JavaScript

A scripting language is a lightweight programming language that is interpreted (not compiled). It automates tasks and is mainly used in web development.

Introduction to JavaScript (JS) - It is a client-side scripting language used to make web pages interactive.

- It works alongside HTML (structure) and CSS (design).
- Created by = Brendan Eich.
- Primary use = Adds interactivity to web pages.
- Supports both = Browser based (ie front end) and non-browser (ie backend) (eg Node.js)
- Works in all web browsers. It is object oriented & event driven.

• HOW TO ADD JS IN HTML ?

1) Inline JS = JS is directly written inside an HTML tag using the onclick, onmouseover, etc

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

2) Embedded / Internal JS = JS is written inside the <script> tag in HTML file

```
<html>
  <body>
    <h1> Embedded JS Example </h1>
    <script>
      alert ('Hello from embedded JS!');
    </script>
  </body>
</html>
```

Adv. Keeps JS separate from HTML tags
Useful for pages with small script.

Disadv: Not reusable across multiple pages
Can make HTML file larger

3) External JS = JS is written in a separate .js file & linked to HTML file using <script tag> with src attribute.

index.html (html file)

```
<html>
<head>
  <script src="script.js"></script>
</head>
<body>
  <h1> External JS Example </h1>
</body>
</html>
```

script.js (JS file external)

```
alert ('Hello from external JS!');
```

For small scripts - Inline or Embedded
For large appln - External.

Adv: Reusable
faster loading
Easy to maintain

Disadv: Requires extra HTTP request
Doesn't work if filepath is wrong.

JS Variables and Constants -

Variables are containers that stores data.

```
var x = 10;
let y = 20;
```

Constants hold fixed values that cannot be changed.

```
const PI = 3.1416;
const z = 111;
```

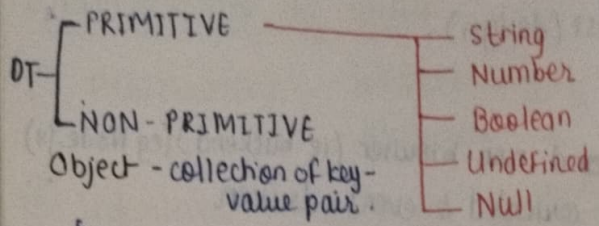
JS variable Scopes – Scope determines where a variable can be accessed

- Types of Scope in JS –

1) Global scope – Available everywhere.
2) Local scope – Available inside function.
3) Block scope – Declared using let or const inside {}

```
let globalVar = "I am global";
function testScope() {
    let localVar = "I am local";
    console.log(globalVar); ✓
    console.log(localVar); ✓
}
console.log(globalVar); ✓
console.log(localVar); ✗
```

JS Data Types



{name : "Bob", age : 30}

```
let name = "Alice";
let age = 25;
let isStudent = true;
let address; ⟶ no value assigned.
let emptyValue = null;
```

Array – List of values ⟶ ["Apple", "Banana"]

Functions – Block of reusable code ⟶ function greet() {}

JS Functions – function is a block of reusable code that perfoms a task.

Function Declaration

```
function greet(name){
    return "Hello"+name;
}
console.log(greet("Alice"));
```

Arrow function

```
const add = (a,b) ⇒ a+b;
console.log(add(5,3));
```

JS Array – Array stores multiple values in a single variable.

Declaration :

```
var House = []; or var House = new array[];
```

Initialization :

```
var House = ["1BHK", "2BHK", "3 BHK"]; or
var House[0] = "1BHK";
```

Array methods :
- push()
- pop()
- concat()
- indexOf()
- shift() – remove & return 1st element

JS Objects – Stores ~~value~~ data in key-value pairs

1) JS Built-in Objects – Array, Boolean, Date, Math, Number, String, RegExp objs
2) Browser Objects – Window, Navigator, Screen, History, Location objects.
3) DOM Objects – HTML document, HTML elements, HTML Attributes, HTML Events

Document Object Model is created by browser when a web page is loaded.

JS Events – Actions that trigger functions

eg:

```
document.querySelector("button").addEventListner("click", function(){
    alert("Button Clicked");
});
```

```
function isPrime(n){
    if (n<=1) return false;
    for (let i=2; i<n; i++){
        if (n%2)===0) return false;
    }
    return true;
}
let num = parseInt (prompt("Enter a number"));
If (isPrime (num)) {
        Console.log(num+"is a Prime No.");
    }
else{
    Console.log(num+" is NOT a Prime No.");
}
```

Advanced JS

→ not a scripting language

JSON (Javascript Object Notation) – It is a lightweight data format used for storing and transferring data. It is based on key-value pairs.

JSON Create – JSON are created using curly braces {}.

```
let person = {
  "name" :"Alice",
  "age" : 25,
  "isStudent": false
}
```

Key-Value Pair – JSON data consist of key:value pairs. (ie obj are in k:v pair)
keys are strings & values can be string, numbers, arrays, objects & boolean
The curly brace {} represent JSON object.

eg. { "employee" : { "name" : "sai",
"salary" : 56000,
"married" : true } }

JSON Array – Represents ordered list of values. JSON array can store multiples values. It can store string, number, boolean or object in JSON array. The [] represents JSON array.

eg. { "employees" : [
{ "name": "A" , "email" : "A.com", "age" : 31 },
{ "name": "B" , "email" : "B.com", "age" : 35 }
] }

JS Arrow Functions – Feature introduced in ES6 version of JS.

Allows to create function in a cleaner way compared to regular functions.

eg:
```
let x = function(x, y){
    return x*y;}
```
// using arrow functions
```
let x = (x, y) => x* y;
```

Syntax: `let myFunction = (arg1, arg2 ..., argN) => { statement(s) }`

- Arrow function with No Argument:
```
let greet = () => console.log("Hi");
greet(); //Hi
```
- Arrow function with One Argument:
```
let greet = x => console.log(x);
greet('Hello'); // Hello
```
- Arrow Function as an Expression:
```
let welcome = (age < 18)?
    () => console.log('Baby'):
    () => console.log('Adult');
welcome(); // Baby.
```

JS Callback Functions – When you need a function inside another function as an argument, that is called a callback.

ie callback is a function passed as an argument to another function.

eg.
```
function Greet(name, callback){
    console.log('Hi' + ' ' + name);
    callback();
}
// callback function
function callMe(){
    console.log('I am callback function');
}
greet('Peter', callMe);
```

output
```
Hi Peter
I am callback function
```

Why we use Callbacks? JS is asynchronous, meaning some functions take time. Instead of waiting, JS uses callbacks to continue executing the code.

CALLBACK WITH SetTimeout (Asynchronous Example)

delayed execution using setTimeout().

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

setTimeout(() => {
    console.log("Delayed msg");
}, 2000); //Runs after 2 sec.

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

output:
Start
End
Delayed msg
↳ 2sec later

* JS doesn't wait for setTimeout()
It moves to next task.

CALLBACK HELL (Problem with Callbacks)

Has too many callbacks = callback hell

eg.
```
setTimeout(() => { console.log("Step1");
    setTimeout(() => {Console.log("Step2");
        setTimeout(() => { console.log("Step3");
            }, 1000);
        }, 1000);
    }, 1000);
```

Hard to debug & read

Solution - Use Promises or Async-Await.

JS Promises - used to handle asynchronous operations like fetching data from a server. It helps avoid callback hell & makes code cleaner. Promise has 3 States:

1) Pending → Initial state, operation not finished.
2) FullFilled → Operation successful (resolve())
3) Rejected → Operation failed (reject())

CREATE A SIMPLE PROMISE

eg.
```
let myPromise = new Promise((resolve, reject) => {
    let success = true;
    if (success) {
        resolve("Task complete");
    } else {
        reject("Task failed");
    }
});

myPromise
.then(result => console.log(result)) //runs if resolved
.catch(error => console.log(error)); //runs if rejected
```

---

resolve() - runs when task is successful
reject() - runs when task is failed

then() - Handles successful results
catch() - Handles error

PROMISE CHAINING - we can chain .then() to execute multiple tasks sequentially.

eg. new Promise(resolve => resolve(10))
.then(num => num * 2)
.then(num => num + 5)
.then(result => console.log("final result:", result));

Output
25

JS Async-Await Functions -

Simpler way to handle Promises in JS. It makes asynchronous code look like synchronous code.
Before Async-Await, we used callbacks or Promises (.then() chaining)

async function - used before a function.

An async function always returns a Promise.

eg. async function greet(){
return "Hello";
}
greet().then(msg => console.log(msg)); // output: Hello

await function - await pauses execution of the function until promise is resolved. It makes code synchronous when its actually its asynchronous.

eg. async function fetchMsg(){
let promise = new Promise(resolve => {
setTimeout(() => resolve("Task complete"), 2000);
});
let result = await promise; // waits for promise to get resolve
console.log(result);
}
fetchMsg();

JS Error Handling - By using try catch block.

eg. let promise = new Promise(function(resolve, reject){
setTimeout(function(){ resolve("Promise Resolved") }, 4000); });
async function Func(){
try { let result = await promise;
console.log(result); }

```
catch (error){
        console.log(error);
  }
//calling function
Func();
```

eg.
```
try { var result = sum(10,20);
    }
catch(ex)
    {
      documet.getElementById("errorMessage")
       .innerHTML = ex;
    }
```

output:

Demo : Error Handling

ReferenceError : sum is not defined

eg:
```
try {
    var result = sum(10,20);
    }
catch(ex){
    document.getElementById("errorMessage").innerHTML = ex;
  }
finally {
    document.getElementById("message").innerHTML="finally executed";
    }
```

Output

Demo : Error Handling

Error : ReferenceError : sum is not defined

finally executed.

eg: throw → used to raise error

```
try {
   throw { number : 101,
           message : "Error occured"
         };
  }
  catch(ex){
     alert(ex.number + " - " + ex.message);
  }
```

output

Demo : Throw

101 - Error occured