);
}

serveTea();
```

✅ Output:

Making tea...

☕ Tea is ready! (after 2 sec)

✅ 2️⃣ What is Fetch API?

Fetch API is used to make network requests (like calling an API).
It returns a **Promise**.

💡 Syntax:

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

💻 Example 2: Fetch API with Async / Await

```
async function fetchData() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.log("❌ Error:", error);
  }
}

fetchData();
```

✅ Explanation:

- `fetch()` sends a request to the server.
- `await` pauses until data is fetched.
- `response.json()` turns the raw data into a JavaScript object.
- Errors are handled with `try...catch`.

💡 Why Async / Await + Fetch is Useful?

Old Way (Callback / Promise)	New Way (Async / Await)
Harder to read	Looks like normal code
`.then()` / `.catch()` chaining	Simple `try...catch` blocks
More confusing error handling	Centralized error handling

🎯 💪 Practice Task for You:

1️⃣ Write an `async function` called `getUserData`.
Inside it:

- Use `fetch()` to call: `https://jsonplaceholder.typicode.com/users/1`.
- Use `await` to handle the response.
- Use `console.log()` to print the data.

💡 Expected Output:

{

```
  id: 1,
  name: "Leanne Graham",
  username: "Bret",
  ...
}
```

Sure — great timing! Understanding **Events** is super important for real-world JavaScript, especially when you work with the DOM. Let's go deep!

💡 JavaScript Lecture 33: Events

🔥 What is an Event?

An **Event** is:

👉 An action that happens in the browser, like:

- Clicking a button
- Typing into a form
- Hovering the mouse
- Pressing a key
- Page fully loading

Your JavaScript can "listen" for these events and **respond** to them!

💡 Common Browser Events

Event Name	Triggered When...
`click`	A user clicks on an element
`mouseover`	Mouse hovers over an element
`mouseout`	Mouse leaves an element
`keydown`	User presses a key
`keyup`	User releases a key
`submit`	A form is submitted
`load`	Page has fully loaded
`change`	Input value is changed

💻 Example 1: Handling a Button Click

```
<button id="myBtn">Click Me</button>

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

  button.addEventListener("click", function() {
```

```
  alert("You clicked the button!");
});
</script>
```

✅ **Explanation:**

- `addEventListener()` listens for an event.
- `"click"` is the event type.
- When the button is clicked, the function runs.

💡 Why use `addEventListener` instead of `onclick`?

`onclick` (old)	`addEventListener` (modern)
Only one handler allowed	Multiple handlers possible
Inline JavaScript	Clean separation of logic
Less flexible	More flexible

💻 Example 2: Mouseover Event

```
<div id="box" style="width:100px; height:100px; background:red;"></div>

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

```
  box.addEventListener("mouseover", function() {
    box.style.background = "green";
  });

  box.addEventListener("mouseout", function() {
    box.style.background = "red";
  });
</script>
```

✅ **Explanation:**

- Changes color to green on mouseover.
- Changes back to red when the mouse leaves.

💡 Event Object (`event`)

When an event happens, JavaScript passes an `event` object automatically that contains:

- Info about the event.
- Info about the element.
- Mouse position, key pressed, etc.

💻 Example:

```
button.addEventListener("click", function(event) {
  console.log(event.target);  // shows which element was clicked
});
```

🎯 💪 Practice Task for You:

1️⃣ Create a `button` in HTML.
2️⃣ Write JavaScript so that:

- When clicked, the button text changes to "Clicked!".
- After 3 seconds, change the text back to "Click Me".

Excellent — you're moving into real-world JavaScript territory now!
This topic is super important when you're handling multiple elements and events. Let's break it down!

💡 JavaScript Lecture 34: Event Bubbling & Event Capturing

🔥 What is Event Propagation?

When you interact with a child element (like clicking a button inside a `div`), the event doesn't just stop there — it "travels" through the DOM.

This behavior is called **Event Propagation**.

It happens in **two phases**:

Phase	Direction	Default?
`Capturing Phase`	Document → Root → Target (top-down)	❌ No
`Bubbling Phase`	Target → Root → Document (bottom-up)	✅ Yes

💡 1️⃣ Event Bubbling (Default)

In **Bubbling**, the event happens on the target element first, then moves **upward** to its parents.

💻 Example: Bubbling

```
<div id="parent" style="padding:20px; background:#ddd;">
  <button id="child">Click Me</button>
</div>

<script>
  document.getElementById("parent").addEventListener("click", () => {
    console.log("👨 Parent clicked");
  });

  document.getElementById("child").addEventListener("click", () => {
    console.log("👦 Child clicked");
  });
</script>
```

👉 Output when you click the button:

👦 Child clicked

👨 Parent clicked

✅ Explanation:

- First the `button` runs its event.
- Then the event bubbles up to the `div`.

💡 2️⃣ Event Capturing (Rarely Used)

In **Capturing**, the event travels from the `document` down to the target.

You can activate capturing like this:

element.addEventListener("click", callback, true);

`true` = listen during **capture phase**
`false` (default) = listen during **bubbling phase**

💻 **Example: Capturing**

```
document.getElementById("parent").addEventListener("click", () => {
  console.log("👨 Parent clicked");
}, true);

document.getElementById("child").addEventListener("click", () => {
  console.log("🧒 Child clicked");
}, true);
```

👉 **Output when you click the button:**

👨 Parent clicked

🧒 Child clicked

✅ **Explanation:**

- In capturing, the parent is triggered **first**.
- The event flows top-down.

💡 3️⃣ Stop Event Propagation

Sometimes you want to stop the event from going further.

You can do that with:

event.stopPropagation();

💻 Example:

```
document.getElementById("child").addEventListener("click", function(event) {
  console.log("🧒 Child clicked");
  event.stopPropagation(); // stop going to parent
});

document.getElementById("parent").addEventListener("click", function() {
  console.log("👨 Parent clicked");
});
```

👉 **Output:**

🧒 Child clicked

✅ The event **stops** at the child, and the parent never gets triggered.

🧠 💡 Why is this important?

- To avoid unexpected behavior when clicking nested elements.
- For efficient event handling in large applications (like event delegation).
- For creating consistent behavior across complex layouts.

🎯 💪 Practice Task for You:

1️⃣ Create:

```
<div id="outer" style="padding:20px; background:lightblue;">
  <div id="inner" style="padding:20px; background:lightgreen;">
    <button id="btn">Click Me</button>
  </div>
</div>
```

2️⃣ Add JavaScript:

- Log `"Outer Clicked"` when `outer` is clicked.
- Log `"Inner Clicked"` when `inner` is clicked.
- Log `"Button Clicked"` when `btn` is clicked.

3️⃣ Try with:

- Bubbling mode (`false`) — default.
- Capturing mode (`true`).

🎯 **Bonus:** Add `event.stopPropagation()` on `btn` and see what happens!

👉 **"Lecture 34 practice done ✅"**

✅ Mission Complete! You've done an amazing job finishing your JavaScript basics — I'm proud of your consistency and effort! Now you've got a perfect revision notebook ready.