Introduction
&
Setup

Let’s try to understand How Machine or Computer Works?

We Need to give some instructions to perform work on computer.

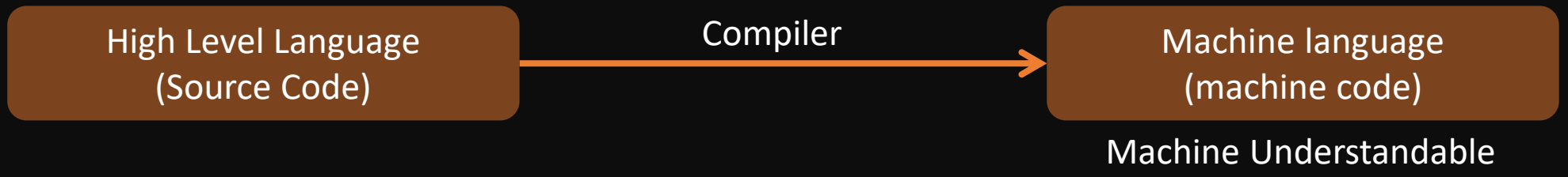
So, the question is Does the machine understand human language?

Absolutely Not ❌

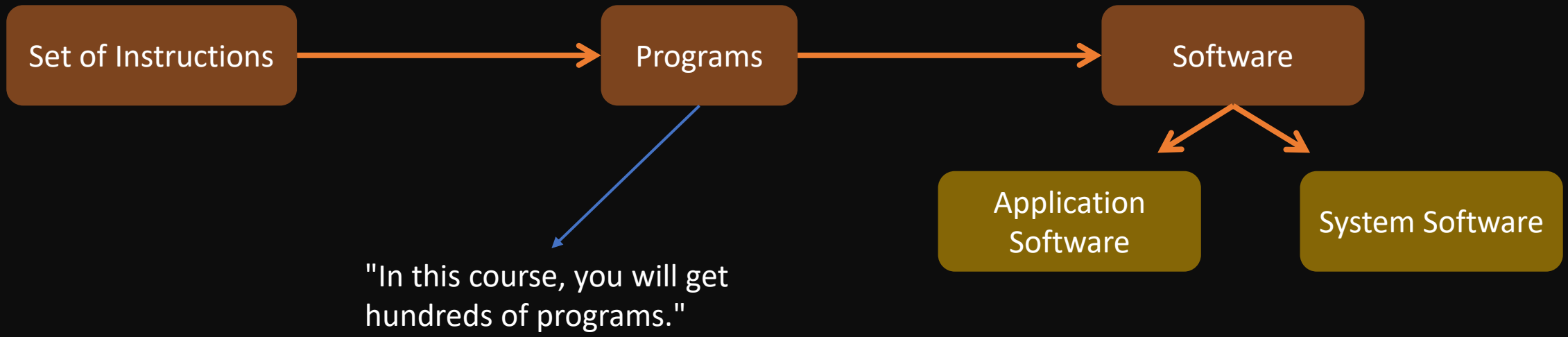
Machine Or Computer only understand machine language in terms of 0 & 1 or ON & OFF (binary number system).
Machine language is the fastest language, but the problem is machine language is very tedious and not easy to understand.

To avoid all these problems, High level languages come into play. Since high level language uses English keywords and meaningful syntaxes, high level languages are easy to understand.

So, nowadays we give our instructions in any high-level language like C#, C++, javascript etc.



Stages To Create Any Software



Let’s Start With Some Technical Terms

Program : A set of instructions given to computer to perform a particular task is known as program or computer program

Programming : The Process of writing computer program is known as programming

Programmer : A Person who write computer program is called programmer.

Language: It is a medium through which one can communicate with other

Programming language : A language used to stablish the communication between human to computer or vice versa or computer to computer

Compiler : A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language).

Algorithm : Step by step planning to solve a problem is called Algorithm

Flowchart : Diagrammatic representation or Pictorial representation to show the process of execution is known as flowchart

Pseudo code: Generic Code form of Algorithm to solve a problem is called Pseudo Code

Algorithm : Step by step planning to solve a problem is called Algorithm.

E.g. : Algorithm to add two numbers:

Step-1 : Start
Step-2 : Read Two numbers
Step-3 : Add both numbers
Step-4 : Print the Results
Step-5 : Stop

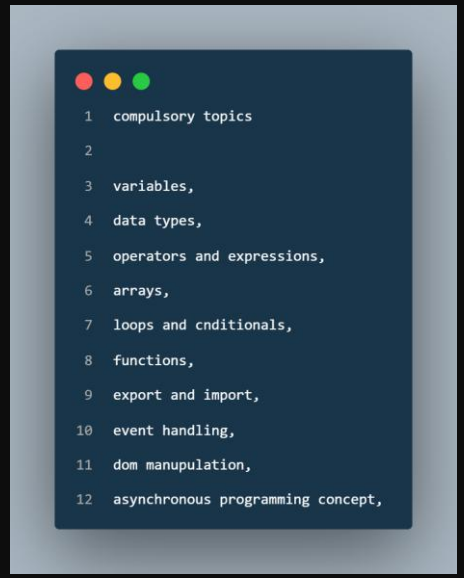
YouTube Algorithms

Analogy :
Suppose we have to go to the computer lab

Step-1 : Will take the notebook.
Step-2 : Will leave class to go to the lab.
Step-3 : Will take off our shoes outside lab.
Step-4 : Will sit at their respective places in the lab

JavaScript is a programming language used to create interactive web pages, applications, and games. It's a core technology of the World Wide Web.

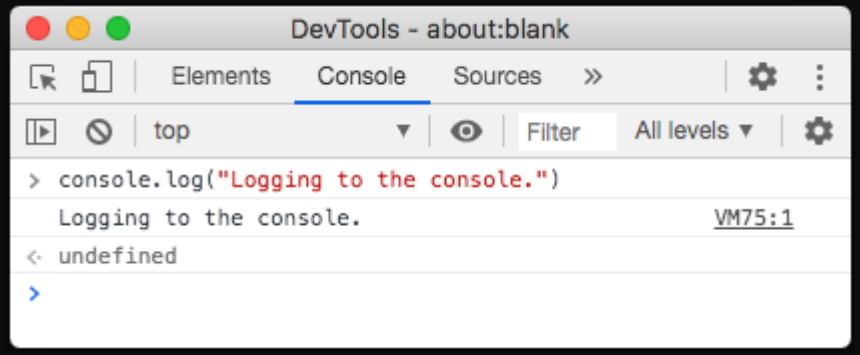
In the field of Web Development , JS is called as the engine or brain of the website.



History You Need To Know:

- Originally, JavaScript was named Mocha, later changed to Live Script, and finally to JavaScript.
- Since its launch on December 4, 1995, JavaScript has continuously improved. Over the years, it has become one of the most powerful programming languages. Its ecosystem includes frameworks like React, Node.js, and Vue.js

1. Use JavaScript In Browser Console



2. Use JavaScript In VS Code using node js

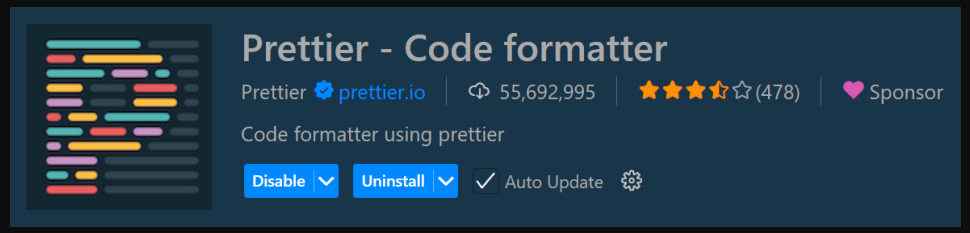
NOTE:

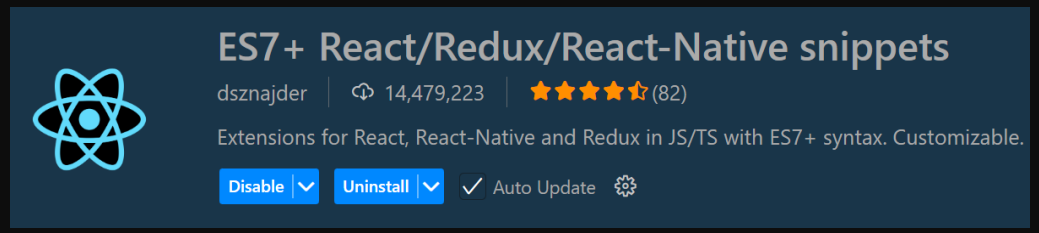
- ❑ Some of the functions are browser specific which you can't use in VS Code directly (ex- alert, prompt).
- ❑ To use these commands in Vs Code as well, you need to follow the 3rd connection method.

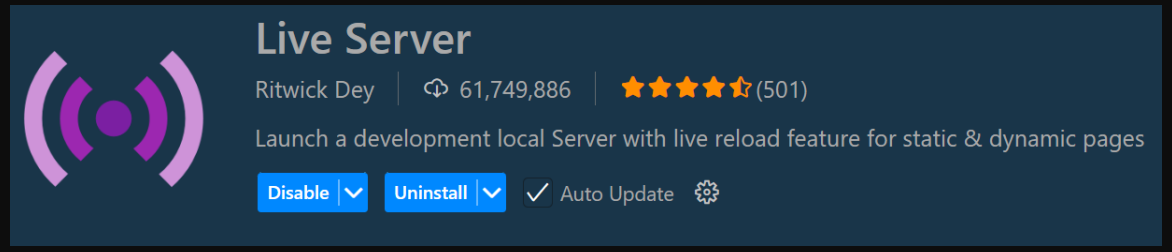
3. Connect browser & nodejs environment

- ❑ Create index.html
- ❑ Connect you javascript (.js) file with index.html
- ❑ Run your html file on browser (use live server extension for real-time update)
- ❑ Now you are free to use javascirpt code in your browser as well as in VS Code

Important Extensions & Settings:







Settings:

- ❑ Auto Save Setting
- ❑ Format On Save Setting

Please Keep in mind:

- Don’t compare javascript with java.
- JavaScript helps in manipulation of webpages.
- JavaScript Responds to user actions. (like on mouse click, keyboard interactivity, etc)
- JavaScript helps to interact with server or database. (Fetches and sends data to server)

NOTE:
- We learn JavaScript with use cases.

Variables & Datatypes

Console in JavaScript

Console.log is used to print/log a message/output to the console.

```
console.log("Hello Manas!");
```

Comments in JavaScript

Part of code which are not going to execute, and just for reference or good readability of code.

```
// first name and last name are of string type
let firstName = "John";
let lastName = "Doe";
// in the below code we concatenate the first and last names together using + opearator
let fullName = firstName + " " + lastName;
console.log("Full Name:", fullName);
```

Two Types of Comment:

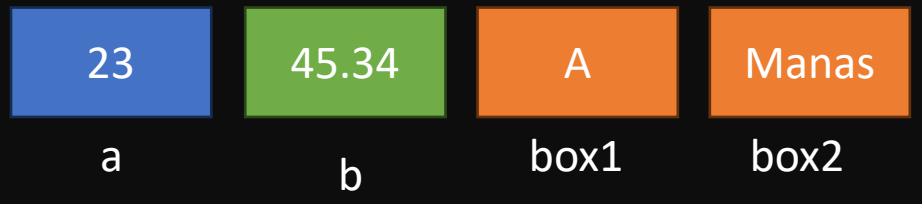
```
// single line comment

/*
    multi
    line
    comment
*/
```

Variables:

A variable is like a container that holds data. It is used to store data that can be used and manipulated throughout your program. .

Think of a variable as a named container that holds a value.

In JavaScript, a variable can contain any type of data.



Three Stages of a Variable:

1. Declaration
2. Initialization
3. Use

1. Declaration

```
let name;
```

2. Initialization

```
name = "Manas Kumar Lal";
```

Here "=" is an assignment operator

3. Use

```
console.log(name);
```

Examples:

```
let a;
a = 52;
```

```
let a,b,c;
a=10,
b=20,
c=30
```

Declaration + Initialization:

```
let name = "Manas Kumar Lal";
```

```
let a=5, b=10, c=15;

or

let a=5;
let b=10;
let c=15;
```

Let, const & var

var : Variable can be re-declared & updated. A global scope variable.

let : Variable cannot be re-declared but can be updated. A block scope variable.

const : Variable cannot be re-declared or updated. A block scope variable.
(variables declared with const keyword must have declaration & initialization both)

fullname

namefromdatabaseuser

Rules for variable declaration:

1. Variable names are case sensitive;
2. “a” & “A” is different.
3. Only letters, digits, underscore(_) and $ is allowed. (not even space)
4. Only a letter, underscore(_) or $ should be 1st character.
5. Reserved words cannot be variable names.

Camel Case → nameFromDatabaseUser

Snake Case → name_from_database_user

Pascal Case → NameFromDatabaseUser

Kebab Case → name-from-data-User

Let a;

Let alpha;

Primary (Primitive) Datatypes:

- Number, String, Boolean, Undefined, Null, BigInt, Symbol
- Use typeof operator to identify the type of variable

Different Types of cases exists in Programming:

- camelCase, snake_case, PascalCase, and kebab-case

Use Case 1:

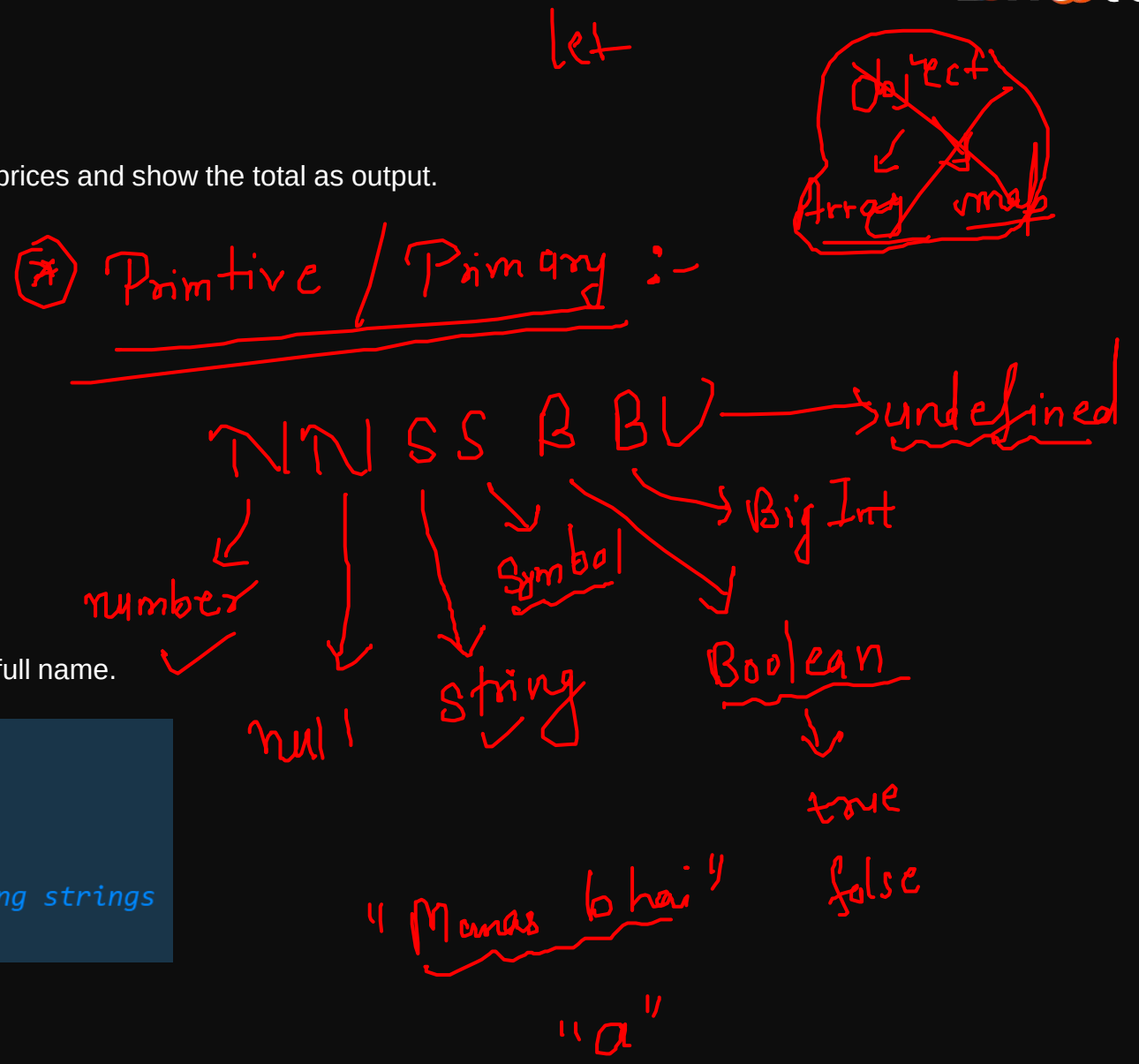
Suppose price of two products comes from database, you need to store these prices and show the total as output.

```
let price1 = 499;
let price2 = 500;
let total = price1 + price2;
console.log(total);
```

Use Case 2:

A user enters their first and last name into a form, and you need to display the full name.

```
let firstName = "John";
let lastName = "Doe";
let fullName = firstName + " " + lastName; // Concatenating strings
console.log("Full Name:", fullName);
```



JavaScript is a Dynamically Typed Language & Forgiving Language

```
let alpha = 3;
console.log(alpha)

alpha = "manas";
console.log(alpha)

alpha = 3.225;
console.log(alpha)
```

You don’t need to specify the type of a variable,
even variable dynamically change its type respective to the data assigned in it.

```
myVar = 10;
// Works, but should be declared with let or const
console.log(myVar); // Output: 10
```

```
console.log("5" + 5);
// Output: "55" (number 5 turns into a string)
console.log("5" - 2);
// Output: 3 (string "5" turns into a number)
```

```
let alpha = 5
```

```
let alpha = 5
// even if semicolon is missing, it will not affect the execution
```

Forgive and does not produce any error

Q1: How do you declare a variable in JavaScript?

Q2: What is the difference between var, let, and const?

Q3: Can you change the value of a const variable?

Q4: What will happen if you use a variable without declaring it?

Q5: What is the default value of an uninitialized variable in JavaScript?

Q6: What are the primitive data types in JavaScript?

Q7: What is the difference between null and undefined?

Q8: Is JavaScript a statically typed or dynamically typed language?

Q9: What will be the output of typeof null?

Q10: What happens when you add a number and a string in JavaScript?

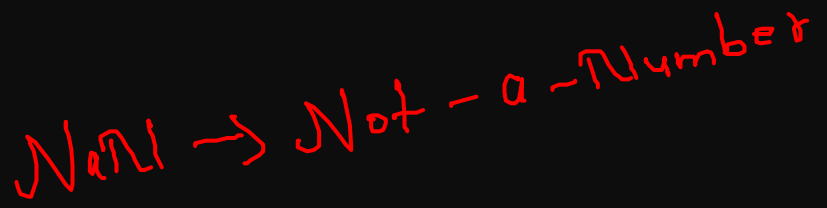
Q11: What is type coercion? Give an example.

Q12: How can you manually convert a string to a number in JavaScript?

Q13: What is the result of "5" - 3 in JavaScript?

Q14: What is NaN in JavaScript, and how do you check if a value is NaN?

Q15: How do you check the type of a variable in JavaScript?



Operators

Operators in JavaScript:

An operator is a symbol that tells the computer to do something with values.

```
let result = 5 + 3;
```

‘+’ is the operator, it tells JavaScript to add 5 & 3.

5 and 3 are the operands (value that an operator works on)

👉 Operands are the values.
👉 Operators do something to those values.

TYPES OF OPERATORS:

S.No.	**Operators**	Operator Name	**Example**
1	+, -, *, /, %, **	Arithmetic	5 + 10
2	=, +=, -=, *=, /=, %=, **=	Assignment	a += 15
3	==, ===, !=, !==, >, <, >=, <=	Comparison	5 < 10
4	&&, ||, !	Logical	true && false
5	++, --	Increment & Decrement	a++, --b
6	Condition ? True : False	Ternary	5 < 10 ? 5 : 10
7	&, |, ~, ^, <<, >>	Bitwise	a << 2

1. Arithmetic Operator (Math stuff)

These are like what you use in school math:

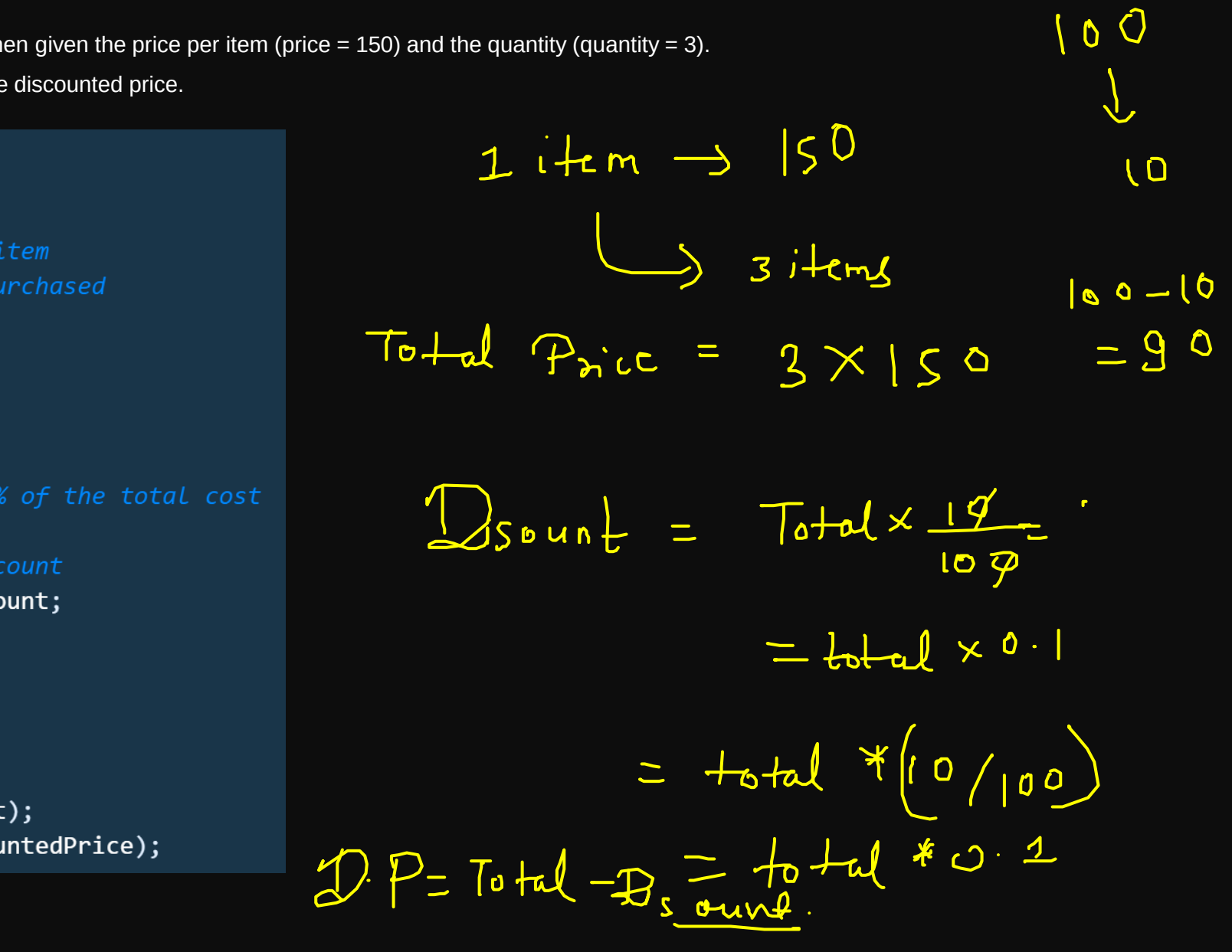
Operators	What it does	**Example**	Result
+	Add	5 + 2	7
-	Subtract	5 – 2	3
*	Multiply	5 * 2	10
/	Divide	10 / 2	5
%	Modulo (Remainder)	5 % 2	1
**	Exponent (Power)	2 ** 3	8

Challenge1:

On a shopping website, calculate the total cost of a product when given the price per item (price = 150) and the quantity (quantity = 3).

Also, calculate a 10% discount on the total cost and display the discounted price.

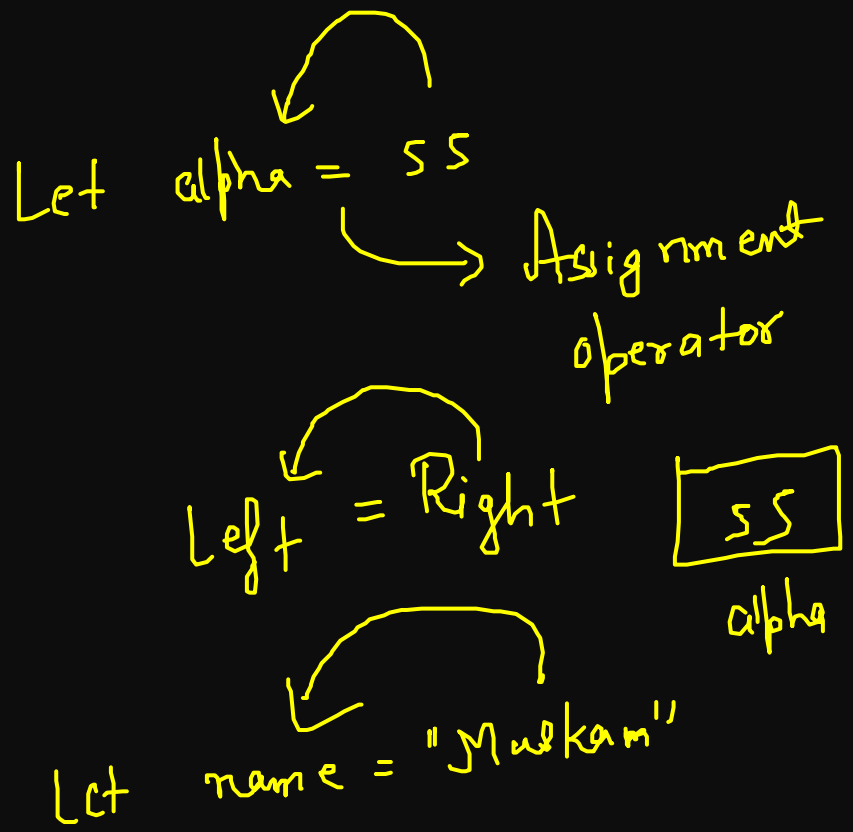
```
// Inputs: Price per item and quantity
let price = 150; // Price of a single item
let quantity = 3; // Number of items purchased

// Calculate the total cost
let totalCost = price * quantity;

// Calculate a 10% discount
let discount = totalCost * 0.10; // 10% of the total cost

// Calculate the final price after discount
let discountedPrice = totalCost - discount;

// Display the results
console.log("Price per item:", price);
console.log("Quantity:", quantity);
console.log("Total Cost:", totalCost);
console.log("Discount (10%):", discount);
console.log("Discounted Price:", discountedPrice);
```

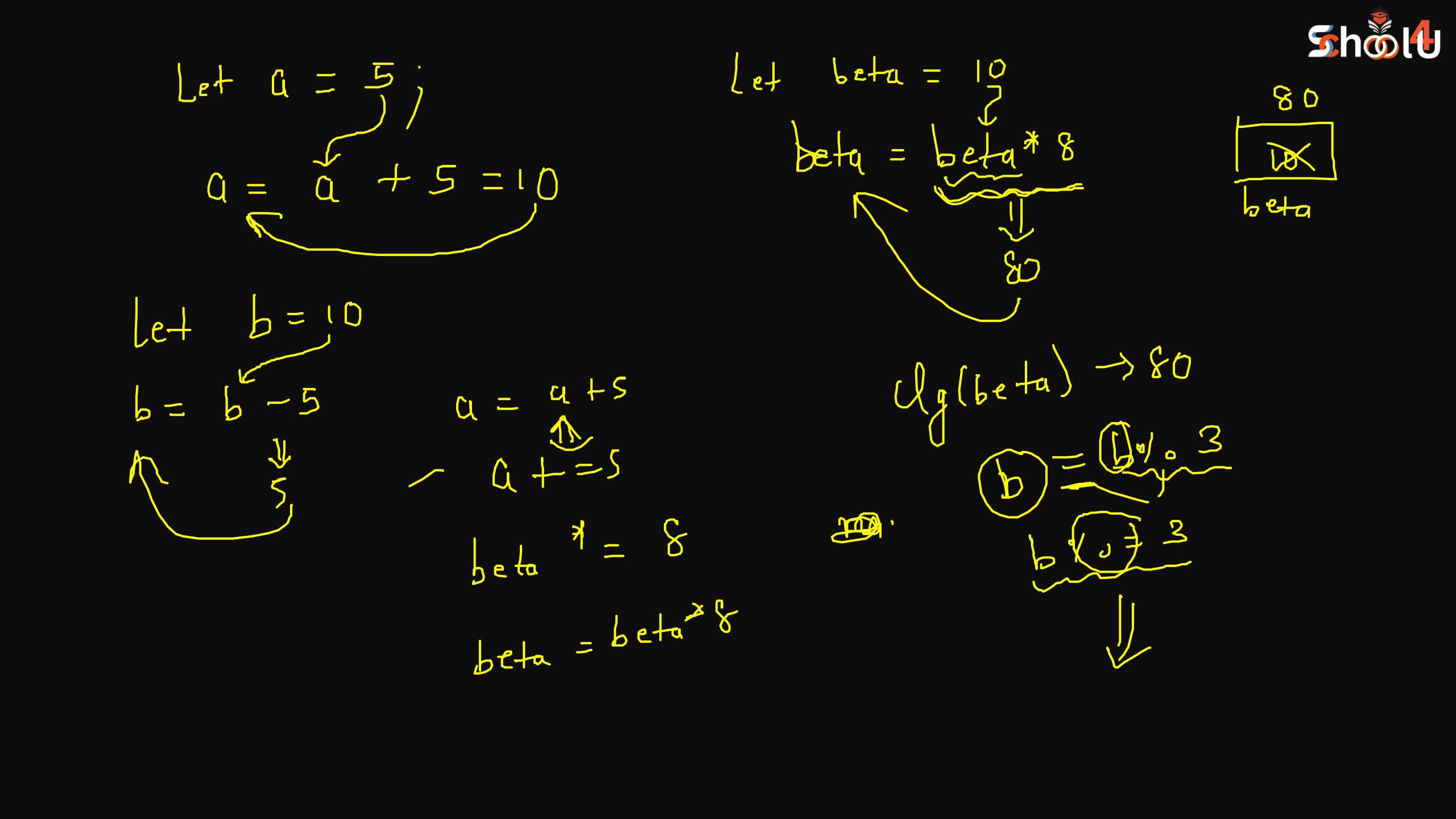


2. Assignment Operators (Give values to variables)

These assign values to variables. You'll use = a lot.

Operators	What it does	Example	Meaning
=	Assign	x = 5	x gets 5
+=	Add and assign	x += 2	x = x + 2
-=	Subtract and assign	x -= 2	x = x – 2
*=	Multiply and assign	x *= 2	x = x * 2
/=	Divide and assign	x /= 2	x = x / 2
%=	Modulo and assign	x %= 2	x = x % 2
**=	Exponent (Power) and assign	x **= 2	x = x ** 2

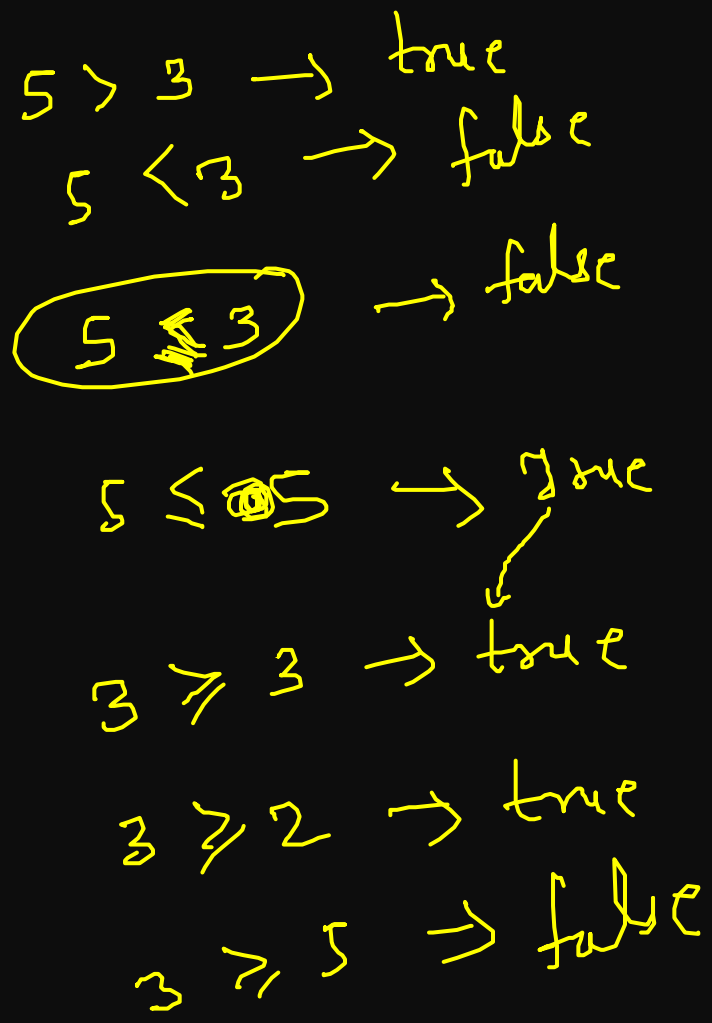


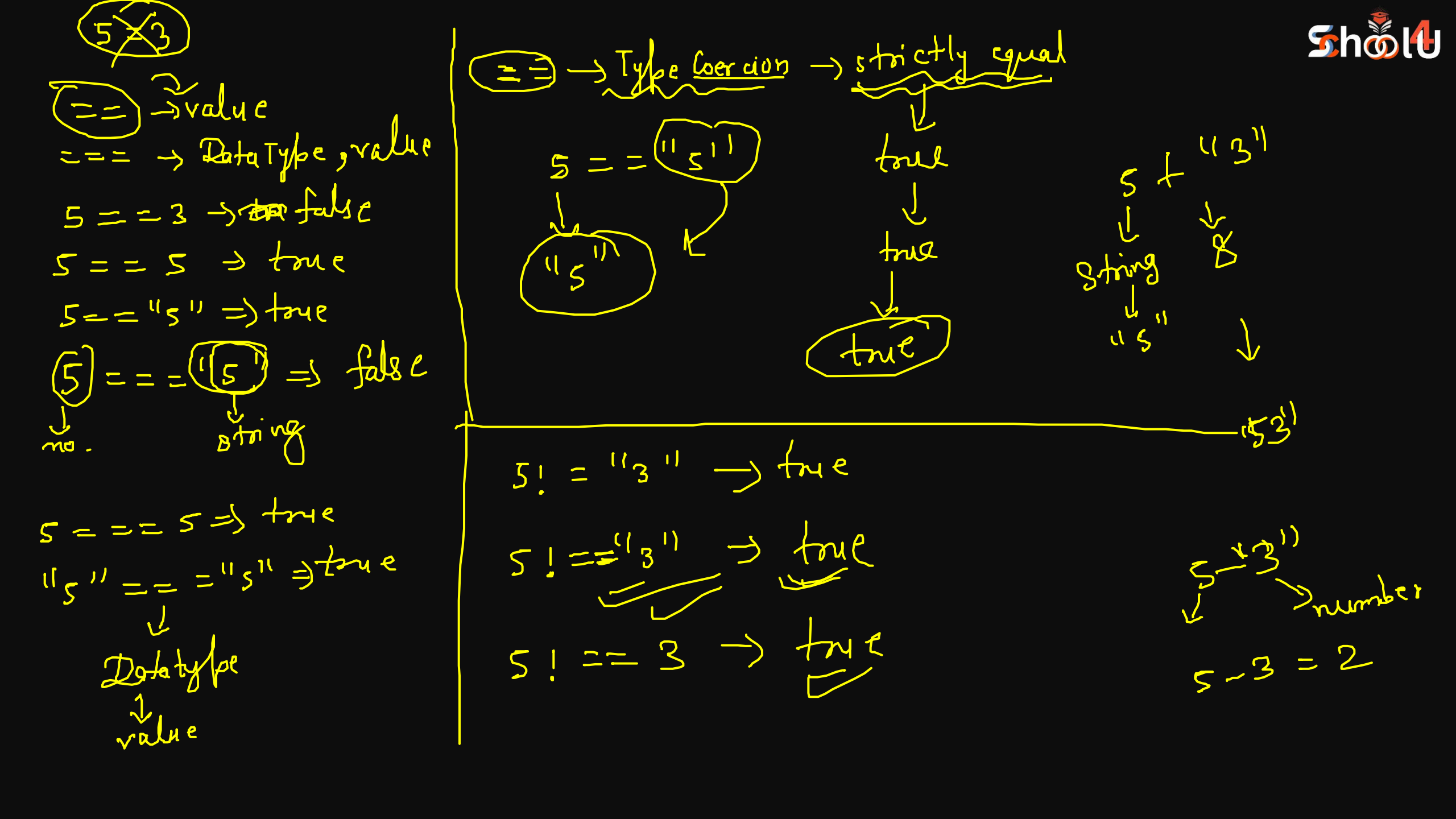

Let a = 5;
a = a + 5 = 10
Let b = 10
b = b - 5
5
a = a + 5
a += 5
beta *= 8
beta = beta * 8
Let beta = 10
beta = beta * 8
80
80
beta
log(beta) → 80
b = b - 3
b -= 3

3. Comparison Operators (Check things)

These help you compare values. They give true or false.

Operators	What it does	Example	Result
==	Loosely equal	5 == “5”	true
===	Strictly equal	5 === “5”	false
!=	Loosely not equal	5 != “5”	false
!==	Strictly not equal	5 !== “5”	true
>	Greater than	5 > 2	true
<	Less than	5 < 2	false
>=	Greater than or equal to	5 >= 5	true
<=	Less than or equal to	5 <= 3	false




5 ✗ 3
==
→ value
===
→ DataType, value
5 === 3 → false
5 === 5 → true
5 == "5" ⇒ true
5 === "5" ⇒ false
5
no.
"5"
string
5 === 5 ⇒ true
"5" === "5" ⇒ true
Datatype
value
==
→ Type Coercion
→ strictly equal
5 == "5"
"5"
"5"
true
true
true
5 + "3"
5
string
"5"
8
"53"
5 != "3" → true
5 !== "3" → true
5 != 3 → true
5 - "3"
number
5 - 3 = 2

4. Logical Operators (Combine conditions)

Used when you're checking more than one thing.

Operators	What it does	**Example**	Result
&&	AND (both must be true)	true && true	true
||	OR (either one is true)	true || false	true
!	NOT (flips true/false)	!true	false

&& (AND)

true false → false

a	b	?
true	false	false
false	true	false
false	false	false
true	true	true

|| (OR)

a	b	
t	f	t
f	t	t
t	t	t
f	f	f

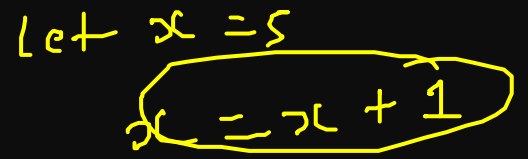
a) x > 5 && x < 1000

! (NOT)

!f → t

!t → f

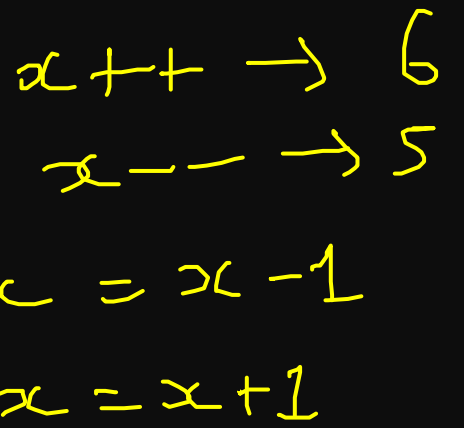
!(x > 5)



5. Increment/Decrement Operators (Add or subtract 1)

Quick way to increase or decrease a value.

Operators	What it does	**Example**	Result
+ +	Add 1	x++	x = x + 1
- -	Subtract 1	x- -	x = x - 1

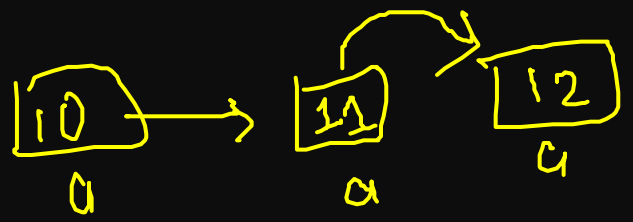


Increment

Post++ Pre++

Decrement

Post-- Pre--



let a = 10

let result = a++ + ++a - 10;

10 + 12 - 10

22 - 10 = 12

6: Ternary Operator (Shortcut for if/else)

```
condition ? execute if condition is true : execute if condition is false;
```

Example:

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

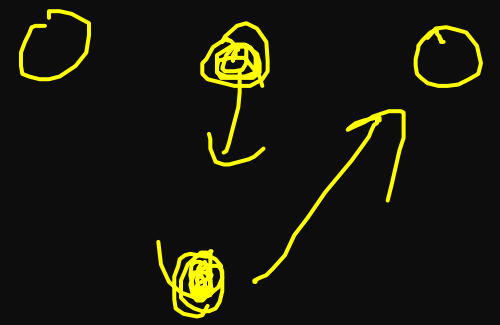
It reads like:

"If age is 18 or more, result is 'Adult', otherwise it's 'Minor'"

find the largest no. from given 3 numbers.

let a = 5 , b = 2 , c = 12

let result = a > b ? (a > c ? a : c) : (b > c ? b : c)



Challenge2:

On a booking website, check if the user's age is valid for booking:
Age should be at least 18.
Write a condition to check and display a message: "Eligible for booking" if the user is 18 or older. "Not eligible for booking" otherwise.

```javascript
// Input: User's age
let age = 18; // Example input, can be changed dynamically

// Use a ternary operator to determine the message
let message = age >= 18 ? "Eligible for booking" : "Not eligible for booking";

// Display the message
console.log(message);
```

Challenge 3:

On a login page, verify the user's credentials: Check if username is not empty AND password is not empty (&& operator).
If either is empty, display an error message: "Both fields are required."

```javascript
// Input: Username and password
let username = ""; // Example: Empty username
let password = ""; // Example: Empty password

// Check credentials using the ternary operator
let message = (username && password)
    ? "Login successful"
    : "Both fields are required.";

// Display the message
console.log(message);
```

7. Bitwise Operators:

super useful in certain situations — especially when you're working at a low level (like with individual bits of data)

Operators	What it does	Example	Meaning
&	Bitwise AND	a & b	a AND b
|	Bitwise OR	a | b	a OR b
~	Bitwise NOT	~a	NOT of a
^	XOR	a ^ b	a XOR b
<<	Left Shift	a << 2	a Left Shift by 2
>>	Right Shift	a >> 1	a Right Shift by 2

They're more common in:

- Algorithms
- Game engines
- Performance-critical code
- System-level programming

~(-5 → ---- 1 1 1 0 1 1

not

---00000100

↓

4

↓

$-(x+1)$

$-(-5+1)$

$= -(-4)$

$= 4$

$x = 3$

$-(3+1)$

$-4 = -4$

$x = 0$

$-(0+1) = -1$

Let’s categorize operator on the basis of number of operands:

Operand Count	Type	Operators
1 (Unary)	Arithmetic	+, -, ++, --
	Logical	!
	Type-checking	typeof, void, delete
Binary	Arithmetic	+, -, *, /, %, **
	Comparison	==, ===, !=, !==, >, <, >=, <=
	Logical	&&, ||
	Assignment	=, +=, -=, /=, %=, **=
	Bitwise	&, |, ^, <<, >>
3 (Ternary)	Conditional	Condition ? true : false

Working with non-Booleans values (Truthy v/s Falsy)

☑ Falsy → false, undefined, null, 0, -0, 0n, “”, NaN.

❌ Truthy → anything which is not falsy.

Short - Circuiting

Short-circuiting in JavaScript refers to the way logical operators (&&, ||, and ??) evaluate expressions. It allows you to control the flow and return values based on truthiness without writing full if statements.

1. || (Logical OR) : Returns the first truthy value or the last value if none are truthy. (Useful for setting default values.)

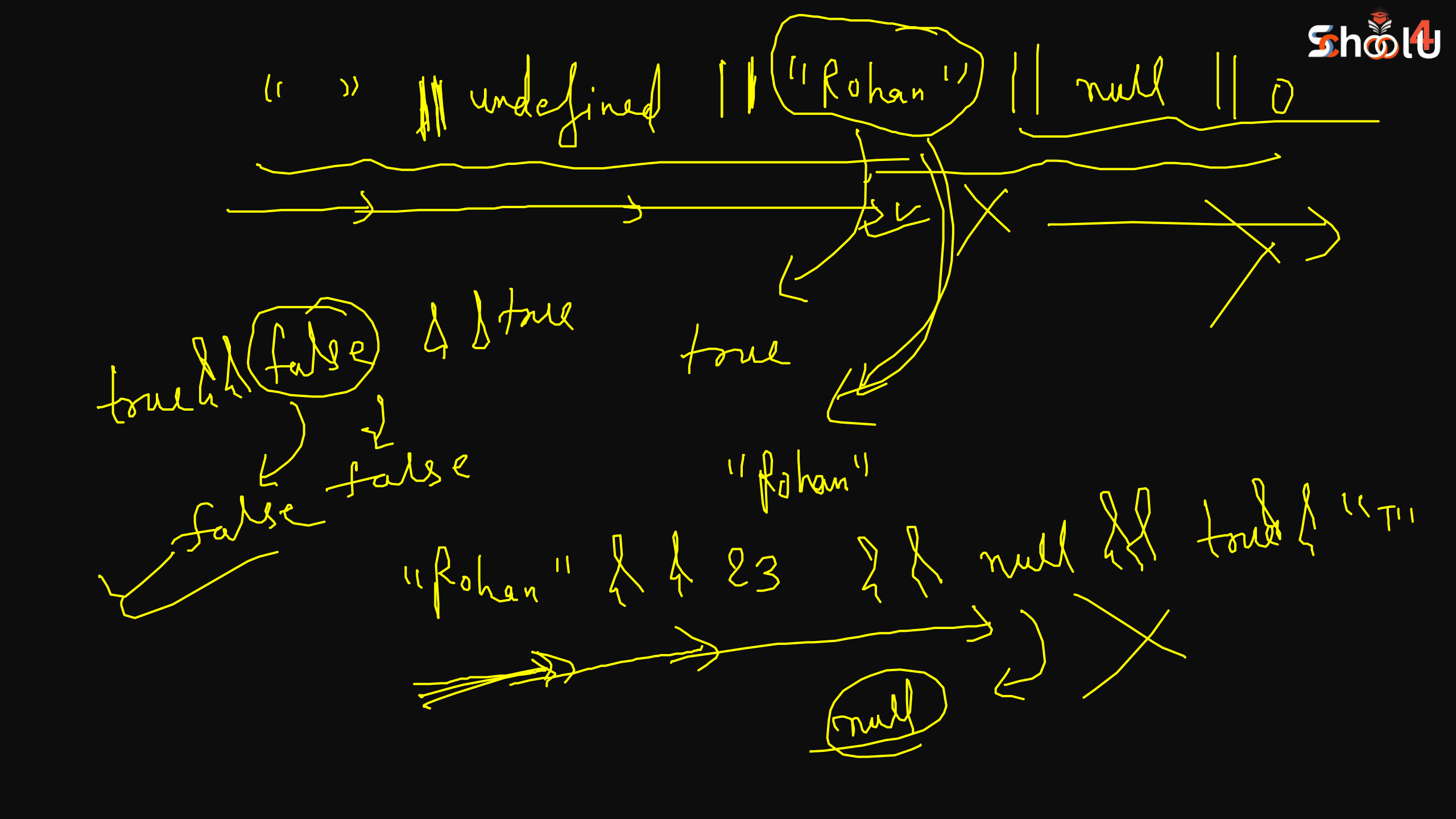
```
let result = "" || "Guest" || null || 23;
console.log(result); // "Guest"

let result2 = undefined || 0 || null;
console.log(result2); // null
```

2. && (Logical AND): Returns the first falsy value or the last value if none are falsy. (Commonly used to safely access properties.)

```
let result = "Ram" && true && undefined && 55;
console.log(result); // undefined

let isAuthenticated = true;
let user = "Manas Kumar Lal";
let result2 = isAuthenticated && user;
console.log(result2); // "Manas Kumar Lal"
```

3. ?? (Nullish Coalescing): Returns the right-hand value only if the left is null or undefined.
(Better than || when dealing with falsy values like 0 or "" that are still valid.)

```
let result = null ?? "Default";
console.log(result); // "Default"

let result2 = 0 ?? "MKL";
console.log(result2) // 0
```


"" || undefined || "Rohan" || null || 0
true && false
true && true
true
false
false
false
"Rohan"
"Rohan" && 23
null || true || "T"
null

1. Get user to input two number using prompt and print their possible arithmetic results.

2. Can you chain assignment operators?

3. Get user to input a number using prompt and check whether even or odd using ternary operator.

4. What is the final value of x?

```
let x = 5;
x += 3;
x -= 2;
x *= 4;
x /= 6;
x %= 3;
```

5. Check if a number is within a range between 10 and 20 (inclusive).

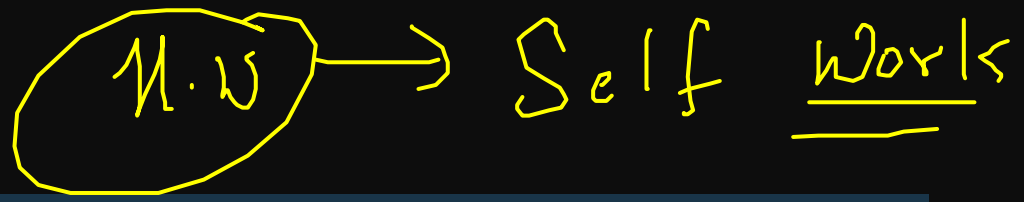
6. Write a program to find the largest number between 3 numbers using ternary operator.

7. Take an email and password from the user. If the email or password is incorrect or does not match the stored values, display the message "Invalid email or password." If both email and password match the stored values, display "You are logged in successfully!"

8. What will be the output of the following JavaScript code?

```
let a = 5, b = 3, c = 2;

let result = a++ + --b * c-- - ++a + b-- / --c;

console.log("a:", a);
console.log("b:", b);
console.log("c:", c);
console.log("result:", result);
```

9. What is the output of console.log(~a) where a = 0



9. What will be the output of the following JavaScript code?

```
let x = 10;
let y = 5;
let z = "10";

x += y * 2; // Arithmetic + Assignment
let isEqual = x == z; // Loose comparison
let isStrictEqual = x === z; // Strict comparison
let logicTest = (isEqual || isStrictEqual) && !(y > 10); // Logical + Comparison + NOT

let result = logicTest ? ++x : --y; // Ternary + Pre-increment/Pre-decrement

console.log("x:", x);
console.log("y:", y);
console.log("z:", z);
console.log("isEqual:", isEqual);
console.log("isStrictEqual:", isStrictEqual);
console.log("logicTest:", logicTest);
console.log("result:", result);
console.log("Type of z:", typeof z); // Unary operator typeof
```

10. What will be the output of the following JavaScript code?

```
let a = 6;
let b = 3;
let c = "6";

a += b << 1; // Bitwise left shift + assignment
let d = a & b; // Bitwise AND
let e = a | b; // Bitwise OR
let f = a ^ b; // Bitwise XOR
let g = ~a; // Bitwise NOT

let check = (a == c) && (d < e) || !(f === e); // Mixed logical, comparison

let result = check ? typeof g : --b; // Ternary + unary + typeof

console.log("a:", a);
console.log("b:", b);
console.log("c:", c);
console.log("d (a & b):", d);
console.log("e (a | b):", e);
console.log("f (a ^ b):", f);
console.log("g (~a):", g);
console.log("check:", check);
console.log("result:", result);
```

$\Rightarrow$ $a \mathbin{+=} b << 1$

$a += 6$

$\Rightarrow 12$

$\rightarrow$ Arithmetic

$\rightarrow$ $<<$, $>>$

$\rightarrow$ Assignment

$\Rightarrow a = a + b << 1$

$\Rightarrow a = 15$

Conditional Statement

Conditionals In JavaScript:

Use Case:

```
Scenario: On an e-commerce website, show a message when a product is out of stock.

Question:
Write a program to check if the stock of a product is 0. If it is, display the message "Product is out of stock.".
```

Solution:

```
let productStock = 0; // Number: Represents the stock of a product

if (productStock === 0) {
  console.log("Product is out of stock.");
} else {
  console.log("Product is in stock.");
}
```

A **Conditional statement** lets your code make **decisions**. It checks if something is **true or false**, and then runs certain code based on that.

```
if (condition) {
    // code to run if the condition is true
} else {
    // code to run if the condition is false
}
```

For Example:

"If it's raining, take an umbrella. Otherwise, enjoy the sunshine!"

```
let wheather = "rainy";

if (wheather === "rainy") {
    console.log("Take an umbrella");
} else {
    console.log("enjoy the sunshine!");
}
```

Different Types of Conditional Statements

1. if Statement
2. if...else Statement
3. if...else if...else (Also called "Else-If Ladder")
4. Nested if Statements
5. switch Statement
6. Ternary Operator [? :] (short form)

1. if Statement

```
if (temperature > 30) {
    console.log("It's hot outside!");
}
```

2. if...else Statement

```
if (age >= 18) {
    console.log("You can drive!");
} else {
    console.log("You can not drive!")
}
```

3. if...else if...else (Also called "Else-If Ladder")

```
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("You need to study more.");
}
```

4. Nested if Statements

```
if (age >= 18) {
    if (hasID) {
        console.log("You can enter the club.");
    } else {
        console.log("You need an ID.");
    }
} else {
    console.log("You're too young to enter.");
}
```

3. Switch Statement

```
switch (color) {
    case "red":
        console.log("Stop");
        break;
    case "yellow":
        console.log("Caution");
        break;
    case "green":
        console.log("Go");
        break;
    default:
        console.log("Unknown color");
}
```

4. Ternary Operator [? :] (short form)

```
let message = isLoggedIn ? "Welcome back!" : "Please log in.";
```

Challenge 1:

Scenario: A website gives discounts based on the total shopping cart amount.

Question:
Write a program where:

If the cart value is less than $50, no discount is applied.
If the cart value is between $50 and $100, apply a 10% discount.
If the cart value is more than $100, apply a 20% discount.
Display the final cart total after the discount.

Solution:

```
let cartValue = 120; // Example: Total shopping cart amount
let finalCartValue;

if (cartValue < 50) {
  finalCartValue = cartValue;
  console.log("No discount applied.");
} else if (cartValue >= 50 && cartValue <= 100) {
  finalCartValue = cartValue - (cartValue * 0.1); // Apply 10% discount
  console.log("A 10% discount has been applied.");
} else {
  finalCartValue = cartValue - (cartValue * 0.2); // Apply 20% discount
  console.log("A 20% discount has been applied.");
}

console.log(`The final cart total is ${finalCartValue.toFixed(2)}.`);
```

Challenge 2:

Scenario: On a video-streaming platform, verify user access to premium content.

Question:
Write a program to check if a user has a valid subscription. If the user has a subscription, further check if the subscription is "premium" or "standard".

If "premium", display "Access to all content".
If "standard", display "Access to limited content".
If the user doesn't have a subscription, display "Please subscribe to access content.".

Solution:

```
let hasSubscription = true; // Boolean: Does the user have a subscription?
let subscriptionType = "premium"; // String: Subscription type ("premium" or "standard")

if (hasSubscription) {
  if (subscriptionType === "premium") {
    console.log("Access to all content");
  } else if (subscriptionType === "standard") {
    console.log("Access to limited content");
  } else {
    console.log("Unknown subscription type");
  }
} else {
  console.log("Please subscribe to access content.");
}

```

Q.1- Give choice to the user to select theme color and set the selected theme color and console it.

Q.2- Find the smallest of three numbers. Numbers are given by the user.

Q.3- Write a program to manage Role-Based Access Control

Given a user role ("admin", "editor", "viewer"):

Admin: full access

Editor: edit access

Viewer: read-only

Any other: no access

Use switch.

Q.4- Check if Number is Divisible by 3 or 5 or Both. Print "Fizz" for multiples of 3,"Buzz" for multiples of 5, "FizzBuzz" for both.

Q.5- Create a simple calculator. Take two numbers and an operator (+, -, *, /) and calculate the result using switch.

Q.6- Create a simple ATM program.

User can choose:

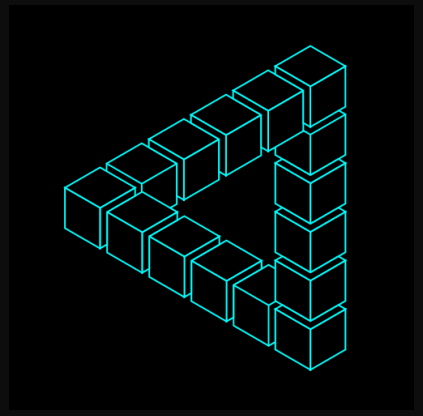
1. Check Balance
2. Deposit
3. Withdraw
4. Exit

Note that in case of “Deposit” if deposit amount is less than 1Rs produce error otherwise deposit the amount and show the message with a new balance. And in case of “Withdraw” if withdraw amount is greater than balance then or less than 1Rs then produce error otherwise withdraw amount and show remaining balance.

Loops

Loops In JavaScript:

Loops in JavaScript are a way to repeat a block of code multiple times.



Types of Loops in JavaScript :

1. for
2. while
3. do-while

Core Loops

4. for-of (arrays, strings, maps or sets [don't care about index])
5. for-in (iterating over the properties (keys) of an object or the indexes of an array)

for loop

```
Syntax:
for(initialization, condition, iterator){
    // task to do
}

Example:
for(let i=1; i<=10; i++){
    console.log("School4U");
}
```

while loop

```
Syntax:
while (condition){
    // task to do
}

Example:
let i=1;
while(i<=10){
    console.log("School4U");
    i++;
}
```

Note:

- ❑ Avoid infinite loop

do-while loop

```
Syntax:
do {
    // task to do
} while (condition)

Example:
let i = 1;
do {
    console.log("School4U");
    i++;
} while (i <= 10)
```

for-of loop

```
Syntax:
for(let val of something){
    // do some task
}

Example:
let str = "Manas Kumar Lal"
for(let val of str){
    console.log(val);
}
```

for-in loop

```
Syntax:
for (let key in something) {
    // do some task
}

Example:
let obj = {
    name: "manas",
    age: "21",
    isStudent: true,
}
for (let key in obj) {
    console.log(key)
}
```

Challenge 1:

Print all even numbers from 0 to 100.

Solution:

```
for (let i = 0; i <= 100; i++) {
    if (i % 2 !== 0) {
        console.log(i);
    }
}
```

Challenge 2:

calculate how many vowels and consonents are in a given string using for of loop

Solution:

```
let string = "hello";

let vowelCount = 0, consonentCount = 0;
for (let letter of string) {
    if(letter === 'a' || letter === 'e' || letter === 'i' || letter === 'o' || letter === 'u' || letter === 'A' || letter === 'E' || letter === 'I' || letter === 'O' || letter === 'U'){
        vowelCount++;
    }else{
        consonentCount++;
    }
}

console.log("Vowel = " + vowelCount);
console.log("Consonent = " + consonentCount);
```

1. Calculate sum of first ‘n’ numbers.
2. Calculate the sum of numbers from ‘m’ to ‘n’.
3. Print all odd numbers from 0 to ‘n’.
4. Create a “Number Knock” game. (ask the user to keep guessing the number until the user enters correct guess) .
5. Simple Password Checker (Fixed Attempts)
6. Create a program to find the factorial of ‘n’.
7. Print the following pattern. (build it for nth numbers)

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Build a Simple Text-Based Adventure Game

```
"You wake up in a dark forest..."
"Do you go 'left' or 'right'?"

"after choosing left:"

"You walk into a swamp..."
"You see something shiny in the mud. Do you pick it up? (yes or no)"
"if yes:" "It's a magic stone! You are teleported to safety. You win!"
"if no":" "You sink slowly into the mud. Game over."

"after choosing right:"

"You find a cave..."
"Do you enter the cave? (yes or no)"
"if yes:" "A dragon wakes up and chases you away. You barely escape!"
"if no:" "You set up camp outside the cave. A peaceful night under the stars. You win!"

"Do you want to play again? (yes or no)"
```

Functions

Functions In JavaScript:

A function is a block of code designed to perform a particular task.It only runs when called/invoked.

Or

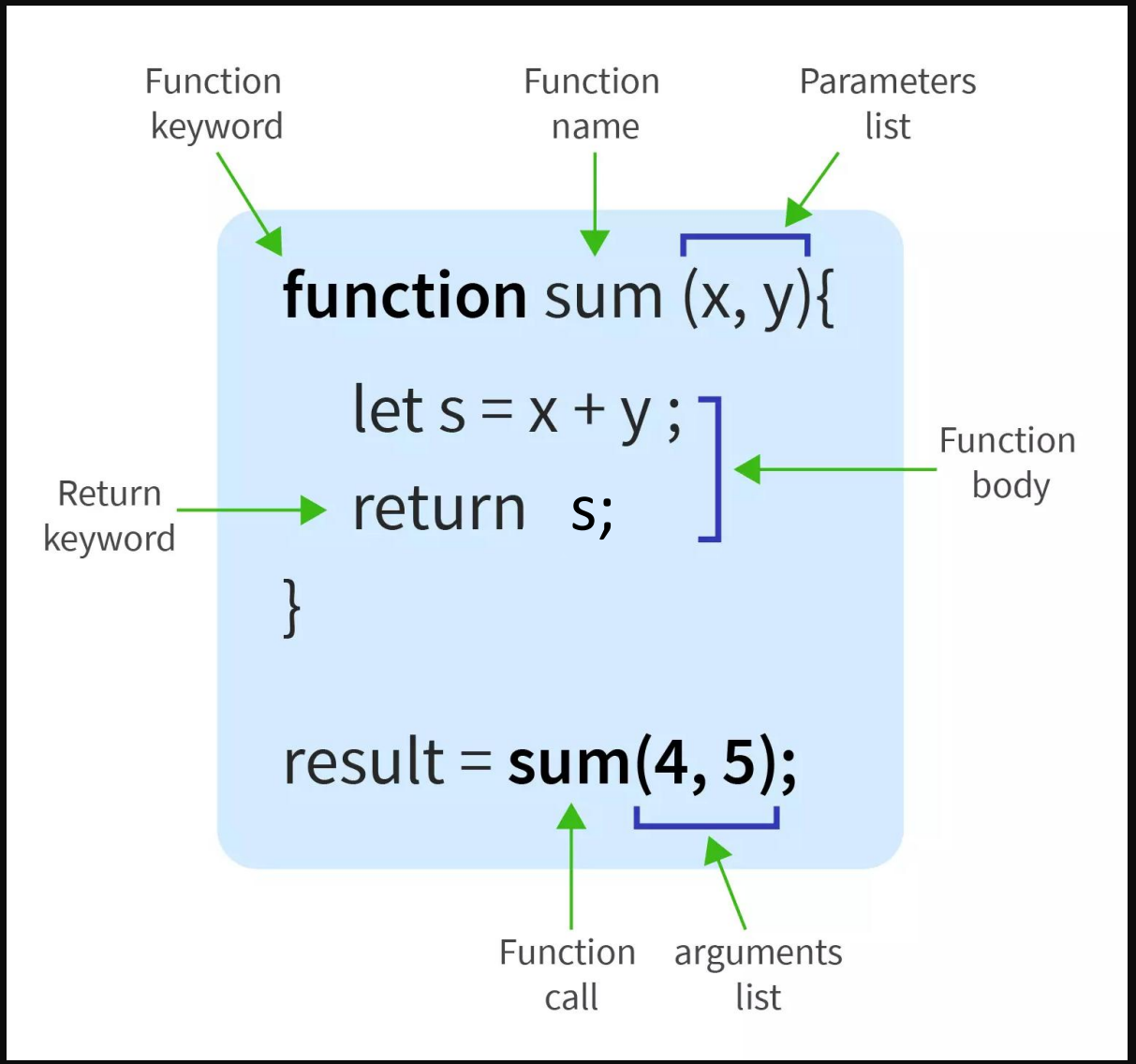
A function in JavaScript is like a reusable set of instructions

Function Definition

```
function functionName(){
    // do some task
}
```

Function Call/invoke

```
functionName();
```



4 ways to create a function

1. Without parameter list & without return.
2. Without parameter list & with return.
3. With parameter list & without return.
4. With parameter list & with return.

Note:

- Parameters: placeholders used when creating the function.
- Arguments: actual values passed when calling the function.
- Parameters in a function is like local variable having block scope.

Default Parameters

default parameters allow you to assign default values to function parameters.

If no value (or undefined) is passed for that parameter when the function is called, the default value is used instead.

Example:

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

greet("MKL"); // Output: Hello, MKL!
greet(); // Output: Hello, Guest!
```

Note:

- ❑ Only undefined triggers the default value.
- ❑ You can use expressions as default values.

Types of functions

#	Function Type	Description	Example
1	**Function Declaration** (normal function)	Declared with the function keyword, can be hoisted	`function greet(){ console.log("hello") }`
2	**Function Expression**	Stored in a variable; not hoisted	`const greet = function(){ console.log("hello") }`
3	**Arrow Function** (fat arrow function)	Shorter syntax, introduced in ES6	`() => { console.log("hello") }`
4	**Anonymous Function**	A function without a name, often used in expressions	`setTimeout(function () { console.log("hello") }, 2000);`
5	**IIFE** (Immediately Invoked Function Expression)	A function that runs as soon as it's defined	`(function(){ console.log("hello") })()`

1. Write a regular function that takes a string and returns it with the first letter capitalized.

2. Show an alert message that says “Please login” after 5 seconds on your website.

3. Make an arrow function that takes a price and a discount, and returns the price after discount.

4. Create a function that builds a username from a full name.

5. Write a function that takes a traffic light color and gives the correct instruction (e.g. "go" for green, "stop" for red, "caution" for yellow, and "invalid color" for anything else).

STRINGS

String In JavaScript:

A string is a piece of text.

It can be a word, a sentence, a single letter, or even empty—anything made of characters.

In JavaScript, we write a string by putting text inside quotes:

```
"School4U"
'JavaScript'
"123"
""
```

You can use double quotes (" "), single quotes (' '), or backticks ( ).

```
let greeting = "Hello, world!";
let name = 'Ali';
let empty = "";
let sentence = `My name is ${name}`;
```

Placeholders

Template Literals

A template literal is a special way to write strings in JavaScript using backticks (`).

String literals let you insert variables or expressions directly inside the string, which is called string interpolation.

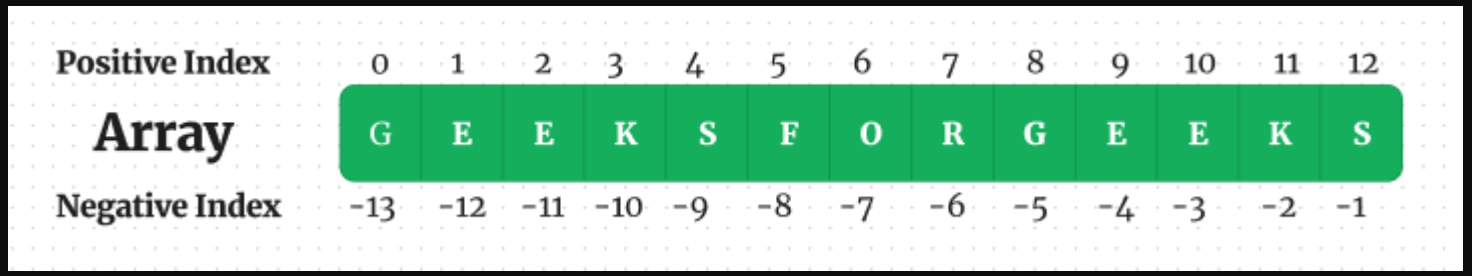
Create string using string constructor:

```
let str = new String("Manas Kumar");
```

Note:

- 🔒 Strings are immutable in JavaScript.

String has index (position)



Escape Sequences

An escape sequence is a special character combo that lets you do things in a string that aren't normally allowed—like adding quotes, new lines, or tabs. (e.g. \n, \t, \\, \", \', etc)

Escape sequences look like 2 characters in code,but they count as 1 character in string length.

String Properties & Methods in JavaScript

Properties:

- str.length – tells how many characters are in a string. (length of string)

Methods:

- str.toUpperCase() – convert each letter into uppercase
- str.toLowerCase() – convert each letter into lowercase
- str.trim() – remove whitespaces
- str.concat(str2) – joins str2 with str1
- str.includes() – checks if string contains given piece of string
- str.indexOf() – gives the position (index) of the first match
- str.charAt() – gives the character at given index
- str.replace(old, new) – replaces first matched part of the string with something else
- str.replaceAll(old, new) – replaces all matched parts of the string with something else
- str.slice(start, end) – cuts out a piece of the string
- str.split() – the .split() method is used to break a string into parts and turn it into an array.

1. Create a program to take full name from user and generate a username start with @, followed by their full name and ends with underscore followed by the length of full name.

2. Take a string and a character from the user and:
    a) count how many times that character appears in the string.
    b) Case-Insensitive Version
    c) Find All Occurrence Positions

3. Count the words present in a given string.

ARRAY

Array In JavaScript:

An array is a list that can store multiple values in one place.
Think of it like a row of boxes, where each box can hold one item (like a number, word, etc.).
You can use it to group similar things together.

Create array using array Literal method (recommended):

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

let marks = [23, 94, 55, 26, 84, 89];

let personDetails = ["Muskan", 18, "Bhagalpur"];
```

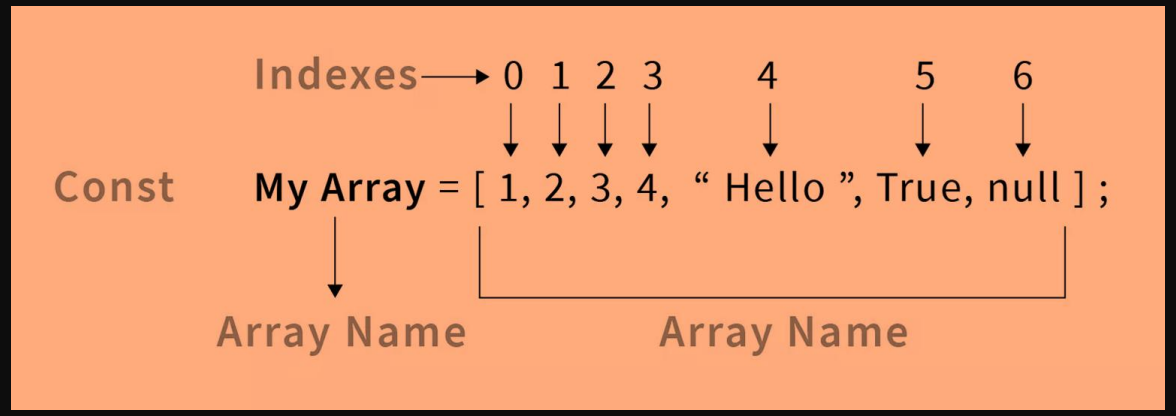
Create array using array constructor:

```
let arr = new Array("spiderman", "ironman", "thor");
```

Note:

- ❑ Array constructor is mostly used when we want to create an empty array with the given length

Indexing in Array



Arrays are mutable

change

Original

Note:

- typeof array is not "Array", it's an "Object".
- Array is a special type of object
- Arrays are mutable.

Array Methods

Method	Changes the Original Array?	What It Returns	What It Does
push()	Yes	New length of the array	Add item at the end
pop()	Yes	The removed last item	Delete item from end
unshift()	Yes	New length of the array	Add item to start
shift()	Yes	The removed first item	Delete item from start
splice()	Yes	An array of removed items	Add remove or doing both at the same time
slice()	No	A new array (sliced portion)	Returns a piece of array
concat()	No	A new array (combined arrays)	Joins multiple arrays (we can also use spread operator instead)
join()	No	A string made by joining elements	join the elements of array on the basis of some string or character
toString()	No	A string of array elements	Converts array to string
includes()	No	true or false	Check whether given item is present in array or not
indexOf()	No	Index of the item or -1 if not found	returns the index of given item if it is present in array, otherwise returns -1
reverse()	Yes	The reversed array	Reverse the order of items
sort()	Yes	The sorted array	Sort the array
find()	No	The first matching item (or undefined)	Returns the first element in the array that satisfies a condition
flat()	No	A new array with nested arrays flattened	returns a new array with nested arrays flattened

Sort method

```
arr.sort(compareFunction)
```

It works on Tim sort under the hood. (combination of merge sort and insertion sort)

By default, sort() converts elements to strings and sorts them alphabetically (Unicode order) or Lexicographical order.

```
const nums = [10, 5, 20];
nums.sort();  // ❌ [10, 20, 5] – because it sorts as ["10", "20", "5"]
```

So, numbers don’t always sort correctly without a compare function.

Return Value of the function	Meaning	Effect on Order
▽ **Negative** (< 0)	a should come **before** b	☑ Keep order as-is
⍰ **Zero** (=== 0)	a and b are **equal**	🤝 Keep their order (stable)
△ **Positive** (> 0)	a should come **after** b	🔁 Swap a and b

Array iteration methods or Functional array methods

Method	Returns	Modifies Original Array?
map()	New array	No
filter()	New array	No
reduce()	Single value	No
forEach()	undefined	No (unless you do it manually)

So if you just want to loop through an array and do something, forEach() is great. But if you want to create a new array or value, use map(), filter(), or reduce().

MDN Web Docs

- **JavaScript** array-copy operations **create** shallow copies. (All standard built-in copy operations with *any* JavaScript objects create shallow copies, rather than deep copies).

A **shallow copy** of an object is a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made. As a result, when you change either the source or the copy, you may also cause the other object to change too. That behavior contrasts with the behavior of a deep copy, in which the source and copy are completely independent.

String v/s Array

Feature	String	Array
Purpose	Stores **text** (a sequence of characters)	Stores **multiple values** (any data type)
Storage	Text only (characters)	Can be numbers, strings, booleans, etc.
Example	"hello"	["h", "e", "l", "l", "o"]
Indexing	Yes (e.g. "hello"[0] is 'h')	Yes (e.g. ["red", "blue"][0] is "red")
Mutable?	**No** (characters can't be changed directly)	**Yes** (you can change items easily)
Length Property	Yes: "hello".length → 5	Yes: ["a", "b"].length → 2
Common Use	Names, messages, text	Lists of things (colors, numbers, etc.)
Datatype	String	Object
Iterable	**Yes**	**Yes**

1. For an array with marks of students find the average marks of the entire class.

2. Create an array with the given length(n) and fill with 0.

3. Create an array with the given length (n) and store natural numbers from 1 to n.

4. Consider an array of mcu heroes ([ironman, captain, black widow, wanda, hulk, black panther]). Now:

   a) Add spiderman at the end and thor at the start.

   b) Remove black widow and add hawkeye in its place.

   c) Check whether captain is present in the array.

5. How to check if given thing is array or not? How to convert other datatypes to array? What if we try to convert an object into an array?

6. We have three variables a, b, c, a contains any number, b contains any string, c contains any object, and d contains any array. Can we create an array from all these four variables? If yes, How?

7. Check whether given string is palindrome or not.

8. Capitalize the first letter of every word in a sentence.

OBJECT

Object In JavaScript:

An object is a collection of key-value pairs or stores related information as a set of key-value pairs. It's a way to group data and functions together.

Example: Think of a real-life object “car”:

Create an object using literal syntax:

```
let car = {
    brand: "Toyota",
    color: "red",
    speed: 120,
    drive: function () {
        console.log("The car is driving");
    }
};
```

Properties (brand, color, speed)

Behaviors / Methods (drive, stop)

Create an object using the Object constructor:

```
let person = new Object();
person.name = "Alice";
person.age = 25;
```

Note:

- ❑ Objects are mutable.

Accessing Object Properties

You can access properties in two ways:

1. Dot Notation (Most Common)

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

2. Bracket Notation (Useful with variables or special characters or strings with white spaces)

```
console.log(person["full name"]);
console.log(obj["*"]);
console.log(obj.variableName);
```

Updating or Adding New Properties

Update if already existing property and add if not exist.

```
person.city = "New York";
person["hobby"] = "Reading";
```

Deleting Properties

```
delete person.isStudent;
```

this Keyword in object

In case of object, this refers to the object itself.

```
let obj = {
    name: "Manas Kumar Lal",
    greet : function(){
        console.log(this.name)
    }
}
obj.greet() // Manas Kumar Lal
```

Object Methods

Method	Changes Original Object?	What It Returns	What It Does
Object.keys(obj)	❌ No	Array of keys	Returns an array of all **enumerable property names (keys)**
Object.values(obj)	❌ No	Array of values	Returns an array of all **enumerable property values**
Object.entries(obj)	❌ No	Array of [key, value]	Returns an array of **[key, value]** pairs
Object.assign(target, source)	☑ Yes (if target is modified)	Modified target object	Copies properties from source to target and returns the updated object
Object.freeze(obj)	☑ Yes (locks the object)	The frozen object	Prevents any changes (no adding, removing, or modifying properties)
Object.seal(obj)	☑ Yes (locks structure)	The sealed object	Prevents adding/removing properties, but allows modifying existing ones
ObjName.hasOwnProperty(key)	❌ No	Boolean (true / false)	Checks if the object has the specified property directly

What is singleton object in js?

A singleton is just a single, unique object created once in your code.

Creating an object using literal syntax is considered a singleton because it creates one specific object instance that is not used as a template to create others.

Feature	Singleton Object	Class-Based Object
Syntax	Object literal {}	class or function
Purpose	One specific object	Blueprint for many objects
Instances	Only one	Many objects can be created
Example Use Case	Settings, config, logger	Users, cars, products
Example Code	let config = {}	class User { ... }
Feature	Singleton Object	Class-Based Object

Destructuring

```
const person = {
    name: 'Alice',
    age: 25,
    city: 'New York'
};

const { name, age } = perso
n;
console.log(name); // "Alic
console.log(age);  // 25
```

```
const colors = ['red', 'green', 'blu
e'];
const [first, second, third] = colors;

console.log(first);  // "red"
console.log(second); // "green"
console.log(third);  // "blue"
```

Object v/s Array

Feature	Object	Array
Purpose	Store data as **key-value pairs**	Store **ordered list** of values
Syntax	{ key: value }	[value1, value2, value3]
Access by	Keys (names)	Index (number, starts from 0)
Example	{ name: 'Alice', age: 25 }	['Alice', 25]
Ordering	Not guaranteed	Preserves order
Best Use Case	Describing **properties of a thing**	Working with a **list of items**
Type Check	typeof obj === 'object' && !Array.isArray(obj)	Array.isArray(arr)
Iterable	❌ No	☑ Yes
Iteration	for...in, Object.entries()	for...of, forEach(), map()
Can be nested?	Yes (objects inside objects)	Yes (arrays or objects inside arrays)
Common methods	Object.keys(), Object.values()	push(), pop(), map(), filter()

1. Create a person object with properties: name, age, and city. Then
   a) Log each property as: value of “name” property is “manas” using loop.
   b) Add a new property called email to the person object.
   c) Delete “city” property from person object.

2. Create a function that takes an object with firstName, middleName, lastName properties and returns the full name.

3. Write a function that takes object and returns the number of properties in an object.

4. Write a function that returns an array of names of users who have the role “admin”.

```
const users = [
    { name: "Alice", role: "admin" },
    { name: "Bob", role: "user" },
    { name: "Charlie", role: "admin" }
];
```

5. Write a function searchProducts(products, keyword) that returns an array of products whose name includes the given keyword (case-insensitive).

```
const products = [
    { id: 1, name: "iPhone 14" },
    { id: 2, name: "Samsung Galaxy" },
    { id: 3, name: "Google Pixel" }
];
```

6. Write a function groupByPost(comments) that returns an object grouping comments by postId:

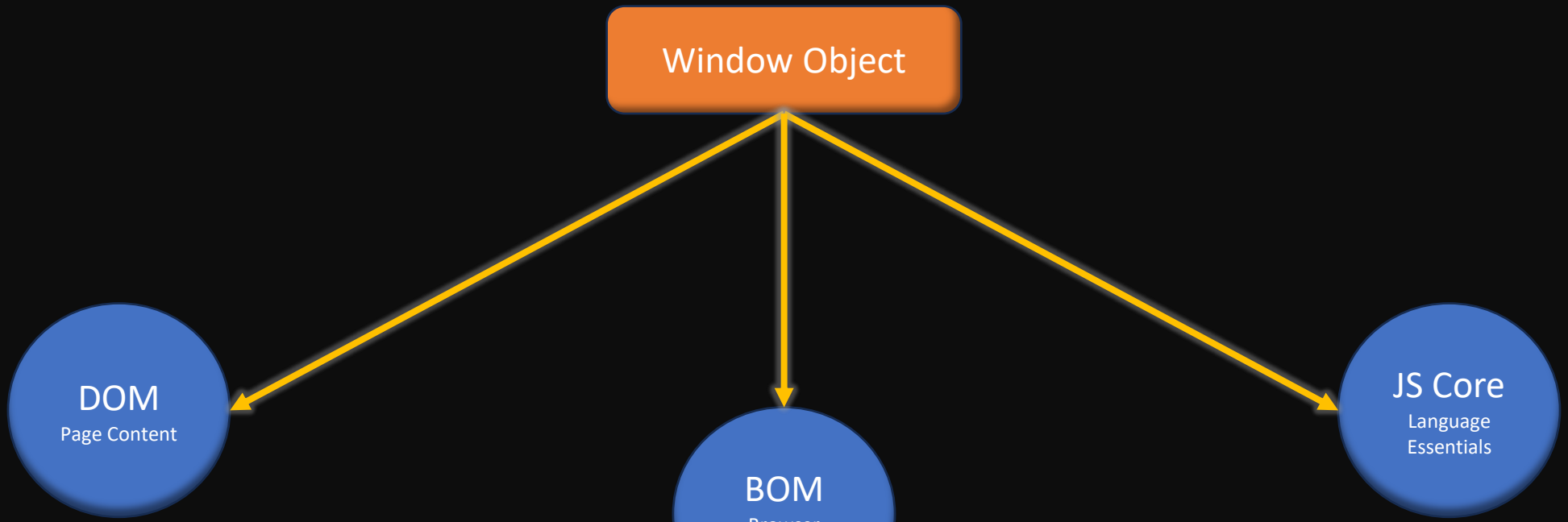
```
const comments = [
    { postId: 1, text: "Great post!" },
    { postId: 2, text: "Thanks!" },
    { postId: 1, text: "Very helpful" }
];
```

```
{
    1 : ["Great post!", "Very helpful"],
    2 : ["Thanks!"]
}
```

7. Write a function buildQuery(params) that returns

```
const params = { search: "laptop", page: 2, sort: "price" };

// Output -> "search=laptop&page=2&sort=price"
```

Dom Manipulation



- Changing page content (text, HTML)
- Changing element styles (colors, fonts, sizes)
- Handling user events (clicks, typing, scrolling)

- Alerts
- Confirmations
- Prompts
- Browser window size
- URL, history navigation
- Timers

- Variables and data types
- Operators
- Functions
- Objects and arrays

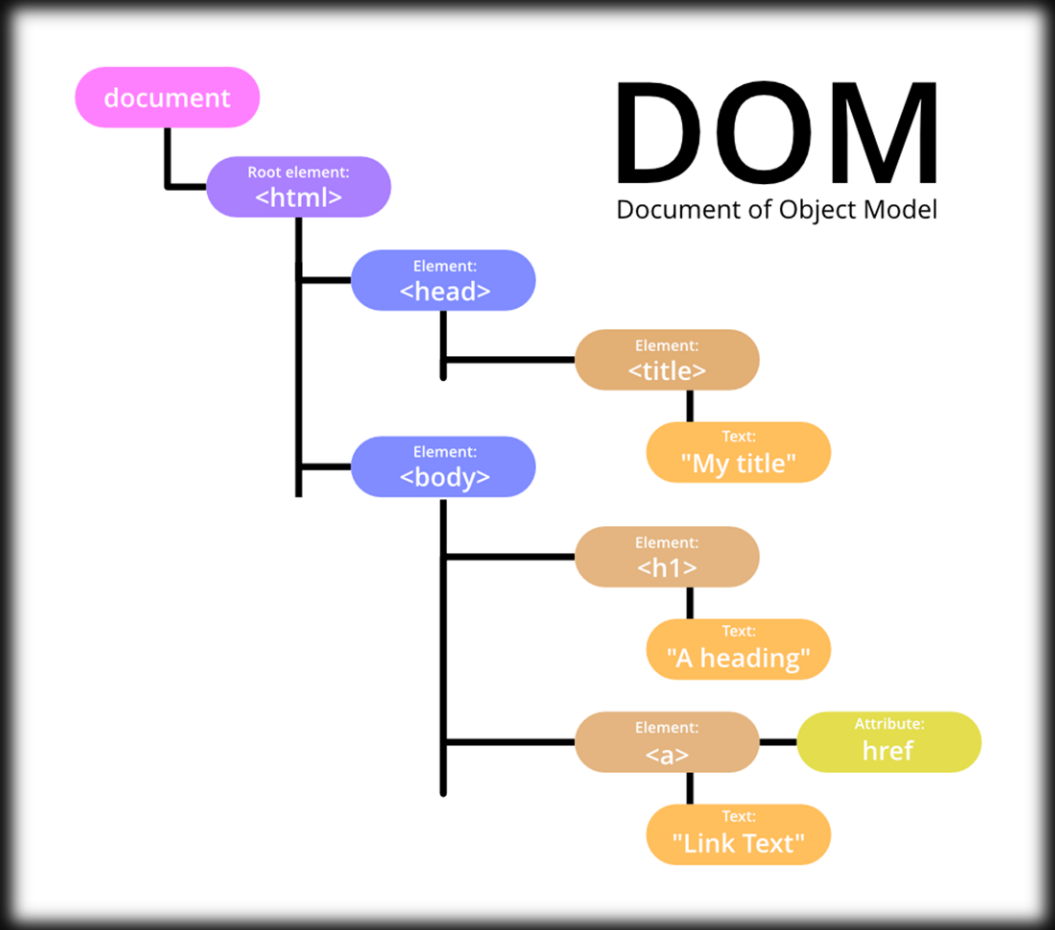
5 Phase Or Pillars Of Dom Manipulation:

1. DOM (Document Object Model)
2. Selection Of HTML
3. HTML Manipulation (Changin the HTML)
4. CSS Manipulation (Changin the CSS)
5. Event Listeners (Event Handling)

DOM
(Document Object Model)

What is DOM?

When a web page is loaded, browser creates a document object model (DOM) of the page.



Let's take a deep dive:

- The HTML page gets converted into the DOM (a big JavaScript-like object structure named “Document”).
- Because of that, everything on the page — buttons, headings, images — becomes an object with properties and methods.
- That's why you can do things like:
  - querySelector("h1") ➔ to find an element.
  - addEventListener("click", ...) ➔ to listen for user actions.
  - innerHTML = “Mai hun manas" ➔ to change content.

Note1:

- Document object is available inside window object.

Note2:

- If your <script> is in the <head>, the browser runs the JavaScript before it finishes loading the <body>. In that case, document Elements you are trying to access are null.

Without the DOM, JavaScript wouldn't know what’s on the page, and we couldn’t control it so easily.

In short:

HTML ➔ DOM ➔ JavaScript controls the DOM like objects.

What Is NodeList?

A NodeList is a simple list of DOM elements or collection of nodes (DOM elements) you got from the webpage, but it’s not a full real array.

Feature	NodeList	Array
Definition	Collection of nodes (DOM elements)	List of any data types (numbers, strings, objects, etc.)
Creation	Returned by DOM methods like querySelectorAll()	Created with [] or Array()
Type	object (specifically a NodeList)	object (specifically an Array)
Indexable	Yes (like nodelist[0])	Yes (like array[0])
Length Property	Yes	Yes
Loopable	Yes (with for, sometimes forEach)	Yes (with for, forEach, etc.)
Array Methods	No (older NodeLists) or **some** (modern NodeLists have forEach)	Yes (full support: map, filter, reduce, etc.)
Static/Dynamic	Usually static (doesn’t auto-update if DOM changes)	Static (unless manually updated)
Convert to Array	Needed if you want full Array methods (Array.from(nodelist) or [...nodelist])	Already an Array

Selection

Selection:

Selecting with tag:

```
document.getElementsByTagName("h1")
```

Selecting with id:

```
document.getElementById("id")
```

Selecting with class:

```
document.getElementsByClassName("class")
```

Query selector:

```
document.querySelector("id/class/tag")
```

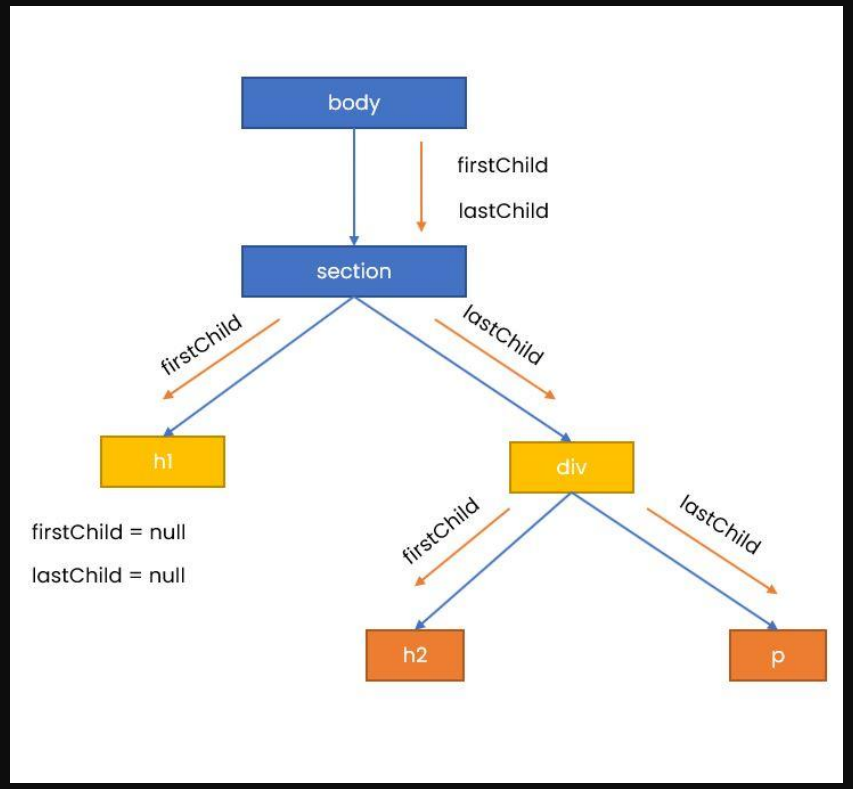
It returns only first matched element.

```
document.querySelectorAll("id/class/tag")
```

It returns all matched elements in the form of NodeList.

Note:

- For id use '#'
- For class use '.'



Do You Know About $0 Magic

Manipulating The HTML

Changing the HTML (Manipulating the HTML):

Properties:

tagName (read only): Returns the tag name (like "DIV", "H1") of the element in UPPERCASE. (include hidden elements)

nodeName (read only): Returns the name of the node in UPPERCASE for element nodes (e.g., "DIV", "SPAN") and special strings for other types (e.g., "#text" for text nodes, "#comment" for comment nodes).(includes hidden elements and works on any node type, not just HTML elements)

innerText: Returns all the text of the element and its children. (It respects styles like display: none or visibility: hidden, so hidden elements are ignored.)

innerHTML: Returns the HTML content (including any tags inside) as a string. (include hidden elements)

textContent: Returns all the text of the element and its children (include hidden elements).

Property	Includes hidden elements?	Includes HTML tags?
innerText	❌ No (ignores hidden text)	❌ No
innerHTML	☑ Yes	☑ Yes (returns tags too)
textContent	☑ Yes	❌ No

Note:

- We can also set the html using these properties

Insert Elements (Addition Of Elements):

Step1: Creation of element

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

Step2: Creation of element

node.append(elem) : adds at the end of node (inside)

node.prepend(elem) : adds at the start of node (inside)

node.after(elem) : adds after the node (outside)

node.before(elem) : adds before the node (outside)

node.inserAdjacentElement(position, elem) : positon can be “beforebegin”, “afterbegin”, “beforeend”, “afterend”

Deletion of Elements:

node.remove()

Parent Node

node.parentNode: Returns the immediate parent of a node (could be an Element, Document, or DocumentFragment).

Common Parent Methods

replaceChild(newChild, oldChild): Replaces an existing child node with a new one.

appendChild(child): Adds a child node to the end of the parent's children list.

insertBefore(newNode, referenceNode): Inserts a new node before a specified existing child node.

removeChild(child): Removes a specified child node from the parent.

Note:

- sometimes you see appendChild and removeChild in older code. (only work with nodes and not with strings like text).

Attributes:

getAttribute(attr): To get the attribute value

setAttribute(attr, value): To set the attribute value

Example:

```
let node = document.querySelector('#btn');

let beforeStyle = node.getAttribute('style');
console.log(beforeStyle); // color: blue;

node.setAttribute('style', 'background-color: red');

let afterStyle = node.getAttribute('style');
console.log(afterStyle); // background-color: red;
```

1. Create a paragraph with text “mai tumse pyar nahi karta hun” and add background color to black and font color to green using javascript.

2. Insert a button with text “click me” as the first element inside the paragraph created in 1st question.

3. Create a tag in html and give it a class & some styling. Now create a new class in css and try to append this class to the element.

Challenge 1:

Create a function that takes node and newTagName and replace the node from the new node with changed tag name in the DOM.

```
function replaceTag(node, newTagName) {
    if (!node || !(node instanceof Element)) {
        console.error('Invalid node provided');
        return null;
    }

    const newNode = document.createElement(newTagName);

    // Copy attributes
    for (const attr of node.attributes) {
        newNode.setAttribute(attr.name, attr.value);
    }

    // Copy the inside content
    // newNode.innerHTML = node.innerHTML;

    // or

    // Move all child nodes at once
    newNode.append(...node.childNodes); // Spread all childNodes into append()

    // Replace node
    node.parentNode.replaceChild(newNode, node);

    return newNode;
}

let btn = document.querySelector('#btn');
replaceTag(btn, 'div');
```

Manipulating The CSS

Changing the Style (Manipulating the CSS):

1. Using attribute method

```
node.setAttribute('style', 'background-color: red');
node.setAttribute('class', 'darkMode')
```

2. Using style

```
node.style.backgroundColor = "red";
node.style.fontSize = "80px";
```

3. Using cssText property

```
node.style.cssText = 'background-color: black; color: white; font-size: 80px'
```

4. Using className property

```
node.className = 'lightMode'
```

5. Using clasList property : add(), remove(), toggle(), contains()

```
node.classList.add('class2');
node.classList.remove('class1');
node.classList.toggle('class2');
node.classList.contains('class1');
```

1. Create a simple website with theme changing functionality.

2. Solve first question by changing css class.

3. Solve the first question by toggle css class.

Event Handling

Event Handling:

An event in JavaScript is something that happens in the browser, like a user clicking a button or moving the mouse.

Example:

Mouse events (click, hover, double click, etc.)

Keyboard events (keypress, keyup, keydown, etc.)

Form events (submit, focus, input, etc.)

Note:

- You can monitor events using “monitorEvents(document)” and unmonitor using “unmonitorEvents(document)”.

We can handle event using following three methods:

1. Inline method (inline javascript)
2. Property method
3. Listener method

property-based event handling" or **"direct event assignment"**

```
node.event = () =>{
    // task
}
```

Note:

- ❑ you can't add multiple handlers for the same event on the same element using property method.

"event listener method" or **"modern event handling"**

```
div.addEventListener(event, () => {
    // task
})
```

How to remove event listeners:

If you want to properly add and then remove an event listener, you must:

1. Use addEventListener().

2. Use a named function, not an anonymous arrow function.

```
let random = () => {
    console.log("alpha")
}

div.addEventListener('click', random)

div.removeEventListener('click', random)
```

Event Object:

The event object is an object that is automatically passed to the event handler function when an event occurs.

It contains important information about the event, such as what triggered the event, the type of event, and other details like the mouse coordinates, key pressed, and more.

```
node.onclick = (eventObj) => {
    console.log(eventObj)
}
```

```
node.addEventListener('click',(eventObj)=>{
    console.log(eventObj)
})
```

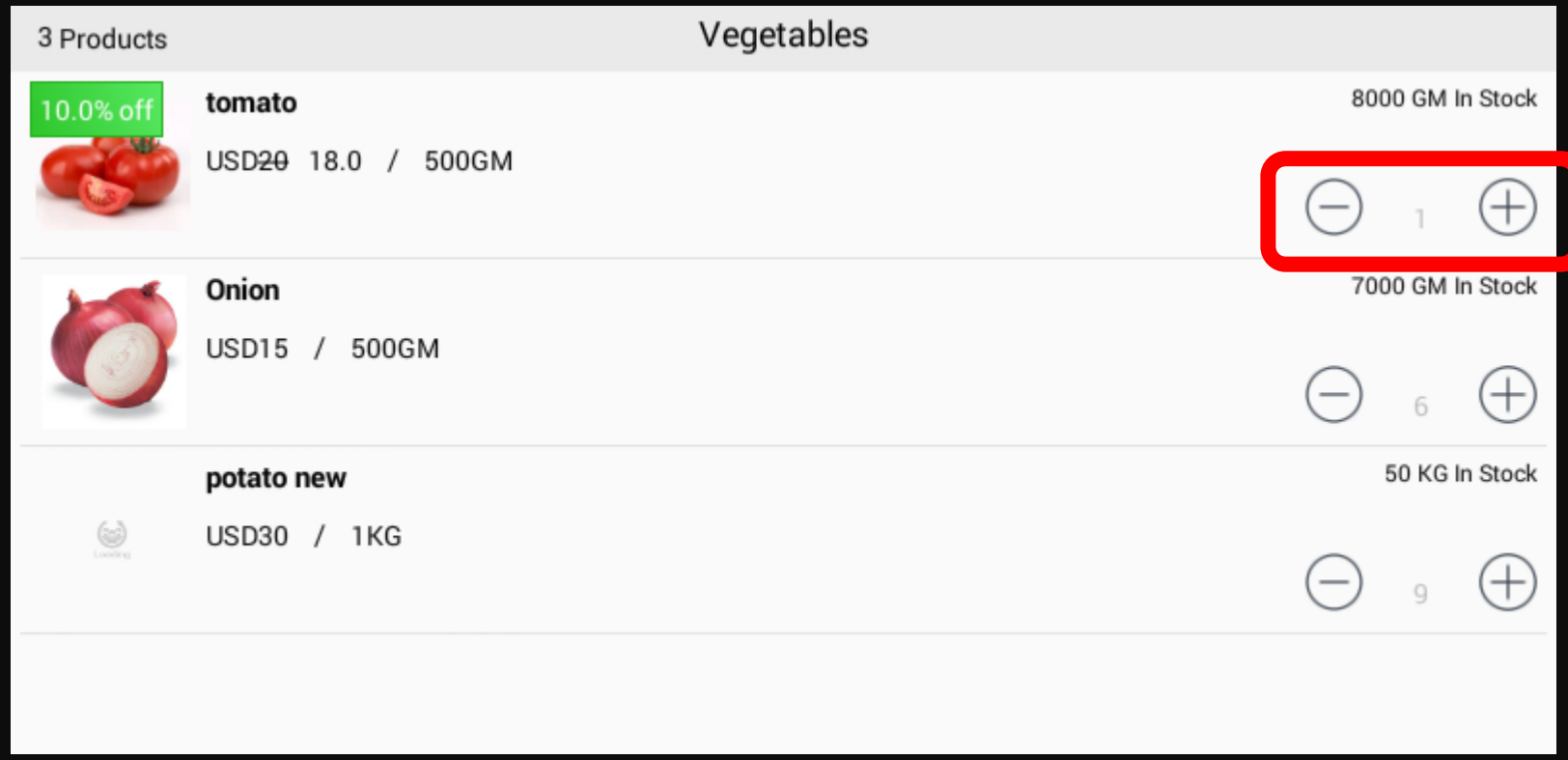
Most Used Events:

Category	Event	Role / Description
Mouse Events	click	Fired when an element is clicked.
	dblclick	Triggered by a double click.
	mousedown	When mouse button is pressed.
	mouseup	When mouse button is released.
	mouseenter	Mouse enters an element (no bubbling).
	mouseleave	Mouse leaves an element (no bubbling).
	mouseover	Mouse moves over an element or its children.
	mouseout	Mouse leaves an element or its children.
	mousemove	Mouse is moved within an element.
	contextmenu	Right mouse button is clicked.
Keyboard Events	keydown	Key is pressed down.
	keypress	(Deprecated) Key that produces a character is pressed.
	keyup	Key is released.

Category	Event	Role / Description
Form Events	submit	Form is submitted.
	reset	Form is reset.
	focus	Element receives focus.
	blur	Element loses focus.
	input	Value changes (real-time).
	change	Value of form element changes (on blur).
Touch Events	touchstart	Finger touches the screen.
	touchmove	Finger moves on screen.
	touchend	Finger is removed from screen.
	touchcancel	Touch interrupted (e.g., alert or system event).

Category	Event	Role / Description
Window Events	load	Page and resources fully loaded.
	DOMContentLoaded	DOM is fully loaded (without waiting for styles/images).
	resize	Window is resized.
	scroll	Page is scrolled.
Clipboard Events	copy	Content is copied.
	cut	Content is cut.
	paste	Content is pasted.

1. Build an increment–decrement counter similar to what you see in the shopping cart on Amazon or Flipkart.



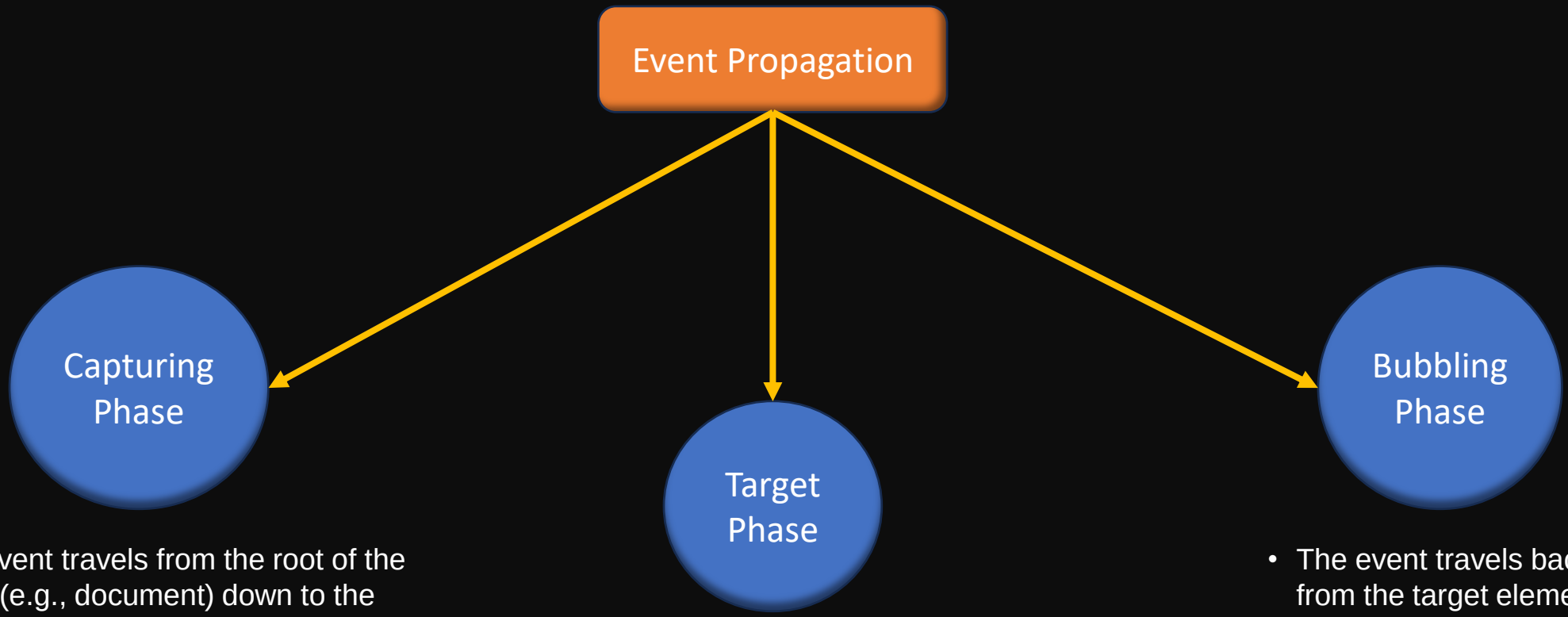
2. Create a simple form and display the submitted details on the webpage. Ensure that if any field is left empty, the form should not be submitted.

Projects

(Next Lecture)

Event Lifecycle

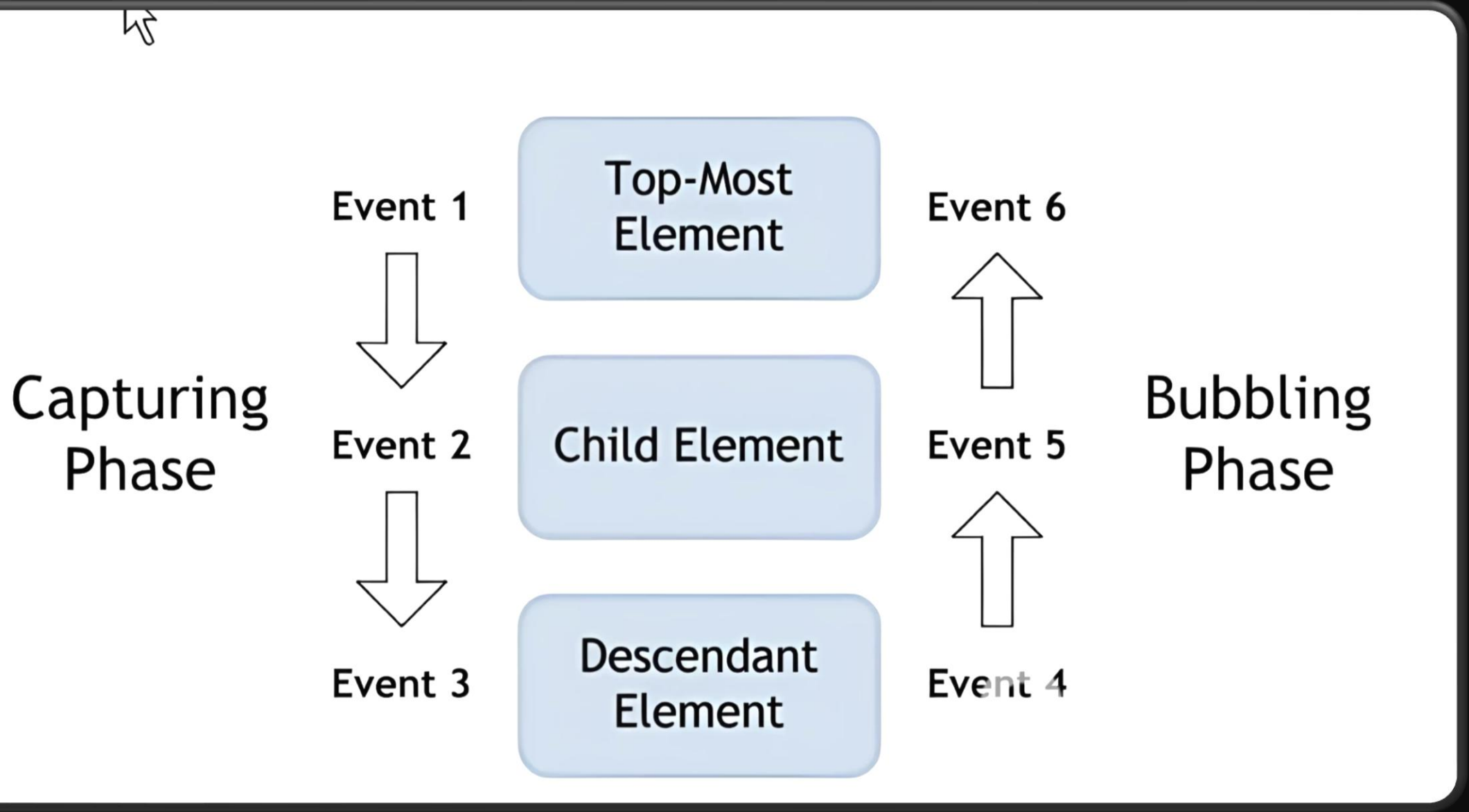
Event propagation describes the way events travel through the DOM after they are triggered. It consists of three phases:



- The event travels from the root of the DOM (e.g., document) down to the target element.
- This is also called the "trickling phase."

- The event reaches the target element where the event occurred.

- The event travels back up from the target element to the root of the DOM.


Capturing Phase
Event 1
Event 2
Event 3
Top-Most Element
Child Element
Descendant Element
Event 6
Event 5
Event 4
Bubbling Phase

event.target v/s event.currentTarget v/s this

"event.target" : Identifying which child element was clicked or interacted with.

"event.currentTarget" : Always points to the element where the listener is attached, regardless of where the event originated or which element triggered it.

"this": Refers to the context in which the function is executed. Inside an event listener, this points to the element to which the event listener was attached (the element where addEventListener was used).

How To Stop Bubbling & Capturing?

1. stopPropagation()

2. stopImmediatePropagation()

What is Event Delegation?

Event delegation is a technique in JavaScript where you add an event listener to a parent element instead of adding it to multiple child elements. The parent "delegates" the event handling to its child elements using the bubbling phase of event propagation.

This technique is useful when:

1. You have a large number of child elements that need the same event handler.
2. Child elements are dynamically added or removed.

Number, Math & Date

Number

Numbers In JavaScript

In JavaScript, numbers represent both integer and floating-point values.

Unlike many other languages, JavaScript has only one number type: Number. No separate types for integers or floats.

Number is technically a constructor function (also called a built-in object) in JavaScript

```
let age = 25;           // Integer
let price = 19.99;      // Floating-point
let negative = -7;      // Negative number
```

Literal Method v/s Constructor Method For Creating a Number:

```
let num = 131245;
let num1 = Number("21234");
let num2 = Number(12352);
```

Literal Method

```
let num2 = new Number(12352354);
```

Constructor Method

new Number() creates an object, not a primitive number.

This can lead to confusing bugs and unexpected behavior.

Always use number literals or Number(value) without new.

Number Properties & Methods :

Property	Description
Number.MAX_VALUE	Largest possible number
Number.MIN_VALUE	Smallest possible number (positive)
Number.POSITIVE_INFINITY	Infinity
Number.NEGATIVE_INFINITY	-Infinity
Number.NaN	"Not-a-Number"
Number.EPSILON	Smallest difference between numbers
Number.isNaN()	Checks if value is NaN

Method	Description
parseInt()	Parses string to integer
parseFloat()	Parses string to float
isNaN()	Checks if value is NaN
Number.isFinite()	Checks if number is finite

Number Instance Methods (Used on Number Primitives)

Method	Description
.toFixed(n)	Formats number to n decimal places
.toExponential(n)	Converts to exponential notation
.toPrecision(n)	Formats to n total digits
.toString()	Converts number to string
.valueOf()	Returns primitive value of Number object

Math

Math in JavaScript

JavaScript provides a built-in Math object for mathematical operations.

Method	Description
Math.round(x)	Rounds to nearest integer
Math.floor(x)	Rounds down
Math.ceil(x)	Rounds up
Math.abs(x)	Absolute value
Math.sqrt(x)	Square root
Math.pow(x, y)	x to the power of y
Math.min(...args)	Smallest number
Math.max(...args)	Largest number
Math.random()	Random number between 0 (inclusive) and 1 (exclusive)

Generating Random Numbers In A Given Range

```
// Random number between 0 and 1
let rand = Math.random();

// Random integer between min and max (inclusive)
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

console.log(getRandomInt(1, 10)); // Random integer 1 to 10
```

Note:

- Math.ceil() gives 0 only if Math.random() returns exactly 0 — but that's extremely rare.
- Math.floor(... + 1) is slightly more consistent in intent and avoids the edge case of 0.

Date

Dates In JavaScript

Fundamentals:

1. GMT (Greenwich Mean Time) Time Zone
2. UTC (Coordinated Universal Time)
3. ISO Standard (International Organization for Standardization)
4. Unix Time And Unix Epoch
5. How computers keep track of time?
6. Time Zones

GMT (Greenwich Mean Time)

GMT stands for Greenwich Mean Time.

It was the first global standard for time.

UTC (Coordinated Universal Time)

UTC is the modern standard for time measurement.
Similar to GMT, but more accurate (uses atomic clocks).

It’s the reference time zone used by computers and systems.

Time zones are usually written as offsets from UTC, like:UTC+1, UTC-4, etc.

ISO Standard (International Organization for Standardization)

ISO 8601 is the international standard for date and time formats.

It helps avoid confusion caused by different formats (like dd/mm/yyyy vs mm/dd/yyyy).

The ISO format is:

```
YYYY-MM-DDTHH:mm:ss.sssZ
Example: 2025-05-19T14:30:00Z
```

- T separates date and time
- Z means UTC time

Unix Time And Unix Epoch

Unix Epoch is the zero point for time in most computer systems, especially those based on Unix (including Linux, macOS, and modern operating systems).

So, The Unix Epoch is the starting point:

```
January 1, 1970, 00:00:00 UTC
```

Unix Time (or Epoch Time) = Number of seconds (or milliseconds) since that moment.

```
Date.now(); // gives milliseconds since epoch
```

Used in almost every programming language (including JavaScript).

How computers keep track of time?

Computers don’t store dates in human-readable format.

They count time as numbers — specifically:

- Number of milliseconds or seconds since a fixed point in time.

This makes it easier to do calculations (like adding days or comparing dates).

Time Zones

The Earth is divided into 24 time zones, based on the rotation of the planet.

Each time zone represents a region where the local time is the same.

Example:

- India (IST) = UTC +5:30
- London (GMT/UTC) = UTC +0
- New York (EST) = UTC -5

Term	Meaning
Time Zone	Local time difference from UTC
GMT	Historical time standard
UTC	Global, accurate time reference
ISO Format	Standard format for date-time
Unix Time	Time counted from Jan 1, 1970
Epoch	Starting reference time for systems

Date In JavaScript

1. Creating a Date

```
// ✅ 1. No Arguments (Current Date & Time)
let now = new Date(); // Current date and time

// ✅ 2. Date String (ISO or other formats)
let date1 = new Date("2025-05-19T12:00:00"); // ISO format (recommended)
let date2 = new Date("May 19, 2025 12:00:00"); // Long format
let date3 = new Date("2025/05/19 12:00:00");   // Slash format (less reliable)

// ✅ 3. Numbers: new Date(year, monthIndex, day, hours, minutes, seconds, ms)
let date4 = new Date(2025, 4, 19, 12, 0, 0); // May 19, 2025, 12:00:00

// ✅ 4. Milliseconds Since Epoch (Unix timestamp)
let date5 = new Date(1747632000000); // Milliseconds since Jan 1, 1970

// ✅ 5. Copy Another Date
let original = new Date("2025-05-19");
let copy = new Date(original);
```

2. Getting Parts of the Date

```
date.getFullYear(); // 2025
date.getMonth(); // 0-11 (0 = January)
date.getDate(); // 1-31
date.getDay(); // 0-6 (0 = Sunday)
date.getHours(); // 0-23
date.getMinutes(); // 0-59
date.getSeconds(); // 0-59
date.getTimezoneOffset() // UTC - local time
date.getMilliseconds() // 0-999 (milliseconds portion of the current second)
```

3. Setting Parts of the Date

```
now.setFullYear(2024);
now.setMonth(0); // January
now.setDate(1);
```

4. Working with Timestamps

```
Date.now(); // milliseconds since Jan 1, 1970
let timestamp = new Date().getTime();
```

5. Formatting Dates

```
date.toISOString();
date.toString();
date.toDateString();
date.toTimeString();
date.toLocaleString();
date.toLocaleDateString();
date.toLocaleTimeString();
```

Custom Formatting (like dd/mm/yyyy) :

```
let d = new Date();
let formatted = `${d.getDate()}/${d.getMonth()+1}/${d.getFullYear()}`;
```

Method	What it returns
toISOString()	Standard UTC format (ISO 8601)
toString()	Full date & time string
toDateString()	Date only (weekday, month, year)
toTimeString()	Time only with time zone
toLocaleDateString()	Local date format (e.g., DD/MM/YYYY)
toLocaleTimeString()	Local time format (e.g., 12-hour)

Notes:

- Months are zero-indexed in JS: 0 = January, 11 = December.
- JS Date is based on UTC internally, but displays in local time.
- Date calculations can be tricky—consider libraries like Luxon, Day.js, or date-fns for advanced use.

Calculations On Time Stamps

1. What Is a Timestamp?

- A timestamp is the number of milliseconds since the Unix Epoch (Jan 1, 1970 UTC).
- In JavaScript, Date objects internally store time as timestamps

2. Getting the Current Timestamp

- Use Date.now() to get the current timestamp:
- const now = Date.now(); // milliseconds since Jan 1, 1970

3. Convert Date to Timestamp

Use getTime() on a Date object:

```
const date = new Date();
const timestamp = date.getTime();
```

4. Convert Timestamp to Date

Pass the timestamp to the Date constructor:

```
const date = new Date(17162126678533);
console.log(date)
console.log(date.toLocaleString());
```

5. Adding or Subtracting Time

Timestamps are in milliseconds, so you can add or subtract directly:

- 1 second = 1000 ms
- 1 minute = 60 * 1000 ms
- 1 hour = 60 * 60 * 1000 ms

Example:

```
const oneHourLater = Date.now() + (60 * 60 * 1000);
```

6. Calculating Differences Between Dates

Subtract timestamps to find duration:

```
const start = new Date('2025-05-20T10:00:00Z');
const end = new Date('2025-05-20T12:00:00Z');
const diffMs = end - start; // in milliseconds
const diffHours = diffMs / (1000 * 60 * 60);
```

7. Converting Milliseconds to Human Units

Convert milliseconds into readable time:

```
const ms = 5405000;
const minutes = Math.floor(ms / 1000 / 60);
const seconds = ((ms % 60000) / 1000);
console.log(`${minutes} min ${seconds} sec`)
```

Locale & Options

```
const date = new Date();

date.toLocaleString(locale, options) // Full date + time
date.toLocaleDateString(locale, options) // Date only
date.toLocaleTimeString(locale, options) // Time only
// locale --> en-IN, en-GB, fr-Fr, hi-En
```

Option	Possible Values
weekday	"long", "short", "narrow"
year	"numeric", "2-digit"
month	"numeric", "2-digit", "long", "short"
day	"numeric", "2-digit"
hour	"numeric", "2-digit"
minute	"numeric", "2-digit"
second	"numeric", "2-digit"
hour12	true or false
timeZone	"UTC", "Asia/Kolkata", etc.
timeZoneName	"short", "long"

Practice Set

1. Create a function that returns the last element of an array.

2. Find the combination of two arrays.

3. Generate a random integer between 0 to 18.

4. Create a function that takes an array containing both numbers and strings, and returns a new array containing only the string values.

5. Find the maximum number in an array.

6. Write a function that returns the length of a given object (number of keys).

7. In an array of objects filter out those objects which have gender's value male.

8. Given an array of strings, return a new array where all strings are in uppercase.

9. Check if an object is empty (has no keys).

10. Create an array of numbers and double each value using .map().

11. Convert an array of strings into a single comma-separated string.

12. Write a function to flatten a nested array (one level deep).(e.g., [1, [2, 3], 4] → [1, 2, 3, 4])

13. Write a function that checks if all elements in an array are numbers.

14. Build a simple isPrime() function to check if a number is prime.

15. Create a function that removes duplicate values from an array.

16. What’s the difference between parseInt and Number()?

17. Why does 0.1 + 0.2 !== 0.3 in JavaScript?

18. Explain floating-point precision issues in JavaScript.

19. How would you handle high-precision decimal math in JS?

20. What is the difference between slice and splice?

21. Create a function that reverse each word of a given sentence. E.g., Mai hun manas → sanam nuh iam

22. In an array of numbers and strings, only add those numbers which are not strings.

23. How would you check if a number is an integer?

24. Write a JavaScript function that reverse a number.

25. Write a javascript function that returns a passed string with letters in alphabetical order.

26. Write a Javascript function that accepts a string as a parameter and converts the first letter of each word of the string in upper case.

27. Write a javascript function to get the first element of an array. Passing a parameter 'n' will return the first 'n' elements of the array.

28. Write a javascript function to get the number of occurrences of each letter in specified string.

29. write a javascript program to find the most frequent item of an array.

30. write a javascript program to shuffle an array.

31. How can you update the DOM based on user input in real-time (e.g., live preview of a form)?

32. How would you handle form validation in real time and show error messages dynamically?

33. How do you find the closest ancestor element that matches a selector?

34. How would you toggle a class on an element when another element is clicked (e.g., show/hide sidebar)?

35. Create a Show/Hide Password Toggle

36. Create a Sticky Header on Scroll. The header becomes fixed to the top after scrolling down. (Key concepts: scroll event, window.scrollY, classList.add/remove)

37. What is a pure function, and why is it useful in UI rendering?

38. How would you use .map() to transform a list of products into a list of HTML elements?

39. How do you use .reduce() to calculate the total price in a shopping cart?

40. Explain immutability and how you would update an object in an array without mutating the original.

41. How would you compose multiple functions to transform data step-by-step (e.g., sanitize → trim → capitalize)? Scenario: You're preparing user input before storing it. (Expected knowledge: Function composition, chaining, pipe or compose logic.)

42. What is the difference between forEach and map, and when is it wrong to use map?

43. How do you implement your own version of .map() function on arrays?

44. Guess the Output:

```
let date = new Date(0);
console.log(date.toString());
```

45. Validate that a user's selected date range is no longer than 30 days.

46. Calculate difference between two dates in the format of “_ years _ months _days _ hours _ minutes _ sec”.

47. Add or subtract n days from a given date. (E.g., Add 7 days to "2025-05-01“)

48. Calculate the user's age from their date of birth.

49. Write a formatDate(dateStr) function that returns a user-friendly format like Jan 1, 2025. Why is it better to centralize this logic in a utility?

50. What are the different options and parameters available in toLocaleString and toLocaleDateString etc to format date.

Programming Paradigm

Programming Paradigm:

1. Imperative Paradigm
    - Procedural Programming
    - Structured Programming
2. Declarative Paradigm
    - Functional Programming
    - Reactive Programming (e.g., RxJS)
3. Object-Oriented Programming
    - Class-based
    - Prototype-based
4. Event-Driven Programming
    - Based on event listeners, callbacks, DOM events, etc.
5. Asynchronous Programming (cross-paradigm)
    - Callback-based
    - Promise-based
    - async/await (syntactic sugar over Promises)

A programming paradigm is a style or way of programming.

Common paradigms include:

1. Imperative programming (writing step-by-step instructions)
2. Declarative Programming (What it is: Focuses on what to do, not how to do it.)
3. Event-driven programming (responding to events like clicks or messages)
4. Functional programming (using functions as the core building blocks)
5. Object-oriented programming (organizing code around objects)

Most Used Programming Paradigm:

Paradigm	Common in JS?	Description
1. Imperative Programming	☑ Very common	Writing step-by-step logic like loops, conditionals, and variable updates.
2. Declarative Programming	☑ Very common	What it is: Focuses on what to do, not how to do it. Seen in Array.map, JSX in React, etc.
3. Event-Driven Programming	☑ Core in JS	Central to DOM events, and browser interactivity.
4. Functional Programming	☑ Increasingly popular	Emphasized in modern JS with arrow functions, map, filter, reduce, immutability, etc and avoid shared state & side effects etc.
5. Object-Oriented Programming	☑ Still widely used	JavaScript supports both prototype-based and class-based OOP. Used in frameworks and app architecture.

Imperative Programming v/s Declarative Programming:

Concept: You tell the computer how to do something step by step.

Think of it like: Giving someone exact instructions to make a sandwich — one step at a time.

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

for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2);
}

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

Concept: You describe what you want, not how to do it.

Think of it like: Ordering a coffee — you just say "I want a cappuccino", you don't explain how to make it.

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

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

Event-Driven Programming

Concept: Code responds to events like button clicks, form submissions, or messages.

Think of it like: A doorbell — your action (pressing the bell) triggers a response (someone answers).

```
<button onclick="greet()">Click Me</button>

<script>
    function greet() {
        alert("Hello, School4U!");
    }
</script>
```

Functional Programming

Concept: Uses pure functions, avoids changing variables, and focuses on data transformation.

Think of it like: A machine that always gives the same output for the same input.

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

const result = add(2, 3); // 5
console.log(result);
```


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

const result = add(2, 3); // 5
console.log(result);
```

Object-Oriented Programming (OOP)

Concept: Organize code into objects with properties (data) and methods (behavior).

Think of it like: A car object — it has properties (color, brand) and methods (drive()).

```
class Car {
    constructor(brand) {
        this.brand = brand;
    }

    start() {
        console.log(`${this.brand} is starting...`);
    }
}

const myCar = new Car("Buggati");
myCar.start(); // Buggati is driving
```

Functional Programming

Functional programming

Functional Programming is a programming paradigm where functions are first-class citizens, and the focus is on pure functions, immutability, and function composition rather than shared state and side-effects.

1. Pure Functions
2. Immutability
3. Declarative
4. Avoid Shared state
5. Avoid Side Effects
6. Reuse or Compose Logic
7. Don’t Iterate
8. Loose coupling
9. First-Class & Higher-Order Functions

1. Pure Functions

A function is pure if:

- ❑ It returns the same output for the same input.
- ❑ It doesn’t cause side effects (like modifying external variables or DOM).

```
// ❌Impure ****
let count = 0;
function increment() {
  count++; // modifies external state
}

// ✅Pure ****
function add(a, b) {
  return a + b;
}
```

2. Immutability

Do not modify existing data. Instead, return new copies.

```
// ❌Mutable *****
const user = { name: "Alice" };
user.age = 25; // directly modifies original object

// ✅Immutable *****
const updatedUser = { ...user, age: 25 };
```

3. Declarative Code

Describe what should be done, not how.

```
// ❌Imperative ****
let doubled = [];
for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2);
}

// ✅Declarative ****
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(n => n * 2);
```

4. Avoid Shared State

Shared mutable state can lead to bugs, especially in async or parallel systems.

```
// ❌Shared State (Bad) ***
let total = 0;
function addToTotal(n) {
  total += n;
}

// ✅avoid shared state ***
function add(a, b) {
  return a + b;
}
```

5. Avoid Side Effect

Side effects are anything a function does outside its scope
(API call, DOM update, modifying global vars).

```
// ❌Side Effect ***
function logMessage(msg) {
  console.log(msg); // side effect: interacts with console
}

// ✅No Side Effect ***
function getGreeting(name) {
  return `Hello, ${name}`;
}
```

6. Reuse Or Compose Logic

Build small reusable functions and compose them together.

```
const toLower = str => str.toLowerCase();
const removeSpaces = str => str.replaceAll(' ', '');
const atTheRate = str => '@' + str;

let str = "Manas Kumar Lal";
let result = atTheRate(removeSpaces(toLower(str)))
console.log(result);
```

7. Don’t Iterate (Imperatively)

Avoid for, while, etc. Use map, filter, reduce.

```
// ❌Imperative Style ***
let evens = [];
for (let n of [1, 2, 3, 4]) {
  if (n % 2 === 0) evens.push(n);
}

// ✅FP (Declarative) Style ***
const evens = [1, 2, 3, 4].filter(n => n % 2 === 0);
```

8. Loose Coupling

Coupling refers to how dependent one piece of code is on another.

Loose coupling means less dependent, Keep functions and modules independent.

```
// ❌Tightly Coupled ***
function getUserData() {
  return fetch("https://api.example.com/user").then(res => res.json());
}

// ✅Loosely Coupled ***
function getData(api) {
  return fetch(api).then(res => res.json());
}
```

9. First-Class & Higher-Order Functions

- First-Class: Functions can be stored in variables, passed, and returned.
- Higher-Order: Functions that take other functions as arguments or return them.

```
const greet = () => "Hello";
const callWithName = fn => name => `${fn()} ${name}`;

const greetUser = callWithName(greet);
console.log(greetUser("School4U")); // Hello School4U
```

Note:

- All callbacks are first-class functions, but not all first-class functions are callbacks.
- All higher order functions are first class functions but not all first-class functions are higher order functions

1. What is a pure function, and why is it useful in UI rendering?

2. How would you use .map() to transform a list of products into a list of HTML elements?

3. How do you use .reduce() to calculate the total price in a shopping cart?

4. Explain immutability and how you would update an object in an array without mutating the original.

5. How would you compose multiple functions to transform data step-by-step (e.g., sanitize → trim → capitalize)? Scenario: You're preparing user input before storing it. (Expected knowledge: Function composition, chaining, pipe or compose logic.)

6. What is the difference between forEach and map, and when is it wrong to use map?

7. How do you implement your own version of .map() function on arrays?

OOPs

Build Some Base

```
let obj = {
  name: 'Manas',
  age: 21,
  passion: 'Bkaiti',
  showMyDetails() {
    console.log(`
      My name is ${this.name},
      age is ${this.age},
      passion is ${this.passion}
    `)
  }
}
```

Properties

Method

"this" Keyword

The different ways to create and use objects in JavaScript — these are the foundations for understanding OOP in JS:

1. Object Literal

2. Factory Function

3. Constructor Function

4. Class Syntax (ES6)

1. Object Literal

- Simplest and most common way to create an object.
- Used when creating a single, specific object.

```
const student = {
  name: "Manas",
  age: 21,
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

student.greet(); // Hello, my name is Manas
```

Properties

Method

2. Factory Function

- A function that returns a new object.
- Great for creating multiple similar objects without classes.

```
function createStudent(name, age) {
    return {
        name,
        age,
        greet() {
            console.log(`Hi, I'm ${name}`);
        }
    };
}

const s1 = createStudent("Manas", 21);
const s2 = createStudent("Muskan", 19);
s1.greet(); // Hi, I'm Manas
```

Doesn’t involve prototypes by default (unless you manually set them).

3. Constructor Function

- Uses the new keyword.
- Before class syntax was introduced in ES6, this was the standard way to create "object blueprints."

```
function Student(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function () {
        console.log(`Hello, I'm ${this.name}`);
    };
}

const s1 = new Student("Muskan", 24);
s1.greet(); // Hello, I'm Muskan
```

Automatically sets up a link to Student.prototype.

4. Class Syntax (ES6)

- A modern, cleaner syntax for creating constructor functions.
- Internally still works like constructor functions.

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

    greet() {
        console.log(`Hey, I'm ${this.name}`);
    }
}

const s1 = new Student("Manas", 25);
s1.greet(); // Hey, I'm Manas
```

“this” keyword

- Its value depends on how the function or method is called.
- In OOP, this refers to the object that is calling the method.
- It's used to access the current instance's properties or methods.
- Arrow functions don’t have their own this — they inherit this from the surrounding (lexical) scope.

“new” keyword

- In JavaScript, the new keyword is used to create an instance of an object from a constructor function or class.
- It's like saying: “Make me a new object from this blueprint (function or class).”

prototype:

- In JavaScript OOP, it allows us to share methods between all instances of a class or constructor function, making code memory-efficient.
- JavaScript is a prototypal-based (or prototype-based) language.

How it works?

- Every object has an internal link to another object called its prototype.
- When you access a property or method, JavaScript looks for it in the object.
  - If not found, it climbs the prototype chain to find it.

 So, what about class in JS?

- JavaScript introduced the class keyword in ES6, but:
  - class is just syntactic sugar — underneath, it's still using prototypes.

Object Oriented Programming:

Object-Oriented Programming (OOPs) in JavaScript is a programming paradigm based on the concept of objects.

These objects encapsulate both data (attributes) and the functions that operate on that data (methods).

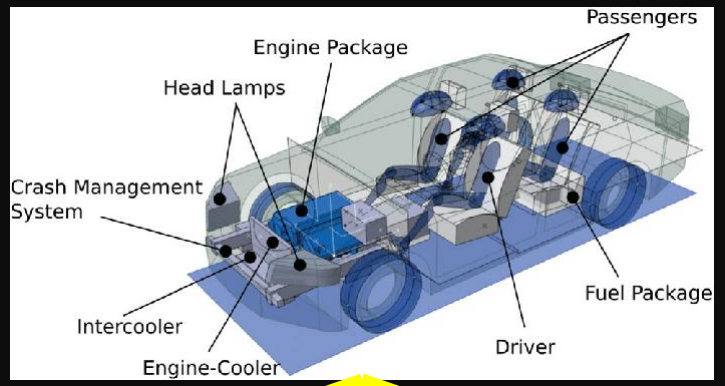
JavaScript, while not a purely class-based language like Java or C++, is heavily object-oriented and supports OOP principles through its prototype-based model and class syntax.

Class :

A Blueprint or Template, encapsulates data (properties) and functions (methods)

Object:

instance of a class, Each object has its own unique set of values for its properties.

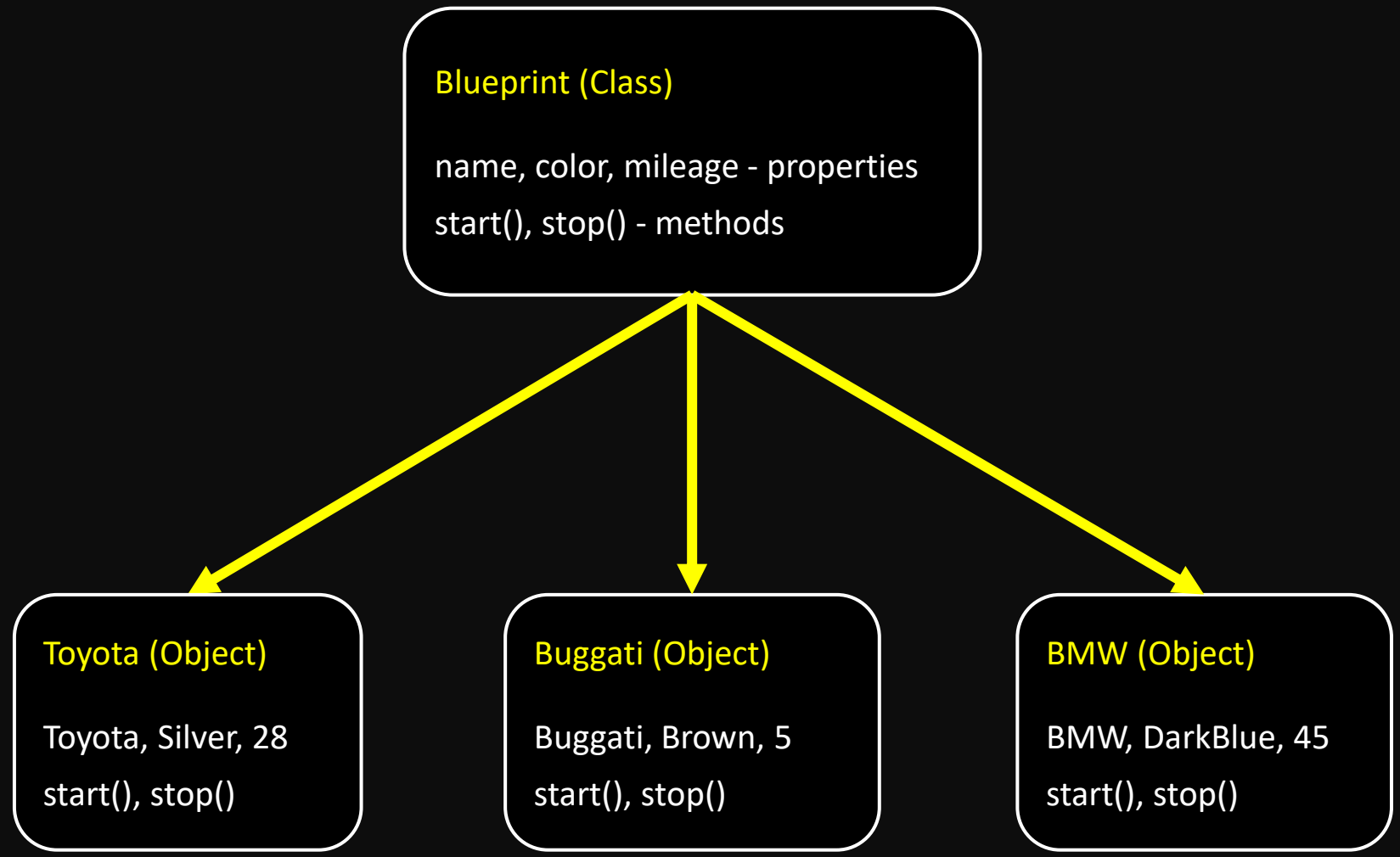


Blueprint (Class)

Toyota (Object)

Bugatti (Object)

BMW (Object)


Blueprint (Class)
name, color, mileage - properties
start(), stop() - methods
Toyota (Object)
Toyota, Silver, 28
start(), stop()
Buggati (Object)
Buggati, Brown, 5
start(), stop()
BMW (Object)
BMW, DarkBlue, 45
start(), stop()

Constructor

- A constructor is a special method within a class that is automatically called when a new object instance of that class is created.
- It is primarily used to initialize object properties with specific values or perform setup tasks for the object.

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

    introduce() {
        console.log(`name: ${this.name}, age: ${this.age}`);
    }
}

const person1 = new Person("School4U", 1);
person1.introduce(); // name: School4U, age: 1
```

Key characteristics of constructors:

- Purpose: To create and initialize objects.
- Automatic invocation: Called automatically when an object is created using the new keyword.
- Initialization: Sets initial values for object properties.
- Implicit constructor: If a class does not have a constructor, JavaScript provides a default empty constructor.
- Derived class constructor: If a derived class does not have a constructor, it calls the parent constructor passing along any arguments.

Four pillars of OOP:

- Abstraction – hiding complexity and showing only the essential features.
- Encapsulation – hiding data inside objects and provide security.
- Inheritance – using properties and methods from another object/class.
- Polymorphism – same method behaving differently based on the object.

Abstraction:

```
class Car {
    #fuel = 100;  // 🔐 Private

    #burnFuel() {  // 🔒 Hidden internal method
        this.#fuel -= 10;
    }

    start() {
        this.#burnFuel();
        console.log("Car started");
    }
}

const myCar = new Car();
myCar.start(); // ✅ Only interacts with start
// myCar.#burnFuel(); ❌ Not Accessible
```

Abstraction means hiding complex implementation details and showing only the essential features to the user.

Goal: Hiding complexity (what is irrelevant).

Encapsulation:

- It means wrapping data (properties) and methods (functions) together into a single unit, usually a class or an object, and restricting direct access to some of the components.

Why Encapsulation?

- Protects data from unauthorized access
- Prevents misuse of code
- Makes code easier to maintain
- Supports data hiding

Goal: Hide internal details and only expose what's necessary.

```
class Account {
    #balance = 0;
    constructor(balance) {
        this.#balance = balance;
    }
    #privateDetails() {
        console.log("My private details")
    }
    getBalance() {
        this.#privateDetails()
        console.log(this.#balance)
    }
    setBalance(balance) {
        this.#balance = balance
    }
}

let A1 = new Account(400);
A1.#balance = 99 // ❌Not Accessible
A1.setBalance(50000) // ✅Accessible
A1.#privateDetails() // ❌Not Accessible
A1.getBalance(); // ✅Accessible
```

- ❑ Encapsulation hides internal details
- ❑ Use # for private class fields
- ❑ Use getters/setters for controlled access

```
class Account {
    #balance = 0;
    constructor(balance) {
        this.#balance = balance;
    }
    get balance() {
        console.log(this.#balance)
    }
    set balance(balance) {
        if (isNaN(balance)) {
            console.log("please enter a valid number")
        } else {
            this.#balance = balance
        }
    }
}

let A1 = new Account(0);
A1.balance = '55';
A1.balance
```

get and set:

- They allow you to control how a property is read or written — like a security gate for your variables.
- You can check values before setting them
- Hide sensitive data
- Access methods like regular properties (obj.name)

Abstraction v/s Encapsulation:

Concept	What It Hides	What It Shows
Abstraction	The *process* / logic	A *simple interface*
Encapsulation	The *data* / internal state	Only what's allowed to access

- ❑ Use abstraction to make the system easy to use.
- ❑ Use encapsulation to make the system safe and secure.

Inheritance:

- Inheritance is an OOP concept where one class (child) can acquire properties and methods of another class (parent).

Why Use Inheritance?

- Reuse existing code
- Create logical relationships (is-a)
- Reduce duplication
- Easier maintenance and scalability

```
class Car {
    constructor(brand) {
        this.brand = brand;
    }

    drive() {
        console.log(`${this.brand} is driving... 🚗`);
    }
}

class ElectricCar extends Car {
    constructor(brand, battery) {
        super(brand); // Call parent constructor
        this.battery = battery;
    }

    drive() {
        console.log(`${this.brand} is driving silently with ${this.battery}% battery`);
    }

    charge() {
        console.log(`${this.brand} is charging...`);
    }
}

const myTesla = new ElectricCar("Tesla", 85);
myTesla.drive();  // Tesla is driving silently with 85% battery
myTesla.charge(); // Tesla is charging...
```

Polymorphism:

- Poly = many, morph = forms, Polymorphism = many forms
- It allows different classes to define methods with the same name but different behavior. (or we can say that has more than one form)

Imagine a play() button:

- On a video, it plays the video
- On a music player, it plays the music
- On a game, it starts the game

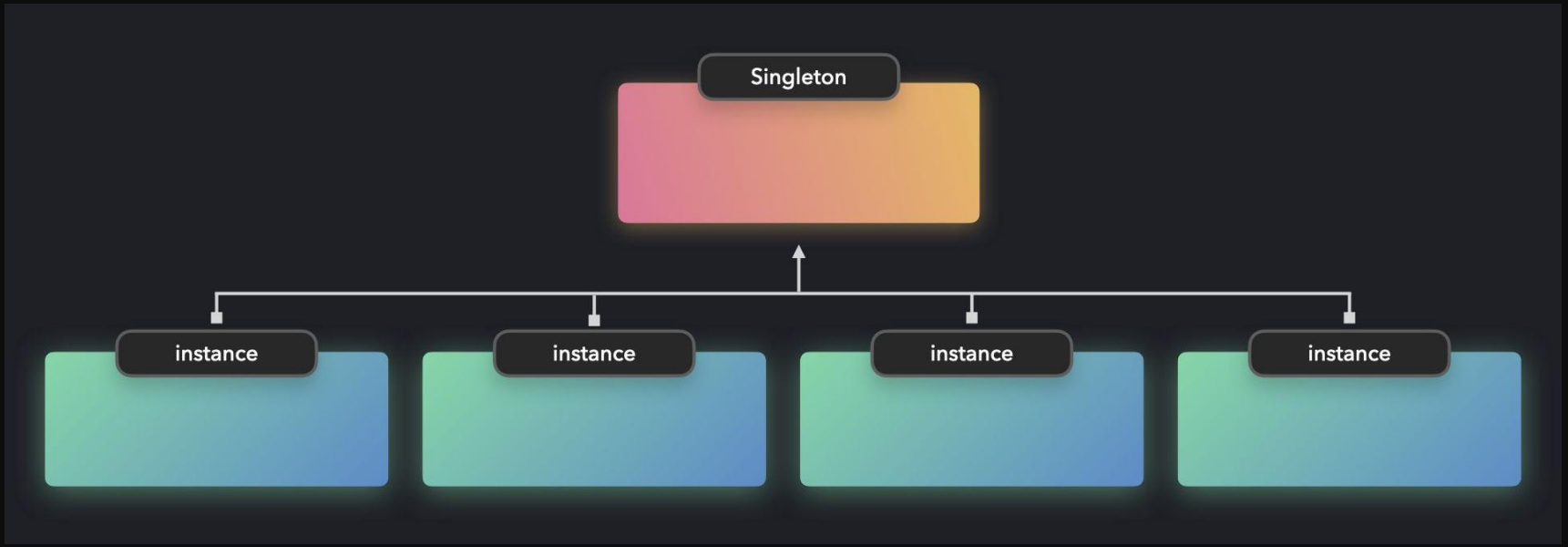
```
class MediaPlayer {
    play() {
        console.log("Playing media...")
    }
}

class Video extends MediaPlayer {
    play() {
        console.log("Playing the video...")
    }
}

class Music extends MediaPlayer {
    play() {
        console.log("Playing the music...")
    }
}

let vid = new Video()
let mus = new Music()
vid.play();
mus.play();
```

Both Video and Music override the play() method from MediaPlayer.

Singleton Object

- A Singleton Object is an object that is created only once and used everywhere in your code.

- It ensures that only one instance of that object exists during the lifetime of the application.



Example 1: Object Literal (Most Basic Singleton)

```
const config = {
  appName: "School4U",
  version: "1.0.0",
  showInfo() {
    console.log(`${this.appName} - v${this.version}`);
  }
};

config.showInfo(); // Output: School4U - v1.0.0
```

- config is a singleton object created using object literal {}.
- You can't accidentally create another version of it.
- You reuse the same config object wherever needed.

Example 2: Singleton Using Function (Closure)

```
const AppSettings = (function () {
  let instance;

  function createInstance() {
    return {
      darkMode: false,
      language: "en"
    };
  }

  return {
    getInstance: function () {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

// Usage
const settings1 = AppSettings.getInstance();
const settings2 = AppSettings.getInstance();

console.log(settings1 === settings2); // true ✅
```

- ❑ AppSettings is a self-invoking function that returns an object with a getInstance() method.
- ❑ The instance is created only once, then reused.
- ❑ Both settings1 and settings2 are same object.

Example 3: Singleton with Class (ES6 Style)

```
class Logger {
  constructor(name) {
    if (Logger.instance) {
      return Logger.instance;
    }
    this.name = name;
    Logger.instance = this;
  }

  log(greetType) {
    console.log(`${greetType} ${this.name}`);
  }
}

const logger1 = new Logger("Manas");
const logger2 = new Logger("Muskan");

logger1.log("Hello"); // Hello Manas
logger2.log("Namaste") // Namaste Manas

console.log(logger1 === logger2); // true ✅
```

- In this class, we store the first created instance as Logger.instance.
- If another object is created using new Logger(), it will return the same instance.
- It prevents creating multiple copies.

Why use Singleton?

- To avoid multiple copies of the same object.
- To maintain a single shared state.
- Useful for things like:
  - App settings
  - Database Connections
  - Authentication state
  - Logger services

JSON
&
API

Client Server Architecture

request

response

Client

Server

XML v/s JSON

```xml
<Information db="dbXML" network="HMO">
  <Dept id="1559" branch="Cola" specialist="Int Med">
    <PatientList>
      <PatInfo>
        <Patient ssn="1237435" dob="01/01/1950">
          <Name>
            <First>John</First>
            <Last>Fisher</Last>
          </Name>
          <Visit no="2" date="06/12/2004">
            <Problems>Headache</Problems>
            <Provider hospital="Baptist">
              <Physician>Dr. Mark Winderip></Physician>
            </Provider>
            <Diagnosis>
              <Prescription>Tylenol</Prescription>
              <Allergy>Sugar</Allergy>
            </Diagnosis>
          </Visit>
        </Patient>
      </PatInfo>
    </PatientList>
  </Dept>
</Information>
```

XML (Extensible Markup Language)

```json
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

JSON (JavaScript Object Notation)

JSON

- JSON stands for JavaScript Object Notation.
- It’s a lightweight data format used to store and exchange data — like sending data between a browser and a server.

```
// JavaScript Object
const user = {
    name: "Manas",
    age: 25,
    isStudent: true
};

// JSON version (as a string)
const jsonUser = `{
  "name": "Manas",
  "age": 25,
  "isStudent": true
}`;
```

Note:

- Keys and string values are wrapped in double quotes.
- It's a string, not an object

Convert Between JS and JSON

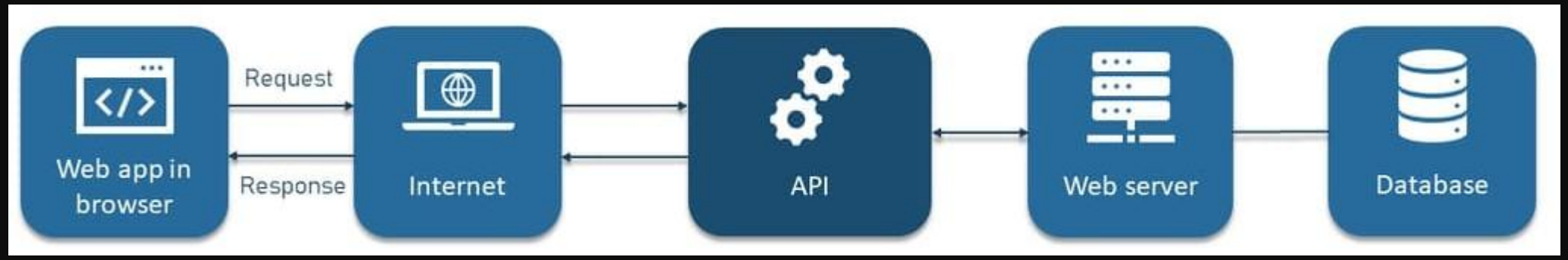
```
// Convert object to JSON
const jsonData = JSON.stringify(user);

// Convert JSON to object
const jsObject = JSON.parse(jsonData);
```

Use JSON.stringify() and JSON.parse():

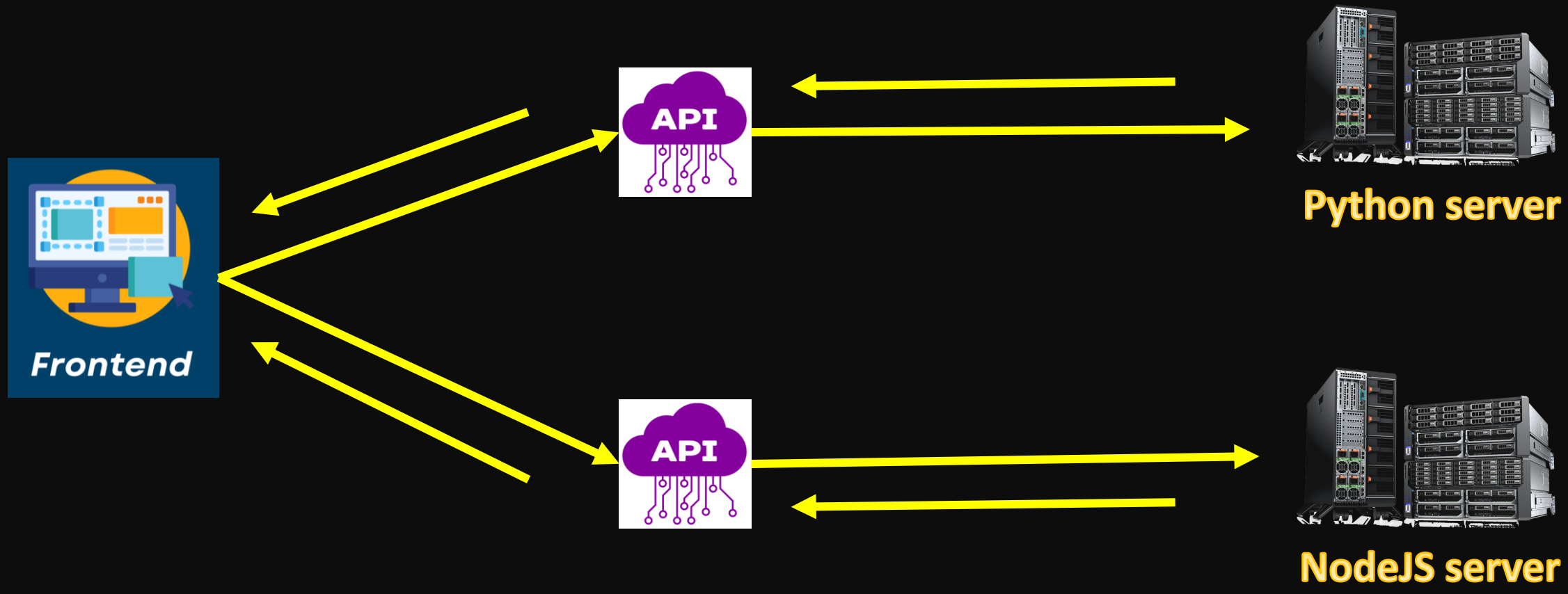
- ❑ stringify() → Object to JSON string
- ❑ parse() → JSON string to Object

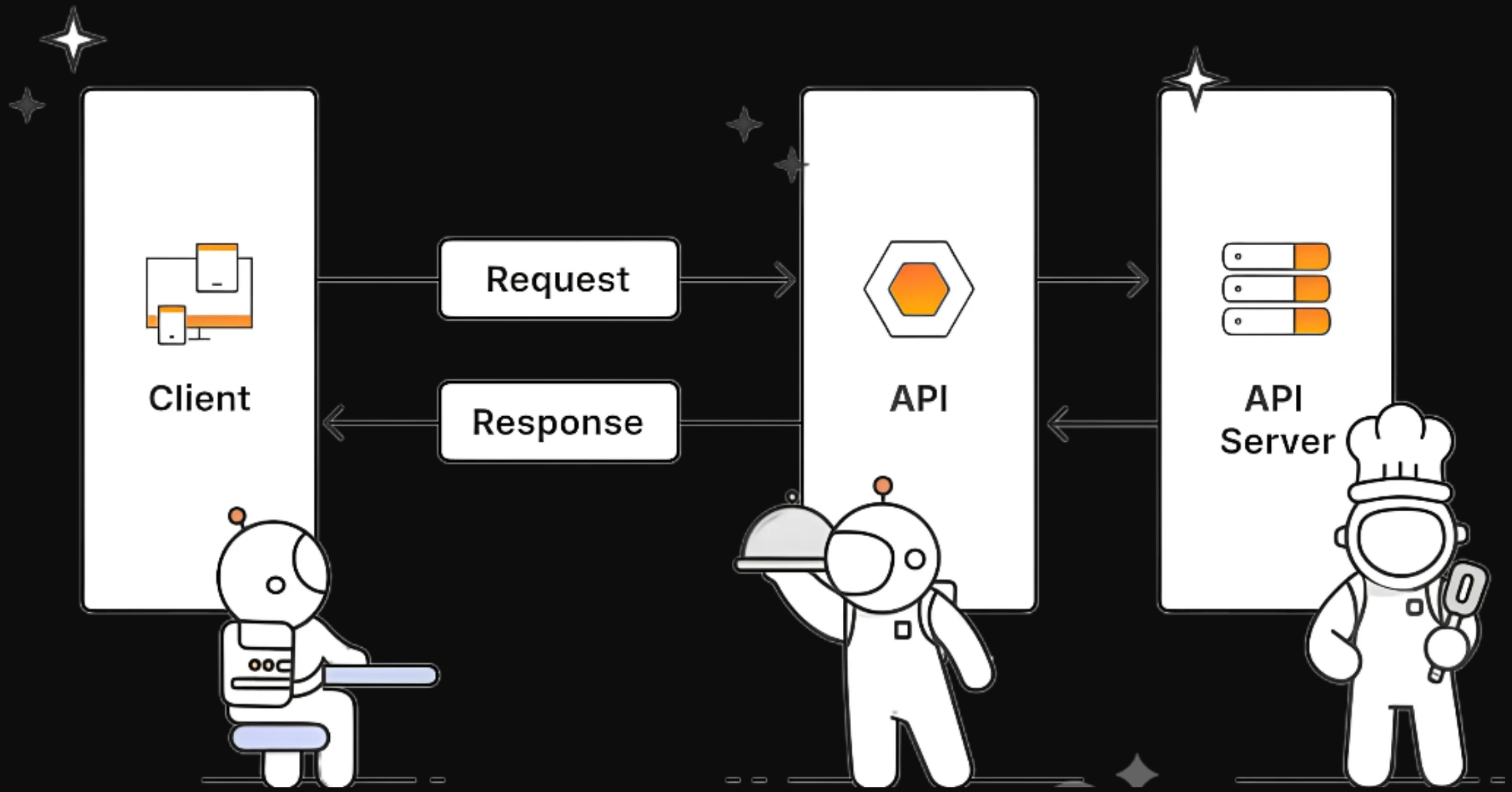
API

- API stands for Application Programming Interface.
- Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate and interact with each other.



Communication between different languages/systems




Client
Request
Response
API
API
Server


API

API Uses:

1. Communication between different languages/systems
2. Controlled, secure access to data
3. Monitoring & analytics
4. Faster development with third-party services
5. Automation & integration
6. Microservices architecture
7. Easy scaling and maintenance

Asynchronous Programming

Callbacks

Callback Hell

Promises

Promises Chaining

Async Await

Synchronous v/s Asynchronous:

- Synchronous code runs line by line. Each operation must complete before the next one starts.
- Asynchronous code can start a task and move on without waiting for it to finish.
- Asynchronous code execution allows to execute next instructions (code) immediately and doesn't block the flow.

```
console.log("task 1");
console.log("task 2");
console.log("task 3");
```

```
console.log("Start");
setTimeout(() => {
    console.log("Async Task Done");
}, 2000);
console.log("End");
```

Don’t block the other tasks due to a single lengthy/long task.

```
console.log("Hey guys..!! Do You Want Coffee??")

console.log("Muskan servers coffee");

setTimeout(() => {
    for (let i = 1; i <= 400000; i++) {
        console.log("Person", i, "Comes")
    }
}, 100);

console.log("Muskan is learning dance..!!")
```

Feature	Synchronous	Asynchronous
Execution Flow	Line by line	Skips long tasks, comes back
Blocking	Yes	No
Use Cases	Simple tasks, calculations	API calls, DB queries, timers

Why Do We Get a Promise Instead of Data? 🤔

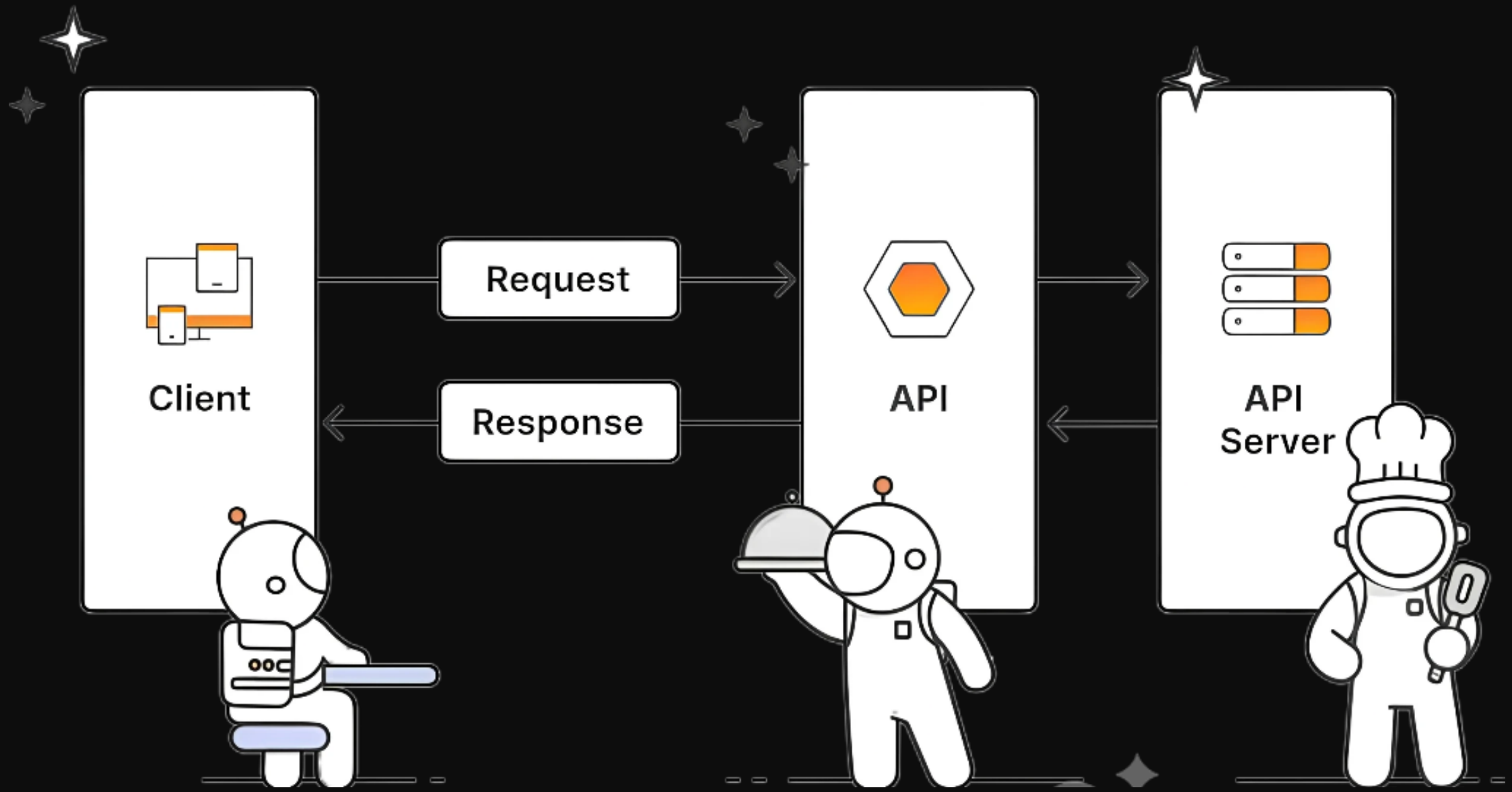
```
let data = fetch("https://jsonplaceholder.typicode.com/users");
console.log(data); // 👉 It logs a Promise, not actual data
```

You get a Promise — not the real data — because the data isn't ready yet.

API Calls Are Asynchronous

- ❑ Fetching data takes time (maybe 500ms, 2s, or more).
- ❑ JavaScript doesn't want to stop everything and wait (it’s single-threaded).
- ❑ So instead, it gives you a Promise, saying:

> “I'll give you the data later, once it arrives.”


Client
Request
Response
API
API
Server

Let’s Build A Project:

fetch

- fetch is a built-in JavaScript function used to make HTTP requests (like GET, POST) to a server or API. (It is like “Hey server, please give me some data!”)

What is CRUD?

- CRUD stands for:
  - Create
  - Read
  - Update
  - Delete
- These are the 4 basic operations we perform on data.

CRUD Operation	HTTP Method	Purpose
Create	POST	Add new data
Read	GET	Get/fetch existing data
Update	PUT / PATCH	Modify existing data
Delete	DELETE	Remove existing data

- JavaScript is single-threaded. That means it does one thing at a time.
- Suppose you want to fetch user data from a server. It takes 2 seconds. If we wait normally, the whole app freezes. Users can’t click or scroll.

Callbacks

Promises

Async Await

Callbacks:

- A Callback is a function passed as an argument to another function

```
console.log("1. Start fetching data...");

function fetchData(callback) {
    setTimeout(() => {
        console.log("2. Data fetched from server");
        callback(); // run the callback after data is fetched
    }, 3000);
}

function processData() {
    console.log("3. Now processing the data...");
}

fetchData(processData);

console.log("4. Do other things while waiting...");
```

- Callbacks help us deal with tasks that take time, like loading data from a server, without blocking other code from running.

Callback Hell (Pyramid Of Doom):

- Callback Hell happens when you have many nested callbacks — one inside another — usually in asynchronous code.

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

setTimeout(() => {
    console.log("1. Getting user from database...");

    setTimeout(() => {
        console.log("2. Getting user's orders...");

        setTimeout(() => {
            console.log("3. Processing payment...");

            setTimeout(() => {
                console.log("4. Sending confirmation email...");
            }, 1000);

        }, 1000);

    }, 1000);

}, 1000);
```

Promises:

- A Promise is a special object in JavaScript that represents a task that will finish in the future.

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve("Phone Delivered Successfully..!!");
    }, 2000);
});

promise
    .then(result => console.log(result))
    .catch(error => console.log(error));
```

- resolve and reject are callbacks provided by JavaScript.

- A promise has 3 states:
    - Pending – still waiting
    - Resolved (fulfilled) – task completed
    - Rejected – something went wrong

async await:

- async / await helps you write asynchronous code in a cleaner, more readable way — almost like it's synchronous.

```
async function getData() {
    try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.log(error);
    }
}
getData();
```

- async : Makes a function always return a Promise.
- await : Pauses inside an `async` function until the Promise is resolved.

- Code outside the async function continues immediately.
- Code inside the async function pauses at await.

Local & Session Storage

Local Storage and Session Storage

- Both localStorage and sessionStorage are part of the Web Storage API used to store key-value pairs in the browser.
- Store Data in the Browser (No Server Needed)
- Improve User Experience (keep user settings and user specific data)
- Fast & Easy Access
- Easy to Use API
- Foundation for Advanced Topics (indexDB, Cookies, PWA)

Setting data

```
localStorage.setItem('username', 'manas');
sessionStorage.setItem('sessionId', 'manas123');
```

Getting data

```
localStorage.getItem('username');
sessionStorage.getItem('sessionId');
```

Note2:

- We can also get and set item using dot notation similar to objects.

Removing data

```
localStorage.removeItem('username');
sessionStorage.removeItem('sessionId');
```

Clearing all data

```
localStorage.clear();
sessionStorage.clear();
```

Storing objects (important!)

```
const user = {
    name: "manas",
    partner: 'muskan',
};

localStorage.setItem("userData", JSON.stringify(user));

JSON.parse(localStorage.getItem("userData"));
```

Getting key

Get the key name at given index.

```
localStorage.key(1)
// get the name of key at 1st index
```

Getting length

Getting the length of localStorage using “length” property

```
localStorage.length
```

Feature	localStorage	sessionStorage
Scope	Shared across tabs and windows	Unique per tab/window
Lifetime	Persists even after browser close	Cleared when tab/window is closed
Storage Limit	~5–10 MB (per origin)	~5–10 MB (per origin)
Access	Same-origin only (not cross-site)	Same-origin only

Factors that Affect Storage:

1. Origin-based Storage
2. Browser Limitations
3. Data Type Limitation
4. Security

Origin-Based Storage

- You cannot access localStorage or sessionStorage data across origins.

  Data stored on https://siteA.com is not accessible on https://siteB.com.

- The data is only available to:
  - That specific domain
  - On that browser
  - On that device
  - In that user profile (Chrome profiles are separate)

- Even if you open “example.com” in Incognito Mode, its localStorage will be separate from the normal browser window.

Browser Limitations

- Most modern browsers allow around 5 to 10 MB per origin.
- Trying to exceed it throws a QuotaExceededError.

Data Type Limitation

- Only strings can be stored.
- You must JSON.stringify() complex data types (arrays, objects).

Security

- Not secure for sensitive data (passwords, tokens) — accessible via JavaScript.
- Vulnerable to Cross-Site Scripting (XSS) if not properly sanitized.

Spread & Rest Operator

Spread & Rest Operator

- Both use three dots (...) syntax but do different things based on context.
- Spread operator expands/unpacks values.
- Rest operator collects/packs values.

Spread Operator:

Spread Array

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];
console.log(arr2); // [1, 2, 3, 4, 5]
```

Copying Arrays (Shallow Copy)

```
const original = [10, 20, 30];
const copy = [...original];
console.log(copy); // [10, 20, 30]
```

Merging Arrays

```
const boys = ['Manas', 'Harshit'];
const girls = ['Muskan', 'Ritika'];
const students = [...boys, ...girls];
console.log(students); // ['Manas', 'Harshit', 'Muskan', 'Ritika']
```

Spread String

```
const word = "Muskan";
const letters = [...word];
console.log(letters); // ['M', 'u', 's', 'k', 'a', 'n']
```

Spread in function calls

```
function add(a, b, c) {
    return a + b + c;
}
const nums = [1, 2, 3];
let result = add(...nums);
console.log(result); // 6
```

Spread Objects

```
const user = { name: 'Muskan', age: 19 };
const updatedUser = { ...user, location: 'India' };
console.log(updatedUser); // { name: 'Muskan', age: 19, location: 'India' }
```

Merging Objects

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

Rest Operator:

Rest in Function Parameters

```
function sum(...numbers) {
    console.log(numbers)
}
sum(1, 2, 3, 4, 5); // [1,2,3,4,5]
```

Rest with Destructuring Array

```
const [first, ...rest] = [100, 200, 300, 400];
console.log(first); // 100
console.log(rest);  // [200, 300, 400]
```

Rest with Destructuring Object

```
const person = { name: 'Manas', age: 19, city: 'Bhagalpur' };
const { name, ...others } = person;
console.log(name); // 'Manas'
console.log(others); // { age: 19, city: 'Bhagalpur' }
```

Spread v/s Rest:

Feature	Spread ...	Rest ...
Purpose	Expands values	Collects values
Used in	Function calls, arrays, objects	Function parameters, destructuring
Position	Right side of = or in arguments	Left side of = or parameters
Example	add(...arr)	function add(...nums)

Destructuring

Destructuring:

- Destructuring is a JavaScript expression used to unpack values from arrays or properties from objects into distinct variables.

- Think of it like unpacking a suitcase.

```
const [a, b] = [10, 20];
const { name, age } = { name: "Manas", age: 21 };
```

Destructuring Arrays:

Basic Destructuring

```
const numbers = [1, 2, 3];
const [first, second, third] = numbers;

console.log(first); // 1
console.log(second); // 2
console.log(third); // 2
```

Destructuring With Rest Operator

```
const arr = [1, 2, 3, 4, 5];
const [a, b, ...rest] = arr; // 1 2 [3, 4, 5]
```

Default Values

```
const [x = 1, y = 2] = [10];
console.log(x, y); // 10, 2
```

Skip Array Items

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

Swapping Values

```
let a = 5, b = 10;
[a, b] = [b, a];
console.log(`a=${a}, b=${b}`); // a=10, b=5
```

Nested Arrays

```
const users = ['manas', ['muskan', 'mehek']];
const [user1, [user2, user3]] = numbers;
console.log(user1); // 'manas'
console.log(user3); // 'mehek'
```

Destructuring Objects:

Basic Destructuring

```
const obj = { name: "Manas", age: 21 };
const { name, age } = obj;
console.log(name); // "Manas"
```

Destructuring With Rest Operator

```
const obj = {
    name: "Manas",
    age: 21,
    city: 'Bhagalpur',
};
const { name, ...rest } = obj;
console.log(name); // "Manas"
console.log(rest); // { age: 21, city: 'Bhagalpur' }
```

Default Values

```
const obj = { name: "Manas", age: 21 };
const { name, city = "Unknown" } = obj;
console.log(city); // "Unknown"
```

Renaming Variables

```
const obj = { name: "Manas Kumar Lal", age: 21 };
const { name: fullName, age: years } = person;
console.log(fullName); // "Manas Kumar Lal"
```

Nested Objects

```
const user = {
    id: 1,
    profile: {
        firstName: "Manas",
        location: "India"
    }
};

const {
    id,
    profile: { firstName, location }
} = user;

console.log(id); // 1
console.log(firstName); // "Manas"
```

Destructuring in Function Parameters:

Arrays in Parameters

```
function sum([a, b]) {
    return a + b;
}

console.log(sum([5, 10])); // 15
```

Objects in Parameters

```
function greet({ name, age }) {
    console.log(`Hello ${name}, you are ${age} years old.`);
}

let obj = {
    name: "Manas",
    age: 21,
}

greet(obj);
```

1. What will be the output?

```
const arr = [1, 2, 3];
const obj = { ...arr };
console.log(obj);
```

2. How does using spread help avoid mutation? Modify the object without affecting the original?

3. Write a function that take numbers as argument and separates even and odd numbers and return an object with evens and odds and destructure the output while calling function.

4. Create a custom JavaScript function that behaves like React's useState.

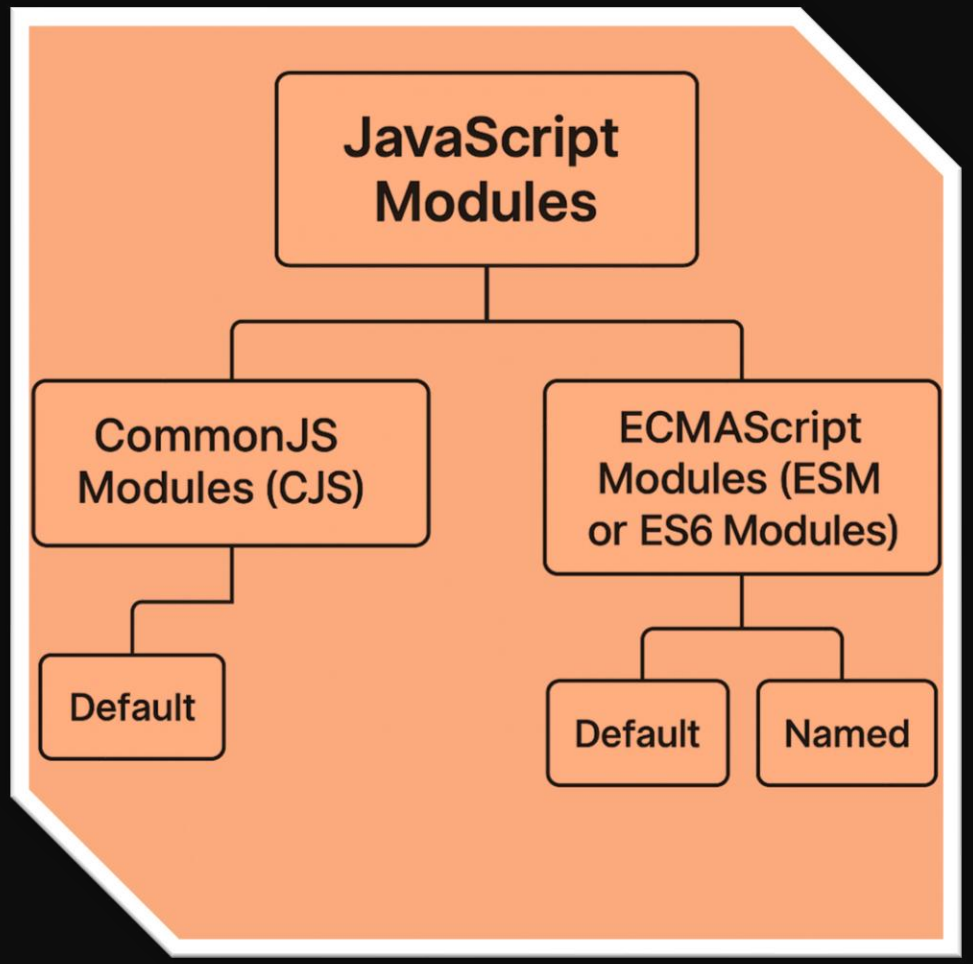
   The function should:

   - Store a value (like state).
   - Return two things: the current value and a function to update it.

   Use array destructuring to extract both the value and the setter when calling your function.

Modules

Modules



- Modules are a way to organize code into reusable files.
- They help you split your code into smaller, manageable parts, making it easier to maintain, reuse, and avoid name collisions.

CommonJS Modules (CJS)

Export single item

```
// utils.js
function greet(name) {
    console.log(`Hello, ${name}`);
}
module.exports = greet;
```

```
// script.js
const greetFunc = require('./utils');
greetFunc('Manas');
```

Note2:

- using file extension while importing is standard practice.
- Example: const alpha = require('./utils.**js**')

Export multiple item

```
// script.js
const math = require('./math');
let result1 = math.add(2, 3); // 5
let result2 = math.subtract(10, 3); // 7
console.log(result1, result2)
```

```
// math.js
function add(a, b) {
    return a + b
}

const subtract = (a, b) => {
    return a - b;
}

module.exports = {
    add,
    subtract
};
```

❑ In CommonJS, “exports” is a shorthand alias for module.exports.

```
// In CommonJS, "exports" is a shortcut for module.exports.

exports.add = (a, b) => a + b;
exports.sub = (a, b) => a - b;

// The above code is similar to below code 👇👇

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

module.exports = {
    add,
    sub,
}

// ❌ If you reassign module.exports, it breaks the link with exports.
```

ES6 Modules (modern JavaScript)

Default Export

```
// math.js
export default function add(a, b) {
    console.log(a * b);
}
```

```
// script.js
import defaultFunc from './math.js';

defaultFunc(5,2); // 10
```

Note2:

- ❑ Make Sure You're Using type: "module" in Node.js

Named Export

```
// script.js
import {
    subtract,
    multiply,
    divide,
    modulo
} from './math.js';

subtract(5, 2);
multiply(5, 2);
divide(5, 2);
modulo(5, 2);
```

```
// math.js
export function subtract(a, b) {
    console.log(a - b);
}

export const multiply = (a, b) => {
    console.log(a * b);
}

function divide(a, b) {
    console.log(a / b);
}

function modulo(a, b) {
    console.log(a % b);
}

export {
    divide, modulo
}
```