- # Internship Task 7 :

1. let and const (Block Scope)

What is Block Scope? A block is code written inside curly braces { } such as if, for, while. Variables declared with let and const are accessible only inside that block.

let

- Block scoped
- Value can be changed
- Used when the value will change

Example: if (true) { let x = 10; x = 20; console.log(x); // 20 } // x is not accessible here

const

- Block scoped
- Value cannot be reassigned
- Used for fixed values

Example: if (true) { const y = 30; console.log(y); // 30 }

Industry Use: In large projects, many developers work together. Block scope prevents accidental overwriting of variables and avoids bugs.

---

2. Data Types (Primitive and Reference)

Data types define what kind of data we store.

Primitive Data Types (store single values)

- String: "Hello"
- Number: 10
- Boolean: true or false
- null: empty value
- undefined: value not assigned
- Symbol, BigInt: advanced use

Example: let a = 10;

Reference Data Types (store multiple or complex values)

- Object
- Array
- Function

Example: let obj = { id: 1 };

Easy Rule: Primitive means simple value Reference means collection of values

---

3. Truthy and Falsy Values

Falsy Values (treated as false)

- false
- 0
- empty string
- null
- undefined
- NaN

Example: if (0) { console.log("True"); } else { console.log("False"); }

Truthy Values Anything that is not falsy is truthy Examples:

- 1
- "hello"
- empty array
- empty object
- true

---

4. Operators

Strict Equality (===) Checks value and type Example: 5 === "5" // false

Strict Not Equal (!==) Example: 5 !== "5" // true

Logical AND (&&) Returns false if any condition is false Example: age > 18 && hasID

Logical OR (||) Returns true if any condition is true Example: false || true

Ternary Operator Short form of if-else Syntax: condition ? valueIfTrue : valueIfFalse

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

---

5. Conditional Statements

if statement let age = 20; if (age >= 18) { console.log("You can vote"); }

if-else statement if (age >= 18) { console.log("Adult"); } else { console.log("Minor"); }

Ternary (short form) let msg = age >= 18 ? "Adult" : "Minor"; console.log(msg);

---

6. Functions

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

Function Expression const sub = function(a, b) { return a - b; };

Why Functions are Important:

- Avoid repeating code
- Make programs clean and reusable

---

7. Arrow Functions

Short and modern way to write functions Example: const multiply = (a, b) => a * b;

Used heavily in React, callbacks, and API handling

---

8. Default Parameters

If value is not passed, default value is used Example: function greet(name = "User") { return Hello ${name}; }

Real Use: Showing Guest when user name is missing

---

9. Return Values

return sends result back from function Example: function sum(a, b) { return a + b; }

Without return, function returns undefined

---

10. Objects

Objects store related information Example: const user = { name: "Asha", age: 22 };

console.log(user.name);console.log(user.name);

Used for:

- User data
- Product details
- API responses

---

11. Arrays

Arrays store multiple values in order Example: const nums = [1, 2, 3];

---

12. Array Methods

map: modifies data filter: selects data find: finds one value reduce: calculates total some: checks if any value is true every: checks if all values are true

```
nums.map(n => n * 2);      // modify data

nums.filter(n => n > 1);  // select data

nums.find(n => n === 2);  // find one

nums.reduce((a,b)=>a+b);  // total

nums.some(n => n > 2);    // any true?

nums.every(n => n > 0);   // all true?
```

Industry use:

Transforming API data for UI display

---

13. Destructuring

Allows extracting values easily from objects or arrays

Object Destructuring

```
const user = { name: "Amit", age: 22 };


const { name, age } = user;


console.log(name); // Amit
console.log(age);  // 22
```

- Instead of:

```
const name = user.name;
const age = user.age;
```

Array Destructuring

```
const nums = [10, 20];
const [a, b] = nums;
console.log(a); // 10
console.log(b); // 20
```

---

14. Spread Operator

Copies values without changing original data

Array Example: const nums = [1, 2, 3]; const newArr = [...nums, 4];

Object Example: const user = { name: "Amit", age: 22 }; const newUser = { ...user, city: "Delhi" };

Used heavily in React state updates

---

15. Rest Operator

Collects multiple values into one variable

Example: function total(...args) { return args.reduce((a, b) => a + b); }

Used when number of arguments is unknown

```
function total(...args) {
  return args.reduce((a, b) => a + b);
}
total(1, 2, 3, 4); // 10
```

- Here, args becomes: [1,2,3,4]

---

16. Template Literals

Used to insert variables inside strings Example: Hello ${name}

Better than string concatenation Supports multi-line strings

❌ **Old way**

"Hello " + name

✅ **Modern way**

`Hello ${name}`

✅ **Example**

let name = "Amit";

let age = 22;

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

---

17. Modules (import and export)

Used to share code between files

math.js export const pi = 3.14;

app.js import { pi } from "./math.js";

Used in React, Node.js, Angular

math.js

export const pi = 3.14;

app.js

import { pi } from "./math.js";

console.log(pi);

---

18. Classes

Class is a blueprint for creating objects

Example: class Person { constructor(name) { this.name = name; } }

class Person {

constructor(name) {

this.name = name;

}}

- Like a template for creating many persons

---

19. Constructor

Runs automatically when object is created Used to initialize values

class Person {

constructor(name) {

this.name = name;

```
 }}
const p1 = new Person("Amit");
```

---

20. Inheritance (extends and super)

Allows one class to inherit another class

Example: class Student extends Person { constructor(name, id) { super(name); this.id = id; } }

Used for roles like User, Admin, Student

```
class Person {
 constructor(name) {
  this.name = name;
 }}
class Student extends Person {
 constructor(name, id) {
  super(name); // calls parent constructor
  this.id = id;
 }}
```

- 🔹 extends → inherits
- 🔹 super() → calls parent constructor

---

21. this Keyword

Refers to the current object

Example: const user = { name: "Amit", greet() { console.log(this.name); } };

this refers to the object calling the function

```
const user = {
 name: "Amit",
 greet() {
  console.log(this.name);
 }};

user.greet(); // Amit
```

- this.name means user.name

✅ Example 2 (Multiple objects)

Copy code

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

```
}}
const p1 = new Person("Amit");
const p2 = new Person("Riya");

console.log(p1.name); // Amit
console.log(p2.name); // Riya
```

- this refers to the object that calls it

---

22. Callbacks

A function passed to another function and executed later.
Used in **asynchronous tasks** like timers, API calls.

Easy Example

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

Real use:
Button clicks, API response handling

---

23. Promises

Represents a value that will be available **in the future**.

Promise States

- pending
- fulfilled
- rejected

Easy Example

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

  if (success) {
    resolve("Data received");
  } else {
    reject("Error occurred");
  }});
```

```
promise.then(msg => console.log(msg))
    .catch(err => console.log(err));
```

-Why better than callbacks:
Cleaner and readable code

---

24. async and await
Simpler way to work with promises.
Makes async code look like normal code.

Easy Example

```
async function getMessage() {
  return "Hello";
}
getMessage().then(msg => console.log(msg));
```

With await

```
async function showData() {
  let data = await Promise.resolve("Data loaded");
  console.log(data);
}
showData();
```

Real use:
API calls in web applications

---

25. try and catch

Used to handle errors safely.
Prevents application crash.

Easy Example

```
try {
  let data = JSON.parse("hello");
} catch (error) {
  console.log("Invalid JSON");
}
```

Real use:
Handling wrong input, API errors

---

26. Scope
Defines where a variable can be accessed.

Global Scope

```
let x = 10;
```

```
function show() {
  console.log(x);
}
```

Function Scope

```
function test() {
  let y = 5;
  console.log(y);
}
```

Block Scope

```
if (true) {
  let z = 3;
}
```

Used to avoid variable conflicts

---

27. Closures

Function remembers variables from its outer function.

Easy Example

```
function counter() {
  let count = 0;
  return function () {
    count++;
    console.log(count);
  };}
let inc = counter();
inc();
inc();
```

Real use:
Counters, private data, security

---

28. Immutability
Do not change original object.
Create a new copy instead.

Easy Example

```
let user = { name: "Asha", age: 22 };
let newUser = { ...user, age: 23 };
```

console.log(user);

console.log(newUser);

Used in:
React state management

---

29. Higher Order Functions
Functions that take or return other functions.

map()

let nums = [1, 2, 3];

let double = nums.map(n => n * 2);

filter()

let even = nums.filter(n => n % 2 === 0);

reduce()

let sum = nums.reduce((a, b) => a + b, 0);

Real use:
Data transformation from APIs

30. DOM Basics

DOM is the structure of HTML that JavaScript can change Used to modify content, styles, and elements

Why DOM is needed?

Because **JavaScript cannot directly change HTML**, it uses DOM to:

- Change text
- Change styles
- Add or remove elements

Example HTML

<h1 id="title">Hello</h1>

Access DOM using JavaScript

const heading = document.getElementById("title");

heading.innerText = "Hello JavaScript";

- Browser updates the page instantly.

---

31. Event Handling

Reacting to user actions like click and submit

<button id="btn">Click Me</button>

js

Copy code

document.getElementById("btn").addEventListener("click", () => {

```
  alert("Button Clicked!");
});
```

- When user clicks → event runs → alert shows.

---

32. Event Bubbling

Event moves from child to parent Can be stopped using stopPropagation

```
<div id="parent">
  <button id="child">Click</button>
</div>
document.getElementById("parent").addEventListener("click", () => {
  console.log("Parent clicked");
});


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

-Output when clicking button:

nginx

Copy code

Child clicked

Parent clicked

- Event goes child → parent

---

33. LocalStorage and SessionStorage

LocalStorage: data stays until deleted SessionStorage: data removed when tab closes Stores data as strings only

LocalStorage

✓ Data stays after refresh
✓ Used for theme, login, cart

Save data

```
localStorage.setItem("name", "Amit");
```

Get data

```
localStorage.getItem("name"); // Amit
```

Remove

```
localStorage.removeItem("name");
```

🟦 **SessionStorage**

✓ Data removed when tab closes

sessionStorage.setItem("token", "abc123");

---

34. JSON

JSON is a text format for data exchange stringify converts object to string parse converts string to object Used in APIs and storage

❌ **JavaScript Object**

```
const user = {
  name: "Amit",
  age: 22
};
```

✅ **JSON format**

```
{
  "name": "Amit",
  "age": 22
}
```

- Keys & strings are in **double quotes**

🟦 **JSON.stringify()**

Converts JS object → JSON string

```
const user = { name: "Amit", age: 22 };
const jsonData = JSON.stringify(user);

localStorage.setItem("user", jsonData);
```

🟦 **JSON.parse()**

Converts JSON string → JS object

```
const data = localStorage.getItem("user");
const userObj = JSON.parse(data);

console.log(userObj.name); // Amit
```