JS - BASICS - 4

FUNCTIONS :-

A Javascript function is a block of code designed to perform a particular task.

Syntax :-

```
function func_name(parameters) {
    ---
    ---
    ---
    console.log()
}
```

function → Keyword
func_name → function name
parameters → Input parameters
--- / --- / --- → function body

Qu: Why functions?

Sol⇒ Reusability of code are provided in functions. that handle issues are :-
- Bulky
- Bad Readable
- Buggy

* Function Invoke :-

When a function is called, then it is known function invoking.
(Automatically - Self Invoked).

ex:-

Type-①

```
function sun()
{
    console.log('Running');
}
sun();
```

go to Top due to JS Hoisting

sun(); // function invoked at below the function

Type-②

```
// function invoked before the function.
sun();
function sun() {
    console.log('Running');
}
```

Hoisting :-

→ JavaScript Hoisting refers to the process where by the interpreter appears to be move the declaration of functions, variables or classes. at the top of the Scope before code execution.

→ It allows us to call functions before even writing them in our code.

→ Function declaration में function call ऊपर & नीचे कहीं भी कर सकते हैं।

→ Function Assignment में सिर्फ नीचे ही करते हैं।

Function Assignment :-

There is another method to create functions. using function assignment.

Two Types :-

- Named Function Assignment
- Anonymous Function Assignment.

(i) Named Function Assignment :-

Function with function name is assigned to a variable name.

example :-

Prefect ✓

```
1. let named = function walk()
   {
       console.log("Walking");
   };

   named();   //function call must be after declaration.
```

Output :-

Walking

* Note :- If we call `walk();` → it gives ERROR Not defined.
→ Does not call like this.

* If we call `named();` before assignment, then it also gives

ERROR :- CANNOT ACCESS 'named' before Initialization.

XXX

example :-

```
named();
let named = function walk(){
    console.log("Walking");
};
```

* `let jump = named;` // Also work ✓

Output :-

Walking ✓

(iii) Anonymous function Assignment :-

Function without function name is assigned to a variable name.

example :-

xx Not present

```
let named2 = function(){
        console.log("Walking");
};

named2();
```

As JS dynamic typed

The Arguments Object :-

Special

Arguments is an array-like object accessible inside functions that contains the values of the arguments passed to that function.

Let us understand with problem in Normally

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

→ It works only on 2 parameters ✓

Not working on :- sum() sum(1) sum(1,2,3,4,5) XXX → NaN WRONG Output

By using Arguments :-

Argument is a Special-Object

```
function Sum(a,b){
    let total = 0;
    for(let value of arguments)
    {
        total += value;
    }
    return total;
}

console.log(Sum(1,2,3,4));
console.log(Sum());
console.log(Sum(1));
```

→ stores all passed parameters

Works for all parameters ✓

Output :-

10
0
1

Note :-

All passed parameters stores into Arguments Object.

To check Arguments

```
console.log(arguments);
```

Output :-

Arguments (n) [1,2,3,4, callee: f]

Object

0 : 1
1 : 2
2 : 3
3 : 4

Rest Parameters (...args)

ex:-

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array.

(1.) → It stores all passed input parameters into an Array

```
function Sum (... args) {
    console.log (args);
}
Sum (1,2,3,4,5,6);
```

all inputs stores into args

Output:-

```
[1, 2, 3, 4, 5, 6]
Prototype : Array
```

(2.) →

```
function Sum (a, ...args) {
    console.log (args);
}
Sum (1, 2, 3, 4);
```

Note:- "1" goes into 'a' variable and '2,3,4' goes into args list parameters.

args → [2,3,4]

(3.)

```
function Sum (a, b, ...args) {
    console.log (args);
}
Sum (1,2,3,4,5,6);
```

output:-

args → [3,4,5,6]

1, 2 goes into a, b respectively others into Rest.

(4.) Very Imp Note:-

We cannot add variable after ...args. It gives ERROR:-

"args must be last formal parameter":

Ex:- function Sum (a, b, ...args, c, d) → ERROR

XXX

Default Parameters :-

When users do not give input parameters for any variable, then we have to assigned the value by default of that variable.

~~example :-~~

```
function Interest (p, r=5, t){
    return p*r*t/100;
}
console.log(Interest(1000,
```

* Rules :-

(1.) After initialization of 1 parameter, after that, all the parameters must be default.

ex:- (p, r=5, t=2) ✓ Right way

✗ (p, r=5, t) XX Wrong way.

(p, r, t=2) ✓ Right way.

(2) If default value of parameters are set, Even ~~we passed~~ after User gives input value.

Then, function will take user value.

example:-

```
function Interest (p, r, t=2){
    return p*r*t/100;
}
Interest(1000, 5, 3)
```

t=2 → By default set → NOT USED ✗

3 → Here User input → This will be Used ✓

Example:-

```
function Interest (p, r=5, t=2){
    return p*r*t/100;
}
console.log(Interest(1000));
```

Output:- 100

GETTER - SETTER :-

getter → access properties. (get keyword)
Setter → Change or mutate properties. (set keyword)

Qw Why we use getter - setter ?

→ Because ${placeholder} is read only. We cannot change the value.

```
let person = {
        fName: 'Love',
        lName: 'Babbar'
    };
console.log(`${person.fName} ${person.lName}`);
```

This is only Read-only

function ही change

```
function fullName(){
    return `${person.fName} ${person.lName}`;
}
```

So, To make user input change, we make getter & setter function inside OBJECT

getter :- and setter :-

```
let person = {
        fName: 'Love',
        lName: 'Babbar',

→   get fullName(){
        return `${person.fName} ${person.lName}`;
    },

→   set fullName(value){
        let parts = value.split(' ');
        this.fName = parts[0];
        this.lName = parts[1];
    }
};

console.log(person.fullName);
person.fullName = 'Ankit Sharma';
console.log(person.fullName);
```

Output :-

```
Love Babbar
Ankit Sharma
```

→ by getter
→ by setter

ERROR Handling :-

Try, catch, throw, finally.

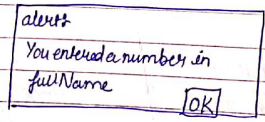
* try → defines a code block to run.
* catch → defines a code block to handle any error.
* throw → defines a custom error.
* finally → defines a code block to run regardless of the result.

try... catch... (always come in pair).

```
try {                          //Error
    person.fullName = 1;
}

catch (e) {                    //Handling Error
    alert ('You entered a number in fullName');
}
```

we have to input string → Number → Error

Output:



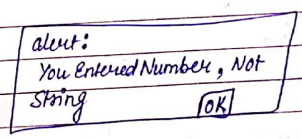
try... (throw), catch...

```
try {
    person.fullName = true;
}

catch (e) {
    alert (e);
}

Set fullName (value) {
    if (typeof value != String) {
        throw new Error ("You Entered Number, Not String");
    }
    let parts = value.split (' ');
    this.fName = parts[0];
    this.lName = parts[1];
}
```

or (1, 2, any other PRIMITIVE)

~~Set fullName (value) {~~

Output:-



JavaScript Scope :-

Three types of Scope :-

* Block Scope
* Function Scope
* Global Scope

(i) Block Scope :-

Only 'let' and 'const' are accessible inside the block.
But Not outside the block.

(1.) ex:-
```
{
  let a=5;
  console.log(a);   ✓
}

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

(2) var can be accessed anywhere.
```
{
  var a=5;
  console.log(a);   ✓
}
console.log(a);   ✓  // Can be outside block.
```

(ii) Function Scope :-

Variables defined inside a function are not accessible from outside the function.

var, let, const all have function scope

example :-

```
function my(){
  let a=5;
  console.log(a);
}
```

```
function my(){
  var a=5;
  console.log(a);
}
```

```
function my(){
  const a=5;
  console.log(a);
}
```

Cannot Accessed Outside function.

(iii) Global Scope :-

A Variable declared outside a function.
var, let, const can be accessed from anywhere.

ex:-
```
let x=2;
{
  console.log(x);   ✓  // global Scope
}

var x=2;   ✓   //
const x=2;   ✓   //
```

X