res\summary.js

```
// link  html file with script ------------------------------[]
// internal js :
```
    <script >
    // code
    </script>
```
    //external js file :
```
    <script src="path"></script>
```

// comments : ------------------------------[]
// single  line comments :
```
    // comment
```
//multiple line comments :
```
    /*

    */
```
// vs code option shortcut :
```
    ctrl + / : comment selected content
```
//output to screen : ------------------------------[]
//show msg in a pop up window L
```
    window.alert("content");
    // or adding window not required :
    alert("content")
```
//add a content to the html page :
```
    document.write("content");
```
//log a content to the console window :
```
    console.log("content");
```
//console methods : ------------------------------[]
//show normal content :
```
    console.log("content");
```
// show error msg :
```
    console.error("errorContent")
```
// show objects key : value in a table
```
```
```

```
    console.table(varName);
```
// add styling to log content %c :
```
    console.log("%cContent","styling");
    //example :
    console.log("%cHello %cworld ","color :red; font-size:50px","color : gray; font-size:40px");
```
// Data Types : ------------------------------[]
```
    string
    boolean
    objects [arrays | objects]
    function
    Number  [ integer | float ]
    undefined
    NaN
```
// know the data type of a content :
```
    console.log(typeof content);
```
// Variables : ------------------------------[]
// what's a variable :
/*
    an amount of space reserved to store a value
    and access it using a variable name
*/
/*
    javaScript is a Loosely Typed Language : Dynamic Typed
    unlike
    c++   that's a Strongly Typed Language
*/

//declare a variable :
```
 //using var
  var varName[=value];
 //using let
  let varName[=value];
```
// declare a constant variable:
```
    constant varName=value;
```
//the different between var && let :
/*
    var :
    1-the oldest one
    2- function scope (limited just by function scope access)
    2-a- 5-variable scope Drama(Added To window) if is not scoped by a function
    3- provide hoisting behavior (use variable before declared)
    4- you can redeclare it multiple type
    let :
    1- cams with ES6

```
    2- block scope (limited by all scope access (if ,else , function ...)  )
    3- don't support hoisting behavior you can't use it before declare it it invokes an error
    4- you can't redeclare variable of it
*/
// id attribute in html
/*
    it's considered  as global variable you access it :
*/
//example --[
```
    // html :
    <h2 id="nice">nice</h2>
    // js :
    console.log(nice);
```
// \ Character : ------------------------------[]
/*
  -it used to tell to the compiler that's not a command and I want you to considered as a text
  - used a line continue
*/
//syntax :--[
```
    console.log(' text\specialCharacters ')
```
//example 1 :--[ Escape
```
    console.log('that\'s it ')
```
//example 2 :--[ line continue
```
    console.log('hello world \
    my name is ayoub ');
```
// add new line: ------------------------------[]
    console.log("\n");
// concatenation:  ------------------------------[]
// concat two string s : ---[
````
    //method 1 :
    console.log("string1" + "string2");
    //method 2 :
    console.log("string1","string2");
````
// concat string with variables :
````
    //method 1 :
    console.log("string1" + varName);
    //method 2 :
    console.log("string1",VarName);
    //method 3 : ES6 Template Literals
    console.log(`string1 ${varName}`)
````
// Arithmetic Operators:  ------------------------------[]
```
    +
```
```

```
    -
    *
    /
    % : modulo
    ** : power
```
// postIncrement :--[

```
    // return then  increment
    varName++;
```
//preIncrement :--[
```
    // increment then return
    ++varName;
```
// Unary Plus And Negation Operators:  ------------------------------[]
//Unary Plus :
```
    // convert value to number
    +value
```
// Negation :
```
    // convert value to number  then make it negative
    -value
```
//Type Coercion / casting : ------------------------------[]
/*
    Type coercion, also known as type conversion or casting, is the process of converting a value from one data type
    to another in JavaScript. This happens automatically when an operation is performed between values of different
    types. JavaScript performs type coercion in various situations, such as:
*/
// 1. **String Concatenation**: When using the `+` operator with strings and other data types, JavaScript converts non-string values to strings.
```
    const num = 10;
    const str = "The number is: " + num; // Type coercion: num is converted to a string
    console.log(str); // Output: The number is: 10
```

// 2. **Comparison Operators**: When using comparison operators like `==` or `!=`, JavaScript may coerce values to a common type before comparing them.
 ```
    console.log(5 == "5"); // true - Type coercion: "5" is converted to a number for comparison
 ```

// 3. **Logical Operators**: The logical operators `&&` and `||` return the value of one of the specified operands, but not necessarily of type `true` or `false`.
```
    console.log(5 && "Hello"); // "Hello" - Type coercion: Both values are considered "truthy"
```

```
// 4. **Explicit Type Conversion**: You can also perform explicit type conversion using functions like `String()`, `Number()`, `Boolean()`, etc.
```
```
    const numStr = "10";
    const num = Number(numStr); // Explicit conversion from string to number
    console.log(num); // Output: 10
```
```
//Assignment Operators :  ----------------------------[]
```
```
    varName+=value;
    varName-=value;
    varName*=value;
    varName%=value;
    varName/=value;
    varName**=value;
```
```
// Numbers  : ----------------------------[]
/*
-- Double Precision: JavaScript uses double-precision floating-point format to represent numeric values.
-- Syntactic Sugar "_": Underscores can be used in numeric literals for better readability.
-- e Notation: Allows representing large or small numbers using exponent notation.
-- ** Operator: Exponentiation operator raises the left operand to the power of the right operand.
*/

console.log(1000000);       // Output: 1000000
console.log(1_000_000);     // Output: 1000000 (underscore for readability)
console.log(1e6);           // Output: 1000000 (e notation)
console.log(10**6);         // Output: 1000000 (exponentiation)

console.log(Number.MAX_VALUE);          // Output: 1.7976931348623157e+308 (maximum representable number)
console.log(Number.MAX_SAFE_INTEGER);  // Output: 9007199254740991 (maximum safe integer)

// Numbers functions : ----------------------------[]
/*
varName.toString() : convert from number to string
varName.toFixed() :  Format number with fixed decimal places
parseInt(value)  parse value to int
parseFloat(value) parse value to float
Number.isInteger(value) [ES6] Check if a number is an integer
note : you can use Number methods without put  Number. before them (not all functions )
*/
//example : --[
const num = 10;

// Convert number to string
    console.log(num.toString()); // Output: "10"

// Format number with fixed decimal places
    console.log(num.toFixed(2)); // Output: "10.00"

// Parse string to integer
    console.log(parseInt("10")); // Output: 10

// Parse string to float
```

```
    console.log(parseFloat("10.5")); // Output: 10.5

// Check if a number is an integer [ES6]
    console.log(Number.isInteger(4)); // Output: true

// Check if a value is NaN [ES6]
    console.log(Number.isNaN("Hello")); // Output: false
    console.log(Number.isNaN(NaN));     // Output: true

//Math Object: ------------------------------[]
/*
. round()
. ceil()
. floor()
. min()
. max()
. pow()
. random()
. trunc() [Es6]
*/
//examples:--[
// Math.round() - Rounds a number to the nearest integer.
    console.log(Math.round(4.7)); // Output: 5

// Math.ceil() - Rounds a number up to the nearest integer.
    console.log(Math.ceil(4.3)); // Output: 5

// Math.floor() - Rounds a number down to the nearest integer.
    console.log(Math.floor(4.7)); // Output: 4

// Math.min() - Returns the smallest of zero or more numbers.
    console.log(Math.min(4, 7, 2, 9)); // Output: 2

// Math.max() - Returns the largest of zero or more numbers.
    console.log(Math.max(4, 7, 2, 9)); // Output: 9

// Math.pow() - Returns the base to the exponent power.
    console.log(Math.pow(2, 3)); // Output: 8

// Math.random() - Returns a random number between 0 (inclusive) and 1 (exclusive).
    console.log(Math.random()); // Output: a random number between 0 and 1

// Math.trunc() [ES6] - Returns the integer part of a number by removing any fractional
digits.
    console.log(Math.trunc(4.7)); // Output: 4

// -String Methods ------------------------------[]
/*
    -Access With Index
    -Access With charAt()
    -length
    -trim()
    -toUpperCase()
    -toLowerCase()
    -Chain .Methods
*/
```

```
// -String Methods ----------------------------[]

/*
    -Access With Index
*/
// You can access individual characters in a string using square brackets [] and the index of the character. Indexing in JavaScript starts from 0.
    let str = "hello";
    console.log(str[0]); // Output: "h"
    console.log(str[1]); // Output: "e"

/*
    -Access With charAt()
*/
// Returns the character at the specified index in a string. If the index is out of range, an empty string is returned.
    console.log(str.charAt(0)); // Output: "h"
    console.log(str.charAt(4)); // Output: "o"

/*
    -Length
*/
// Returns the length of a string, i.e., the number of characters in the string.
    console.log(str.length); // Output: 5

/*
    -trim()
*/
// Removes whitespace from both ends of a string.
    let strWithWhitespace = "   hello   ";
    console.log(strWithWhitespace.trim()); // Output: "hello"

/*
    -toUpperCase()
*/
// Returns the string with all its characters converted to uppercase.
    console.log(str.toUpperCase()); // Output: "HELLO"

/*
    -toLowerCase()
*/
// Returns the string with all its characters converted to lowercase.
    let strUpperCase = "HELLO";
    console.log(strUpperCase.toLowerCase()); // Output: "hello"

/*
    -Chain .Methods
*/
// You can chain multiple string methods together, with each method being called on the result of the previous one.
    console.log(strWithWhitespace.trim().toUpperCase()); // Output: "HELLO"

// -String Methods Advance: ----------------------------[]

/*
    -indexOf(Value)
    -lastIndexOf(Value)
```

```
    -slice(Start. [start,end)
    -repeat(Times) . [ES6]
    -split(Separator [Opt])
    -substring(start,end)
    -substr(start ,characters To Extract)
    -includes(value)
    -startWidth(value)
    -endWidth(value)
*/

//✨ -indexOf(Value,StartIndex)
/*
    Returns the index within the calling String object of the first occurrence of the specified value,
    starting the search at fromIndex. Returns -1 if the value is not found.

    Parameters:
      - Value: The substring to search for within the string.
*/
  str = "hello world";
console.log(str.indexOf('o')); // Output: 4
console.log(str.indexOf('l')); // Output: 2


//✨ -lastIndexOf(Value,StartIndex)
/*
    Returns the index within the calling String object of the last occurrence of
    the specified value, searching backwards from fromIndex. Returns -1 if the value is not found.
    Parameters:
      - Value: The substring to search for within the string.
*/
console.log(str.lastIndexOf('o')); // Output: 7
console.log(str.lastIndexOf('l')); // Output: 9

//✨ -slice(Start, [end])
/*
    Extracts a section of a string and returns it as a new string. You specify the start index and optionally the end index (exclusive).
    Parameters:
        - Start: The index where to begin the extraction (inclusive).
        - End: The index where to end the extraction (exclusive). If omitted, slice extracts to the end of the string.
*/
console.log(str.slice(6)); // Output: "world"
console.log(str.slice(0, 5)); // Output: "hello"

//✨ -repeat(Times) . [ES6]
/*
    Returns a new string containing the specified number of copies of the string on which it was called.
    Parameters:
        - Times: The number of times to repeat the string.
*/
let repeatedStr = "abc";
console.log(repeatedStr.repeat(3)); // Output: "abcabcabc"

```

```
//✨ -split(Separator [Opt])
/*
    Splits a string into an array of substrings based on the specified separator and returns the array.
    Parameters:
        - Separator (optional): Specifies the character(s) to use for separating the string.
        If omitted, the entire string will be returned as the only element in the array.
*/
let sentence = "This is a sentence";
console.log(sentence.split(' ')); // Output: ["This", "is", "a", "sentence"]

/*
-✨substring(start,end)
*/
// Returns a new string that includes characters from the original string between the specified start and end indexes (end index not included).
     str = "hello world";
    console.log(str.substring(1, 4)); // Output: "ell"

/*
-✨substr(start ,characters To Extract)
*/
// Returns the characters in a string beginning at the specified location through the specified number of characters.
    console.log(str.substr(1, 4)); // Output: "ello"

/*
-✨includes(value)
*/
// Determines whether a string contains the specified value. Returns true if the string contains the value, otherwise false.
    console.log(str.includes('world')); // Output: true

/*
-✨startsWith(value)
*/
// Determines whether a string begins with the characters of a specified string. Returns true if the string starts with the value, otherwise false.
    console.log(str.startsWith('hello')); // Output: true

/*
-✨endsWith(value)
*/
// Determines whether a string ends with the characters of a specified string. Returns true if the string ends with the value, otherwise false.
    console.log(str.endsWith('world')); // Output: true

//assignment : --[
let a="Elzero Web School";
//1 Zero
console.log(a.slice(2,6).charAt(0).toUpperCase() + a.slice(3,6));
// 2  HHHHHHH
console.log(('H').repeat(8)));
//3  ['Elzero']
console.log([a.substr(0,6)])
//4 Elzero School
console.log(a.substr(0,6) + " " + a.substr(-6))
```

```
480
481 // 5 :  eLZERO WEB SCHOOl
482 console.log(a[0].toLowerCase()  + a.slice(1,-1).toUpperCase()  + a[a.length-1].toLowerCase()
    );
483
484 //Comparison/Relational operators :-----------------------------[]
485 /*
486 == : compare value
487 === : compare value & datatype
488 != : different
489 !== : different value  &&  datatype
490 > : grater than
491 < : less than
492 >= : grater or equal
493 <= : less than or equal
494
495 */
496
497 //logical operators :-----------------------------[]
498 /*
499 -- !  : not
500 -- && : and
501 -- || : or
502 */
503 // if condition :-----------------------------[]
504 if(condition){
505 // code
506 }
507 else if(condition){
508 //code
509 }
510 else{
511  //code
512 }
513
514 // Ternary operator  :-----------------------------[]
515 //          [if true]   : [if price ]
516  condition ? expression1 : expression2;
517
518 // Nullish Coalescing Operator And Logical Or  -----------------------------[]
519 // LOGICAL OR || : --[ return backValue when  NULL + UNDEFINED + ANY FALSY VALUE :
520  varName= value || backValue;
521 // NULLISH COALESCING OPERATOR  :--[ return backValue when  NULL + UNDEFINED :
522  varName= value ?? backValue;
523
524 //Switch Statement : -----------------------------[]
525 switch (key) {
526     case value1:
527          //code
528         break;
529     case value2:
530         //code
531         break;
532
533     default:
534         //code
```

```
        break;
}
//Arrays : ------------------------------[]
/*
-Create Arrays  [Two Methods ]  new Array () |  []
-Access Arrays Elements
-Nested Arrays
-Change Arrays Elements
-Check For Array Array isArray(arr);
*/
//Arrays : ------------------------------[]

/*
-Create Arrays  [Two Methods ]  new Array () |  []
*/
// You can create arrays in JavaScript using two methods: using the Array constructor or using array literal notation.
let arr1 = new Array(); // Using Array constructor
let arr2 = []; // Using array literal notation

/*
-Access Arrays Elements
*/
// You can access individual elements of an array using square brackets [] and the index of the element. Indexing in arrays starts from 0.
let fruits = ["apple", "banana", "orange"];
console.log(fruits[0]); // Output: "apple"
console.log(fruits[1]); // Output: "banana"

/*
-Nested Arrays
*/
// Arrays can also contain other arrays, creating nested arrays.
let nestedArray = [[1, 2], [3, 4], [5, 6]];

/*
-Change Arrays Elements
*/
// You can change the value of an element in an array by assigning a new value to the corresponding index.
fruits[0] = "pear"; // Changing "apple" to "pear"

/*
-Check For Array Array.isArray(arr);
*/
// You can check if a variable is an array using the Array.isArray() method. It returns true if the variable is an array, otherwise false.
console.log(Array.isArray(fruits)); // Output: true
console.log(Array.isArray("apple")); // Output: false
//Arrays Methods : ------------------------------[]
// add element to the first :
arrName.unshift(value1,value2);

// add element to the end :
arrName.push();

// remove the first element L
```

```
arrName.shift();

// remove the last element
arrName.pop();

// get the index of a value  from first index to last :
arrName.indexOf(value,startIndex=0);

//get the index of a value from last index to first :
arrName.lastIndexOf(value,startIndex=0);

// check if value exist :
arrName.includes(value);

// sort Array :
arrName.sort();
// reverse an array :
arrName.reverse();

// get slice array  ;
 result=arrName.slice(start,end);

// remove || add  elements :
arrName.splice(startINdex,numberOfElementsToRemove,replacedValue1,replacedValueN);

// concat Arrays :
newArr=arr1.concat(arr2,[arr3]);

// convert from array to string :
str=arr.join(separator=',');

//assignment :
let zero = 0;
let counter = 3;

let my = ["Ahmed", "Mazero", "Elham", "Osama", "Gamal", "Ameer"];

// ["Osama","Elham","Mazero","Ahmed"];
console.log(my.reverse());

console.log(my.slice(-++counter));

// ["Elham","Mazero"]
console.log(my.slice(--counter, ++zero + ++counter));

// "Elzero"
console.log(my[--counter].substr(--zero, --counter) + my.reverse()[++zero].substring(++zero)
);
//my.reverse()[zero].substring(++zero)
console.log(counter, zero);
// rO;
console.log(my[--zero][my[zero].length - ++zero] + my[++zero].substring(counter - zero, --
counter));

// Loops :  ----------------------------[]
// for loop :
   // [label :]
```

```
    for(let i=start ; i<end; iteration){
        // code
        // [continue  | break label];
    }
//for in : index
for (let element of elements){
    //code
    // [continue | break]
}
// for of : element
for(let elementIndex of elements){
    //code
    // [continue | break]
}

// while loop :
    while(condition){
        // code
        // [continue | break]
    }
// do while loop :
    do{
    //code
    // [continue | break]
    }while(loop);

// foreach
    elements.foreach((element,index)=>{
    // code
    // return
    })

//assignment :------------------------------[]
// optimized version :
const body = document.body;
let myAdmins = ["Ahmed", "Osama", "Sayed", "Stop", "Samera"];
let myEmployees = ["Amged", "Samah", "Ameer", "Omar", "Othman", "Amany", "Samia"];

let adminNumber = myAdmins.slice(0, myAdmins.indexOf("Stop") == -1 ? 0 : myAdmins.indexOf("Stop")).length;
body.innerHTML = `<h3>we have ${adminNumber} Admins</h3>`;
for (i = 0; i < adminNumber; i++) {
    body.innerHTML += `<hr> <h4> the Admin For Team is <span style='color:red;'> ${myAdmins[i]} </span> </h4>`;
    body.innerHTML += `<h2>Team Members :</h2>`;

    let counter = 0;
    myEmployees.forEach((element) => {
        if (element.startsWith(myAdmins[i][0])) body.innerHTML += `<h4> -${++counter} ${element} </h4>`;
    });
}

//functions :------------------------------[]
function functionName(args){

    // [return value]
```

```
}
//function with variables
let functionName=function(args){



}

// arrow function :  (this = window always)--[
let functionName=()=>{

}
// call a function :--[
functionName(v1,v2,vn);
// default value
function functionName(arg1,arg2=defaultValue){ // if the user did not enter the value of the second parameter :



}

// reset Parameters :--[
function functionName(...args){
// code :
}
// example :
function sum(...values){
return values.reduce((prev,current)=>prev+current);
}
// assignment: ------------------------------[]
function showDetails(name,age,available){

    let strContent="Hello ";
     strContent+=(typeof name ==="string" ?  name : typeof  age =="string" ? age : typeof available =="string" ? available : "unknown") + ", Your Age Is ";
     strContent+=(typeof name ==="number" ?  name : typeof  age =="number" ? age : typeof available =="number" ? available : "unknown") + " Your Are ";
     let availableStatus= typeof name ==="boolean" ?  name : typeof  age =="boolean" ? age : typeof available =="boolean" ? available : "unknown";
     strContent+=(!availableStatus ? "Not " : "") + "Available For Hire"
      console.log(strContent);
    }

    /*
    Hello amine, Your Age Is 20 Your Are Not Available For Hire
    Hello amine, Your Age Is 20 Your Are Available For Hire
    Hello amine, Your Age Is 38 Your Are Not Available For Hire
    */
    showDetails("amine",20,false);
    showDetails(20,"amine",true);
    showDetails(false,"amine",38);

// function inside function example :  ------------------------------[]
    function sayMessage(fName, lName) {
        let message = `hello`;
        function concatMsg() {
            function getFullName() {
                return fName + " " + lName;
```

```
            }

            return `${message} ${getFullName()}`;
        }

        return concatMsg();
    }
    console.log(sayMessage("ayoub", "km"));

// add a function to Array type :  ------------------------------[]
    Array.prototype.functionName= function(){
        //code
    };

// map function : return new  array  with specific values :   ------------------------------
[]
    newArr=arr.map( (element,index,arr)=>{
    //code
    })
// map function implementation :--{
 let mapTest=function(callBack){
     const newArray=[];
     this.forEach(element => {

     newArray.push(callBack(element))
     });
     return newArray;
     }
     Array.prototype.mapTest = mapTest;

     // or --[

 Array.prototype.mapFunction=function(callback){
     var  newArray=[];
     for(element of this){
     newArray.push(callback(element));
     }
     return newArray;
     }
     arr1=[1,2,3,4,6,7,8];
     arr2=arr1.mapFunction(function(element){
     return element*10;
     });

// example : return newArr=arr*10 --[
        newArr=arr.map(element=> element *10);
// example : invented case :
    str="Good job";
    console.log(str.split("").map(ele=>ele === ele.toUpperCase() ? ele.toLowerCase( ) :
ele.toUpperCase()).join(""));
// example : inverted numbers :
    let numbers=[1,-1,5,-20];
    console.log(numbers.map(ele=> -ele ));

// filter function :  ------------------------------[]
// this function help us to copy  oldArray  to new array
 // with  specific  condition  :
```

```
 newArr=arr.filter( (element,index,arr)=>{
    return condition;
    })
// implementation :--[
    Array.prototype.filterFunction = function (callback) {
        let newArray = [];
        this.forEach(function (element) {
        if (callback(element)) newArray.push(element);
        });

        return newArray;
    };
//example :
    let  arr3=[1,3,4,5,6,7,8];
    let  newArray1=arr1.filterFunction(function(element){
    return element>5 ? true :false;

  });

// reduce :  ------------------------------[]
/*
    this function explore element by element
    in the  end the function will return a Number :

*/
    var total=arr.reduce(function(preValue,currentValue){
    // code
    });

//example : --[
     var arr=[1,3,4,5,6,8];
     var total=arr.reduce(function(preValue,currentValue){

     return preValue+currentValue;
     });
// assignment: ------------------------------[]
//1 :--[
let  theBiggest=["Bla","Propaganda","other","AAA","Battery","Test"];
let max=theBiggest.reduce((acc,current)=> acc.length > current.length ? acc:current);
console.log(max);
let removeChars=['E','@','@','L','Z','@','@','E','R','@','O'];
let mot=removeChars.filter(ele=> ele !='@').reduce( (acc,current)=>`${acc}${current}`);
console.log(mot);
//2 :--[
let myString="1,2,3,EE,l,z,e,r,o,_,W,e,b,_,S,c,h,o,o,l,2,0,z";
let newStr= myString.split(",").filter((ele,index)=> ele!=',' && index !=myString.split(",")
.length -1)
.map(ele=> ele == '_' ? " " : isNaN(ele) ? ele.charAt(0) : "" )
.reduce((acc,current)=>acc+current);
console.log(newStr);  // Elzero web School :

// objects : ------------------------------[]

/*
    In JavaScript, an object is a collection of key-value pairs, where each key is a string
(or Symbol)
    and each value can be any data type, including arrays, functions, and other objects.
```

```
*/
// syntax : --[
 objectName={
    key:value,
    key:value,
    functionName:function(){
        //code
    }
    // ....
    }
// or
 objectName= new Object({
    key:value,
    key:value,
    functionName:function(){
        //code
    }
    // ....
    })


// access to object :--[
 //Dot notation :
    objectName.key;
 // Bracket notation :
    objectName['key'];
 // example a : --[
 const person1 = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
  };

//example  b :--[
  let student = {
      FirstName: "ayoub",
      lastName: "majid",
      printObject: function() {
        console.log("the name ", this.FirstName);
        console.log("the lastName ",this.lastName);
      }
    };
    student.printObject();

// example c :--[
    // we can add a default value to an argument
  const obj = {
    name: "John",
    sayHelloArrow: () => {
      console.log("Hello, " + this.name); // `this` is lexically scoped (undefined in this
case)
    },
    sayHelloNormal: function() {
      console.log("Hello, " + this.name); // `this` is dynamically scoped (refers to obj)
    },
  };
```

```
915
916   obj.sayHelloArrow(); // Output: Hello, undefined
917   obj.sayHelloNormal(); // Output: Hello, John
918
919 //example  arrayOfObject  :--[
920       let student1= {
921         FirstName: "ayoub",
922         lastName: "majid",
923       };
924
925       let arr = [
926         {
927           FirstName: "ayoub",
928           lastName: "majid",
929         },
930
931         {
932           FirstName: "youness",
933           lastName: "majid",
934         },
935
936         {
937           FirstName: "adam",
938           lastName: "majid",
939         },
940         student1,
941       ];
942  // to access to one element you need to specified by index of array element :
943       element[index].attribute //...
944
945 //nested objects : ------------------------------[
946 /*
947     In JavaScript, nested objects refer to objects that are properties
948     of other objects, forming a hierarchical data structure.
949     This means that an object can contain other objects as values for its
950     properties.
951     These nested objects can themselves have properties and values,
952     creating a tree-like structure.
953     Here's an example of a simple nested object in JavaScript:
954     javascript
955     Copy code
956 */
957   const person = {
958     name: "John",
959     age: 30,
960     address: {
961       street: "123 Main St",
962       city: "New York",
963       zipCode: "10001"
964     }
965   };
966
967 //You can access properties of nested objects
968 //using dot notation or bracket notation:
969   console.log(person.name);                    // Output: "John"
970   console.log(person.address.city);           // Output: "New York"
```

```
  console.log(person["address"]["zipCode"]);  // Output: "10001"

// create a prototype object :
/*
    The Object.create() method in JavaScript is used to create a new object with a specified prototype object and optionally,
    with specified properties. It allows you to create an object that inherits properties and methods from another object,
    known as the prototype object.
*/
//syntax : --[
    Object.create(prototypeObject);
//example :--[
    person = {
        name: "John",
        age: 30,
        address: {
        street: "123 Main St",
        city: "New York",
        zipCode: "10001"
        }
    };

    obj = Object.create(person);
    console.log(obj);
    console.log(obj.name);
// Object.assign method: ------------------------------[]
//-- Purpose:
/*
    The `Object.assign()` method is used to copy the values of all enumerable own properties from
    one or more source objects to a target object.
    It returns the modified target object.
*/

// Syntax:--[
 Object.assign(target, ...sources)

// Parameters: --[
/*
     - target: The target object to which the properties will be copied.
     - sources: One or more source objects whose properties will be copied to the target object.
*/

// Return value:--[
// The modified target object after copying the properties from the source object(s).

// Example: --[
    let target = { a: 1, b: 2 };
    let source = { b: 3, c: 4 };

    // Copy properties from source to target
    Object.assign(target, source);

    console.log(target); // Output: { a: 1, b: 3, c: 4 }

```

```
//     Notes:
/*
    1. If the target object already has a property with the same name, its value will be overwritten by the value from the source object.
    2. If a source object is `null` or `undefined`, it will be ignored.
    3. `Object.assign()` only copies enumerable own properties from the source objects. It does not copy properties from the
        source's prototype chain.
    4. The properties are copied in the order they are listed in the `sources` parameter, with properties from later
        sources overwriting properties from earlier sources.
*/
// get object keys :
    Object.keys(obj);

// get object values :
    Object.values(obj);

// DOM: ------------------------------[]
// Purpose:
/*
    The DOM, or Document Object Model, is a programming interface for web
    documents. It represents the structure of an HTML document in a tree-like
    structure, where each node represents a part of the document, such as
    elements, attributes, and text.
*/

// Document:
/*
    The document is the root node of the DOM tree. It represents the entire
    HTML document and provides methods and properties for interacting with its
    contents.
*/

// Object:
/*
    Each element in an HTML document, such as <div>, <p>, <h1>, etc., is
    represented as an object in the DOM. These objects can be accessed and
    manipulated using JavaScript.
*/

// Model:
/*
    The DOM provides a structured representation of the document, allowing
    developers to programmatically access, modify, and manipulate the content,
    structure, and style of the document.
*/

/*
    The DOM is essential for dynamic web development because it allows
    JavaScript to interact with and modify the HTML content of a webpage in
    real-time. It enables tasks such as adding or removing elements, changing
    element attributes and styles, and responding to user events like clicks
    and keypresses.
*/

/*
```

```
    In summary, the DOM is a programming interface that allows JavaScript to
    interact with and manipulate the structure and content of HTML documents,
    making web pages dynamic and interactive.
*/
// get page title :
    console.log(document.title);
// get page body :
    console.log(document.body);
// get page forms collection :
    console.log(document.forms);
// get value of an input that has a name='one' from form 1 in the page
    console.log(document.forms[0].one.value);
// get page  links(a)  collection
    console.log(document.links);
// get href of link number 1 in the page :
    console.log(document.links[0].href);
// get page imgs
console.log(document.images);
// change the src of the first img in the page :
document.images[0].src="newSource";

// selectors: ------------------------------[]
// get element by id :
  var element=document.getElementById('idName');

// get elements by tag name :
    var elements = document.getElementsByTagName("TagName");

// get elements by class name :
    var elements=document.getElementsByClassName("className");

// get element by id or class or tag:
    var element = document.querySelector("[TagName] or [#idName] or [.className]");

// get elements by id or class or tag:
    var elements = document.querySelectorAll("[TagName] or [#idName] or [.className]");

// inner text as html to an  element (you can insert html element also ) :   ---------------
---------------[]
  element.innerHTML="innerHtmlWanted";

//inner text to an element : ------------------------------[]
    element.innerText="innerHtmlWanted";

// get content of element : ------------------------------[]
    console.log(element.innerHTML);

// get input value :  ------------------------------[]
    var input=document.querySelector("input");
    console.log(input.value);

// set attribute  :   ------------------------------[]
      element.setAttribute("attributeName","value");
      -- or
      element.attribute=value;
// example :
```

```
    let h1=document.querySelector("h2");
    let input=document.querySelector("input");
    let content=h1.innerHTML="hello world";
    input.setAttribute("value",content);

// get  value of an attribute : ----------------------------[]
    element.getAttribute("attributeName");
    --or
    element.attributeName;

// get all attributes :  ----------------------------[]
    element.attributes;

// check if element has an attribute :  ----------------------------[]
    element.hasAttribute('attributeName');

// check if element has any attribute :  ----------------------------[]
    element.hasAttributes();

// remove attribute :  ----------------------------[]
    element.removeAttribute("attributeName");

// style :  ----------------------------[]
 // set styling :
    element.style.styleName="value";
    -- or
    element.setProperty("propertyName",'value',"priority");

// remove style :
    element.style.styleName="unset";
    -- or
    element.removeProperty("propertyName");

//example 1:--[
    element.style.color="red";
//example 2:--{
    element.style.backgroundColor="black";

// cssText :
    element.style.cssText=`
    styling...
    `
//example : --[
     h1=document.querySelector("h1");

    h1.style.cssText=`
    background-color: red;
    color : green;
    padding :20px;
    text-align : center;

    `
// edit styling of style css file : -- [
    document.styleSheets[ccsFileNumber].rules[StylingBlockNumber].style.removeProperty("
property")
/*
```

```
     document.styleSheets : object contains all css files
     rules : contains all  css styling block  of a specific file :
     style access to the of a specific block :
*/
//example : --[
    document.styleSheets[0]
    // html :  <link rel="stylesheet" href="style.css">

    // css :
    ```
    //  rules[0]
        h1{
            background-color: red;
        //Property
        color : green;
        padding :20px;
        text-align : center;

        }
    // rules[1]
        h2{
            //Property
            color: green;
        }
    ```
    console.log(document.styleSheets[0].rules[0].style.removeProperty("color"))
    console.log(document.styleSheets[0].rules[1].style.removeProperty("color"))

// classes : ------------------------------[]
// add class List :
element.classList='cls1 cls2 clsN';
// --or:
    element.classList.add("cl1","cls2","clsN");

//replace a class :
  element.classList.replace("oldName","newName")

// remove a class :
  element.classList.remove("className");

// get the number of classes of an element :
    element.classList.length;

// check if a class  is exist
    element.classList.contains("className");

// get className using order :
    element.classList.item(index);

// add & remove in the same time , if class exist it will remove otherwise it will add it
element.classList.toggle("className");

//create element : ------------------------------[]
// create element  :
    let newElement=document.createElement('tagName');
// create textContent:
```

```
    var textContent=document.createTextNode("TextWanted");
// create attribute :
    var attribute=document.createAttribute("name");
// change value of attribute element :
    attribute.value='value';
// create comment :
    var comment=document.createComment("comment");

// append to an html element   :
   element.appendChild(element1);
// prepend child to an html element :
    element.prepend(element1)

// add an element before an element
    parentElement.insertBefore(newElement,referenceElement);
     // ref : the element you wanna insert before :
      // or:
      referenceElement.before(element);

// insert element after element :
    referenceElement.after(element);
// insert element in a position :
    refElement.insertAdjacentElement("position", newElement);
// insert html content :
    refElement.insertAdjacentHTML("position", HtmlContent);
// insert Text content :
    refElement.insertAdjacentText("position", HtmlContent);
// position ___________:
    "beforebegin" //Inserts immediately before the target element.
    "afterbegin"//Inserts as the first child of the target element.
    "beforeend"//Inserts  as the last child of the target element.
    "afterend" //Inserts  immediately after the target element.

// clone child  :
    copy=element.cloneNode([true]); // pass true if want to clone also the content

// Notes -- [
/*
    content :  (include spaces && comments)
    Element : (exclude spaces && comments)
*/

// know the parent of an element :
    element.parentNode;
    -- or
    element.parentElement;

// get the first content in  a div :
    element.firstChild;
// get the first element in a div
element.firstElementChild;

// get the last content in a div :
    element.lastChild
// get the last content in a div :
    element.lastElementChild
```

```
1300
1301 // get next sibling Element :
1302     element.nextElementSibling
1303 // get next sibling content
1304     element.nextSibling
1305
1306 // get  previous sibling Element :
1307     element.previousElementSibling
1308 // get previous sibling content
1309     element.previousSibling
1310
1311 // remove child :
1312     element.removeChild(liElement);
1313 // remove an element itself :
1314    element.parentNode.removeChild(element);
1315    // or
1316    element.remove();
1317
1318 // check if is the same node (same reference)
1319     element.isSameNode(element2);
1320 // check if the same content :
1321     element.isEqualNode(element2);
1322
1323
1324 //examples : -------[
1325 //example 1: --[
1326      myElement = document.createElement("div");
1327       h1=document.createElement("h1");
1328     h1.innerHTML='hello world';
1329
1330      myAttr = document.createAttribute("data-custom");
1331      text = document.createTextNode("Product One");
1332      myComment=document.createComment("this is div");
1333
1334     myElement.className = "product";
1335     myElement.setAttributeNode(myAttr);
1336     myElement.appendChild(text);
1337     console.log(myElement);
1338
1339     myAttr.value='nice';
1340     console.log(myAttr);
1341     //append comment to element :
1342     myElement.prepend(myComment);
1343     myElement.append(h1);
1344     document.body.appendChild(myElement);
1345 //example 2:--[
1346     // Create productContainer, title, and details elements outside the loop
1347     let productContainer = document.createElement("div");
1348     var title = document.createElement("h1");
1349     let details = document.createElement("p");
1350
1351     productContainer.className = "product";
1352     // Set initial content for details element
1353     details.textContent = `Amazing product you should try it at home.`;
1354
1355     const productNumber = 100;
```

```
1356
1357        // Clone productContainer, title, and details elements within the loop
1358        for (let i = 1; i <= productNumber; i++) {
1359            let productClone = productContainer.cloneNode(true); // Clone productContainer and
    its descendants
1360            let titleClone = title.cloneNode(true); // Clone title and its descendants
1361            let detailsClone = details.cloneNode(true); // Clone details and its descendants
1362
1363            // Set unique content for title element
1364            titleClone.textContent = `Product ${i}`;
1365
1366            // Append cloned elements to productContainer clone
1367            productClone.classList.add(`products${i}`);
1368            productClone.appendChild(titleClone);
1369            productClone.appendChild(detailsClone);
1370
1371            // Append productContainer clone to document body
1372            document.body.appendChild(productClone);
1373        }
1374
1375        document.head.innerHTML += `
1376        >
1377        .product{
1378            <style>
1379                .product{
1380                    padding: 10px;
1381                    padding-left: 15px;
1382                    border-radius: 12px;
1383                    box-shadow: 1px 5px 10px rgba(0,0,0 ,0.3);
1384                    margin-bottom: 20px;
1385                    width: 80%;
1386                    margin: 30px auto;
1387                    border-left: 5px solid #262643;
1388                }
1389            </style>
1390        }`;
1391    // assignment video 100 : ------------------------------[]
1392    // js ---[ :
1393    let body = document.querySelector("body");
1394
1395    //start upDown animation ---------------------------- :
1396    function getAnimation() {
1397        let parentDiv = document.createElement("div");
1398        parentDiv.classList.add("parent");
1399        let h3Anim = document.createElement("h3");
1400        h3Anim.textContent = "Build By Ayoub Majid";
1401        let pointsDiv = document.createElement("div");
1402        pointsDiv.classList.add("points");
1403
1404        for (let i = 1; i <= 3; i++) {
1405            let tempPointDiv = document.createElement("div");
1406
1407            tempPointDiv.classList.add(`point${i}`);
1408            pointsDiv.appendChild(tempPointDiv);
1409        }
1410
```

```
    // append elements to parent div :
    parentDiv.appendChild(h3Anim);
    parentDiv.appendChild(pointsDiv);

    return parentDiv;
}

function pushAnimationToDom() {
    let startPointsComment = document.createComment("Start upDown animation ");
    let endPointsComment = document.createComment("End upDown animation ");

    body.appendChild(startPointsComment);
    body.appendChild(getAnimation());
    body.appendChild(endPointsComment);
}

pushAnimationToDom();

var parent = document.querySelector(".parent");

function delay(milliseconds) {
    return new Promise((resolve) => setTimeout(resolve, milliseconds));
}
function scrollToTop() {
    document.documentElement.scrollTop = 0; // For modern browsers
    document.body.scrollTop = 0; // For older browsers
}

async function upDownAnim() {
    scrollToTop();
    parent.style.display = "flex";
    for (let i = 1; i >= 0; i -= 0.01) {
        parent.style.opacity = i;
        await delay(24);
    }
    parent.style.display = "none";
}
upDownAnim();

// start header ---------------------------- :
function getHeader() {
    let header = document.createElement("header");
    let nav = document.createElement("nav");
    nav.classList.add("navBar");
    let h1Header = document.createElement("h1");
    h1Header.classList.add("title");
    h1Header.textContent = "ELZERO";

    let ulHeader = document.createElement("ul");
    let arrLiList = [];
    let arrLiContent = ["Home", "About", "Services", "Contact"];
    for (let i = 0; i < 4; i++) {
        arrLiList.push(document.createElement("li"));
        arrLiList[i].textContent = arrLiContent[i];
    }
    arrLiList.forEach((li) => {
```

```
        ulHeader.appendChild(li);
    });

    nav.appendChild(h1Header);
    nav.appendChild(ulHeader);

    header.appendChild(nav);

    return header.cloneNode(true);
}
function pushHeader() {
    let startComment = document.createComment("start header");
    let endHeader = document.createComment("end header");
    body.appendChild(startComment);
    body.appendChild(getHeader());
    body.appendChild(endHeader);
}
pushHeader();
// start main content ---------------------------- :

function getMainContent() {
    let main = document.createElement("main");
    let products = document.createElement("div");
    products.classList.add("products");

    let product = document.createElement("div");

    product.classList.add("product");
    let h1Product = document.createElement("h1");
    h1Product.classList.add("ProductNumber");
    let pProduct = document.createElement("p");
    pProduct.textContent = "Product";
    product.appendChild(h1Product);
    product.appendChild(pProduct);

    for (let i = 1; i <= 50; i++) {
        let tempProduct = product.cloneNode(true);
        tempProduct.firstChild.textContent = i;
        products.appendChild(tempProduct);
    }
    return products;
}
function pushProductsToDom() {
    let startMainContentComment = document.createComment("start main Content");
    let endMainContentComment = document.createComment("end main Content");
    body.appendChild(startMainContentComment);
    body.appendChild(getMainContent());
    body.appendChild(endMainContentComment);
}
pushProductsToDom();

// start footer  ---------------------------- :

function getFooter() {
    let footerDom = document.createElement("footer");

```

```
    let h2Footer = document.createElement("h2");
    h2Footer.innerHTML = "Copyright &copy 2023";

    footerDom.appendChild(h2Footer);

    return footerDom;
}

function pushFooterToDom() {
    let startFooterComment = document.createComment("start Footer");
    let endFooterComment = document.createComment("end Footer");

    body.appendChild(startFooterComment);
    body.append(getFooter());
    body.appendChild(endFooterComment);
}
pushFooterToDom();

// css : --[
```
    /* start default  parameters  */
    * {
        padding: 0;
        margin: 0;
        box-sizing: border-box;
    }

    ul, li {
        list-style: none;
    }

    a {
        text-decoration: none;
    }

    input, button {
        outline: none;
    }

    /* end default  parameters  */

    /* start components   */
    .container {
        padding-left: 10px;
        padding-right: 10px;
        margin-left: auto;
        margin-right: auto;

    }

    /* small */
    @media (min-width:768px) {
        .container {
            width: 750px;
        }
    }
```
```

```
1579
1580      /* medium*/
1581      @media (min-width:992px) {
1582          .container {
1583              width: 970px;
1584          }
1585      }
1586
1587      /* large*/
1588      @media (min-width:1200px) {
1589          .container {
1590              width: 1170px;
1591          }
1592      }
1593
1594      /* end components   */
1595
1596      /*  start ui color   */
1597      :root {
1598          --primaryColor: #5e9b8b;
1599          --primaryBg: #eeedef24;
1600          --secondaryBg: #ffffff;
1601      }
1602
1603      /*  end ui color   */
1604      /* start scrollbar */
1605      ::-webkit-scrollbar {
1606          width: 5px;
1607      }
1608
1609      ::-webkit-scrollbar-thumb {
1610          background: var(--primaryColor);
1611      }
1612
1613      /* end scrollbar */
1614      /* start body   */
1615      body {
1616          font-family: "Rubik", sans-serif;
1617          font-optical-sizing: auto;
1618          font-style: normal;
1619      }
1620
1621      /* end body  */
1622      /* Start upDown animation */
1623      .parent {
1624          position: fixed;
1625          width: 100%;
1626          height: 100vh;
1627
1628          display: flex;
1629          flex-direction: column;
1630          justify-content: center;
1631          align-items: center;
1632          gap: 100px;
1633          background-color: rgba(0, 0, 0, 0.892);
1634          z-index: 100;
```

```
        display: flex;
        transition: opacity 0.3s;
        flex-wrap: wrap;
        z-index: 99999;
    }

    .points {
        display: flex;
        justify-content: center;
        gap: 30px;
    }

    .parent h3 {
        color: #2c4755;
        margin-left: 20px;
        font-size: 2.3em;
        font-weight: 800;

    }

    @media (max-width:768px) {

        .parent h3 {
            font-size: 2em;
            text-align: center;
        }
    }

    .points div {
        width: 50px;
        height: 50px;
        border-radius: 50%;
        background-color: var(--primaryColor);
        animation-name: upDown;
        animation-duration: 2s;
        animation-iteration-count: 10;
        animation-direction: alternate;


    }

    .parent .point2 {
        animation-delay: 0.3s;
    }

    .parent .point3 {
        animation-delay: 0.6s;
    }

    @keyframes upDown {

        to {
            opacity: 0.6;
            transform: translateY(-40px);
        }
    }
```

```
1691
1692      /* End upDown animation */
1693
1694      /* start header  */
1695      header {
1696          padding: 5px 10px;
1697          box-shadow: 1px 5px 10px rgba(0, 0, 0, 0.03);
1698      }
1699
1700      header nav {
1701          align-items: center;
1702          display: flex;
1703          justify-content: space-between;
1704          padding: 10px;
1705      }
1706
1707      nav .title {
1708          color: var(--primaryColor);
1709          margin-right: 50px;
1710      }
1711
1712      nav ul {
1713          display: flex;
1714          min-width: 30%;
1715          max-width: 70%;
1716          justify-content: space-between;
1717          gap: 30px;
1718          font-size: 1.04rem;
1719      }
1720
1721      ul li {
1722          transition: 0.3s;
1723          padding: 12px 10px;
1724          transition: 0.4s cubic-bezier(0.165, 0.84, 0.44, 1);
1725          position: relative;
1726      }
1727
1728      ul li::after {
1729          content: "";
1730
1731          position: absolute;
1732          bottom: -4px;
1733          left: 50%;
1734          transform: translateX(-50%);
1735          height: 2px;
1736          background-color: var(--primaryColor);
1737          transition: 0.3s;
1738          width: 0;
1739
1740      }
1741
1742      ul li:hover {
1743          color: var(--primaryColor);
1744          transform: translateY(-2px);
1745      }
1746
```

```
    ul li:hover::after {
        width: 90%;
    }

    /* end header  */
    /* start main content  */
    main {
        display: flex;
        flex-direction: column;
        justify-content: space-between;
        margin-bottom: 60px;

    }

    /* start products  */
    .products {
        margin-top: 50px;
        padding: 20px;
        display: grid;
        grid-template-columns: repeat(auto-fit, minmax(240px, 1fr));
        row-gap: 50px;
        column-gap: 30px;
        background-color: var(--primaryBg);

    }

    .products .product {
        background-color: var(--secondaryBg);
        text-align: center;
        padding: 26.5px 20px;
        border-radius: 13px;
        font-size: 1.2rem;
        box-shadow: 0px 10px 10px rgba(3, 36, 13, 0.045);
        transition: 0.4s;
    }

    .products .product:hover {

        transform: translateY(-4px);
    }

    .product .ProductNumber {
        margin-bottom: 16px;
        color: var(--primaryColor);
    }

    .product p {
        color: #3e3e3e;
    }

    /* end products  */
    /* start footer  */
    footer {
        position: fixed;
        width: 100%;
        bottom: 0;
```

```
        background-color: rgb(31, 114, 23);
        padding: 3px;
        text-align: center;
        color: white;
    }

    footer h2 {
        padding: 7px;
        text-shadow: 1px 2px 10px rgba(0, 0, 0, 0.503);
    }

    /* end footer  */

    /* end main content  */
```
```
// html : --[
```
```
    <!DOCTYPE html>
    <html lang="en">
        <head>
            <meta charset="UTF-8" />
            <meta name="viewport" content="width=device-width, initial-scale=1.0" />
            <title>Document</title>

            <!-- google fonts  -->
            <link rel="preconnect" href="https://fonts.googleapis.com" />
            <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
            <link href="https://fonts.googleapis.com/css2?family=Rubik:ital,wght@0,300..900;1,300..900&display=swap" rel="stylesheet" />

            <link rel="stylesheet" href="style.css" />
        </head>

        <body>
            <script src="script.js"></script>

            <style></style>
        </body>
    </html>

```
```
// Events : ------------------------------[]
//type of events :
      // 1. **Mouse Events:**
      /*
         ✔️- `click` - Occurs when the mouse is clicked.
         ✔️- `dblclick` - Occurs when the mouse is double-clicked.
         ✔️- `mousedown` - Occurs when a mouse button is pressed down.
         ✔️- `mouseup` - Occurs when a mouse button is released.
         ✔️- `mousemove` - Occurs when the mouse pointer moves.
         ✔️- `mouseover` - Occurs when the mouse pointer enters an element.
         ✔️- `mouseout` - Occurs when the mouse pointer leaves an element.
      */
      // 2. **Keyboard Events:**
      /*
         ✔️ - `keydown` - Occurs when a key is pressed down.
```

```
        ✔ - `keyup` - Occurs when a key is released.
        ✔ - `keypress` - Occurs when a key is pressed.
      */
      // 3. **Form Events:**
      /*
        ✔- `submit` - Occurs when a form is submitted.
        ✔- `reset` - Occurs when a form is reset.
        ✔- `change` - Occurs when the value of an input element changes.
        ✔- `input` - Occurs when the content of an input element changes.
      */
      // 4. **Window Events:**
      /*
        ✔- `load` - Occurs when a webpage has finished loading.
        ✔- `unload` - Occurs when a user leaves a page.
        ✔- `resize` - Occurs when the browser window is resized.
        ✔- `scroll` - Occurs when the user scrolls in the window.
      */
      // 5. **Document Events:**
      /*
        ✔ - `DOMContentLoaded` - Occurs when the HTML document has been completely loaded
and
            parsed without waiting for stylesheets, images, and subframes to finish loading.
        ✔ - `readystatechange` - Occurs when the readyState property of the document
changes.
      */
      // 6. **Focus Events:**
      /*
        ✔- `focus` - Occurs when an element gains focus.
        ✔- `blur` - Occurs when an element loses focus.
      */
      // 7. **Touch Events:**
      /*
        ✔ - `touchstart` - Occurs when a touch is initiated on a touch-enabled device.
        ✔ - `touchmove` - Occurs when a touch point is moved along the touch surface.
        ✔ - `touchend` - Occurs when a touch point is removed from the touch surface.
      */

// add event :
    element.onEvent=function(){
        //code
    }
    -- or
        element.addEventListener("event",function(){
        //code

        });
    -- or
        //html code :
        /*
            <tagName onEvent="functionName() or JavaScriptCode" >
        */
        //js code :
        function functionName(){
        // code
        }
//event object -------------------------[
/*
```

```
    The event object in JavaScript is a special object that contains information about an
    event when it occurs, such as a user
    action (e.g., a mouse click, a key press, or a form submission). The event object
    is automatically passed as an argument
    to the event handler function when an event occurs, and it provides details about the
    event, such as the type of the event,
    the target element, and additional information specific to the event type.
*/
// ✨1. **`event.type`:**
    //  - Returns a string representing the type of the event ( "click", "keydown",...).

// ✨2. **`event.target`:**
    //  - Returns the DOM element that triggered the event.

// ✨3. **`event.currentTarget`:**
    //  - Returns the current DOM element that is handling the event.
//example : --[
<div id="outerDiv">
        <div id="innerDiv">
            Click me!
        </div>
    </div>
    document.getElementById('outerDiv').addEventListener('click', function(event) {
        // currentTarget refers to the element that the event listener is attached to
        console.log('Current Target: ' + event.currentTarget.id);

        // target refers to the actual element that triggered the event
        console.log('Target: ' + event.target.id);
    });

// ✨4. **`event.preventDefault()`:**
/*
  - Prevents the default behavior associated with the event. For example,
    preventing the default action of a form submission or a hyperlink click.
*/
// ✨5. **`event.stopPropagation()`:**
/*
  - Stops the event from propagating up or down the DOM hierarchy. It prevents the event
  from reaching other event listeners on ancestor or descendant elements.
*/
//*example
    <div id="outerDiv">
        <div id="innerDiv">
            Click me!
        </div>
    </div>
        document.getElementById('innerDiv').addEventListener('click', function(event) {
            // This prevents the event from reaching the outer div
            event.stopPropagation();

            alert('Inner div clicked!');
        });

        document.getElementById('outerDiv').addEventListener('click', function() {
            alert('Outer div clicked!');
        });
```

```
1969
1970 // ✨6. **`event.key` and `event.code`:**
1971 /*
1972     For keyboard events, `event.key` returns the value of the pressed key
1973     (e.g., "Enter," "A") and `event.code` returns a string representing the physical
1974     key on the keyboard (e.g., "Enter", "KeyA").
1975 */
1976 // ✨7. **`event.clientX` and `event.clientY`:**
1977 /*
1978     For mouse events, these properties provide the X and Y coordinates
1979     of the mouse pointer relative to the viewport.
1980 */
1981 //✨ 8. **`event.preventDefault()`:**
1982 /*
1983     Prevents the default action associated with the event. For example,
1984     preventing the submission of a form or the opening of a link.
1985 */
1986
1987 // Validate form example :
1988     document.forms[0].onsubmit = function (event) {
1989         let userValid = false;
1990         let ageValid = false;
1991
1992         if (this.username.value != "" && this.username.value.length <= 10) userValid = true;
1993         if (Number.isInteger(+this.age.value) && this.age.value >= 18) ageValid = true;
1994
1995         if (userValid === false || ageValid === false) {
1996             event.preventDefault();
1997         }
1998     };
1999 // Dom [Event Simulation]
2000 /*
2001     -click
2002     -focus
2003     -blur
2004 */
2005 //syntax  :
2006 element.onEvent();
2007 // example : --[
2008     let form=document.forms[0];
2009
2010     window.onload=function(event){
2011     form.age.focus();
2012
2013     }
2014     form.username.onblur=function(){
2015
2016     document.links[0].click();
2017     }
2018
2019
2020 // Add event to dynamic content : -------------------------[]
2021 // Add event listener to the addButton to dynamically add elements
2022 document.getElementById('addButton').addEventListener('click', function() {
2023     const newElement = document.createElement('div');
2024     newElement.textContent = 'Dynamically added element';
```

```
    newElement.classList.add('dynamic-element');
    document.getElementById('container').appendChild(newElement);
});

// Add event listener to a parent element that will contain dynamically added content
document.getElementById('container').addEventListener('click', function(event) {
    // Check if the clicked element matches a specific selector
    if (event.target.matches('.dynamic-element')) {
        // Your event handling logic here
        console.log('Dynamic element clicked');
    }
});
//event.target.matches :
/*
    you can use any valid CSS selector string with the matches() method to check if an element matches the specified selector.
    This includes class selectors (.className), ID selectors (#idName), attribute selectors ([attributeName="value"]),
    tag selectors (tagName), pseudo-class selectors (:pseudo-class), and more.
*/
//Example : -------------------------[
    let myP=document.querySelector("h1");

    myP.addEventListener('click' ,function () {
        let newP = myP.cloneNode(true);
        newP.className = "clone";
        document.body.appendChild(newP);
    });


    document.addEventListener("click", function (e) {
        if (e.target.className === "clone") {
        e.target.addEventListener('click',function(){
            console.log("clone Element ");
        })
        }
    });

// remove event listener :  -------------------------[]
    element.removeEventListener('click', callbackFunction)
// example : --[
    // Define the event listener function
    function clickHandler() {
        let newP = myP.cloneNode(true);
        newP.className = "clone";
        document.body.appendChild(newP);
    }

    // Add the event listener to myP
    myP.addEventListener('click', clickHandler);

    // Later, when you want to remove the event listener
    myP.removeEventListener('click', clickHandler);

// BOM [Browser Object Modal] : -------------------------[]
//  show a pop up window with a message  need No response only ok Available
```

```
    [window.]alert("message")
```
//show pop up window with  a message : need response and return a  boolean  ok=true & cancel=false
    var status=confirm("message  :");

// show pop up window with a message : collect Data
    var data=prompt('msg','defaultMsg');

// *setTimeout -------------
  /*
  this function pause the execution of  the code
  until we get a specific time  (arrow function ) :
  */
  setTimeout(() => {
    // code
}, timeout); // time with ms
//clear interval :
clearInterval(intervalName);

// BOM :[browser object modal]-------------------------[]

// window.location object : -------------------------[

// set / get the link  of the page with saving session history :
/*
    if you pass the short link it will just change the page name
    oldPat/newHrefLink
*/
    window.location.href;

//set / get the hostname (ip address of the server )
    window.location.hostname;
//set /get  the host= [hostname  + portNumber] (ip of server + port)
    window.location.host;

// get /set the protocol of the page (http  / https)
    location.protocol;

// reload the page :
    location.reload();

// replace the link  of the page with removing the old link from
//session   history :
    location.replace("newLink")

// set  the link  of the page with saving session history :
    location.assign("newLink");
// window functions : -------------------------[
// open a window :
window.open("url","_self | _blank","window style(put values without px)", wannaSaveThePreviousInHistory);
// example :
    setInterval(() => {

        window.open('https://google.com','','width:200px;height:200px');
    }, 200);

```
2135
2136 // close the window
2137     window.close();
2138
2139 // history object  -------------------------[
2140     // show history link :
2141         history.length;
2142
2143     // get back to previous history page :
2144         history.back();
2145
2146     // get forward to next history page :
2147         history.forward();
2148
2149     // get back/forward  base on the position :
2150     /*
2151         position 0 reload the page :
2152     */
2153         history.go(position);
2154
2155 // print the page :
2156 window.print();
2157
2158 // focus at a window :
2159 window.focus();
2160 //example :
2161 let  newWindow=document.open('www.google.com','_blank','width :300;height 500; ')
2162 newWindow.focus();
2163
2164 //scroll to a position :
2165 scrollToTop(xValue,yValue);
2166 scroll(xValue,yValue);
2167
2168 // scroll to a position with base on the same position
2169 //in the next scroll
2170 scrollBy(xValue,yValue);
2171
2172
2173 // scroll function with  more arguments
2174 window.scrollTo({
2175     left :value,
2176     top :value,
2177     behavior:"smooth | instance | auto"
2178     })
2179
2180
2181 // get the scroll top position :
2182     scrollTop=window.scrollY;
2183     window.pageXOffset // old browser
2184
2185 // get the window height :
2186     windowHeight = window.innerHeight;
2187     window.pageYOffset // old browser
2188
2189 // get the total height of page :
2190     totalHeight = document.documentElement.scrollHeight;
```

```
2191
2192 // check if get the end of the page :
2193     if (scrollTop + windowHeight >= totalHeight);
2194
2195 // get url params :
2196 // method 1 :
2197     // Function to get URL parameters
2198     function getUrlParams() {
2199         // Create a new URL object with the current URL
2200         const url = new URL(window.location.href);
2201
2202         // Return the searchParams object, which contains the parameters
2203         return Object.fromEntries(url.searchParams.entries());
2204     }
2205 // method 2 :
2206     // Function to get URL parameters
2207     function getUrlParams() {
2208         // Create a new URLSearchParams object with the URL query string
2209         const queryParams = new URLSearchParams(window.location.search);
2210
2211         // Create an empty object to store the parameters
2212         const params = {};
2213
2214         // Loop through each parameter and store it in the object
2215         for (const [key, value] of queryParams.entries()) {
2216             params[key] = value;
2217         }
2218
2219         return params;
2220     }
2221
2222  // local storage : -------------------------[]
2223 // check if if key exist :
2224     if (!object.hasOwnProperty("key")){}
2225
2226 // store primitive data type in local storage :
2227   localStorage.setItem('key',value);
2228   localStorage.key=value;
2229   localStorage['key']=value;
2230 // store no primitive data type in local storage :
2231     localStorage.setItem('key',JSON.stringify(value));
2232
2233 //get primitive data type in local storage :
2234     result=localStorage.setItem('key');
2235 // get no primitive data type in local storage :
2236     result=JSON.parse(localStorage.setItem('key'));
2237
2238 //remove item from  local storage :
2239     localStorage.removeItem('key');
2240     delete localStorage.key
2241
2242 //clear local storage :
2243     localStorage.clear();
2244
2245 // get key by index ;
2246     localStorage.key(index);
```

```
// dataset  : -------------------------[]
/*
 it's a way to set some information about an html an element
 to access getting using js dataset
*/
// example :
// html
```
    <button data-color="red" data-background="green">stop</button>
```
// js
    let  btn=document.querySelector('button');
    console.log(btn.dataset);


// session storage :
```
    Local Storage and Session Storage are both mechanisms provided by web browsers to store key-value
    pairs locally within the user's browser. However, they have some differences in terms of scope,
    lifetime, and usage:
```
// 1. **Scope:**
/*
    - **Local Storage:** Data stored in local storage persists even after the browser is closed
        and reopened. It is scoped to the origin (i.e., the combination of protocol, host,
        and port of the website). This means that data stored in local storage for one
        website is accessible to that same website even if the user navigates away and
        comes back later.
    - **Session Storage:** Data stored in session storage is only available for the
    duration of the page session. It is scoped to the tab or window in which the page
        is opened. Once the tab or window is closed, the data is cleared and no longer accessible.
*/
// 2. **Lifetime:**
/*
   - **Local Storage:** Data stored in local storage persists indefinitely until explicitly removed
        by the web application or cleared by the user through browser settings.
   - **Session Storage:** Data stored in session storage persists only for the duration of the page
       session. If the user navigates to another page within the same website or closes the tab/window,
       the session storage data is cleared.
*/
// 3. **Usage:**
/*
   - **Local Storage:** Local storage is often used for storing data that needs to be preserved across browser
       sessions, such as user preferences, cached data, or application state.
   - **Session Storage:** Session storage is useful for storing temporary data that is only relevant
       for the current browsing session, such as data needed for multi-step processes or data that should not
       persist beyond the current session.
```

```
*/
/*
    In summary, local storage provides persistent storage across browser sessions and is scoped to the origin,
    while session storage provides temporary storage for the duration of a page session and is scoped to the tab
    or window. The choice between them depends on the specific requirements of the web application and the
    desired behavior of the stored data.
*/

/*
    info :
    -new tab = new session
    -duplicate Tab = copy session
    -new tab with same url = new session
*/

// store primitive data type in session storage :
  sessionStorage.setItem('key',value);
  sessionStorage.key=value;
  sessionStorage['key']=value;
// store no primitive data type in session storage :
    sessionStorage.setItem('key',JSON.stringify(value));

//get primitive data type in session storage :
    result=sessionStorage.setItem('key');
// get no primitive data type in session storage :
    result=JSON.parse(sessionStorage.setItem('key'));

//remove item from  session storage :
    localStorage.removeItem('key');
    delete sessionStorage.key

//clear session storage :
    sessionStorage.clear();

// get key by index ;
    sessionStorage.key(index);

// Destructuring Assignment :--------------------------[]
 /*
  Destructuring assignment is a feature introduced in ES6 (ECMAScript 2015) that allows you to
  extract values from arrays or properties from objects into distinct variables.
  It provides a concise and
  convenient way to unpack values from complex data structures like arrays and objects.
 */
// Array Destructuring: -------------------------[
   /*
    In array destructuring, you can extract elements from an array and assign them to variables
    in a single line. The syntax uses square brackets [] on the left side of the assignment.
   */
    // Array with values
     numbers = [1, 2, 3, 4, 5];

```

```
// Destructuring assignment
    const [first, second, , fourth] = numbers;

    console.log(first);   // Output: 1
    console.log(second);  // Output: 2
    console.log(fourth);  // Output: 4

    //In object destructuring, you can extract specific properties
    //from an object and assign them to variables.
    // The syntax uses curly braces `{}` on the left side of the assignment.

// object Destructuring : -------------------------[
    /*
      Destructuring assignment is a powerful feature that makes working
      with arrays and objects more concise and expressive,
      especially when dealing with complex data structures.
      It is widely used in modern JavaScript applications.
    */
    var { firstName, age } = person1;
      console.log(firstName);  // Output: "John"
      console.log(age);        // Output: 30

//You can also provide default values for variables in case the property
    // or array element does not exist:
      const Numbers = [1, 2];
      var [A, b, c = 3] = Numbers;

      console.log(A); // Output: 1
      console.log(b); // Output: 2
      console.log(c); // Output: 3 (default value)

//You can also rename variables during destructuring using a colon `:`:
      const person2 = {
        firstName: "John",
        lastName: "Doe",
        age: 30,
      };

      var { firstName: fName, lastName: lName } = person2;

      console.log(fName); // Output: "John"
      console.log(lName); // Output: "Doe"
// you can use variables already declared :
    const user={
        firstName :"ayoub",
        lastName:"majid",
        age :20 ,
        skills : {
        html :"html",
        css :"css"
        }
    }
        var firstName,lastName;

        ({firstName,lastName}=user);
// function destructuring :
```

```
    user = {
        firstName: "ayoub",
        lastName: "majid",
        age: 20,
        skills: {
            html: "html",
            css: "css",
        },
    };

    function showUser({ firstName, lastName, skills: { css } } = user) {
        console.log(`your name is ${firstName}`);
        console.log(`your  lastName is ${lastName}`);
        console.log(`your skills is ${css}`);
    }

    showUser(user);

// mix content destructuring :
user = {
    firstName: "ayoub",
    lastName: "majid",
    age: 20,
    skills: ["html","css"],
    address:{
     morocco :"Rabat",
     egypt:"cairo"
    }
};
const {firstName,lastName,age,skills:[html,css],address:{morocco,egypt}}=user;

// assignment : --[
    let chosen =3;
    let myFriends=[
    {title:"nice",age:30,available:true,skills:["html","css"]},
    {title:"amine",age:23,available:false,skills:["python","django"]},
    {title:"ayoub",age:20,available:true,skills:["node js","express"]}
    ]

    if(chosen==1){
    var [{title,age,available,skills:[,skill2]},,]=myFriends;
    }

    if(chosen==2){
    var [,{title,age,available,skills:[,skill2]}]=myFriends;
    }

    if(chosen==3){
    var [,,{title,age,available,skills:[,skill2]}]=myFriends;
    }
    console.log(`title ${title}`);
    console.log(`age ${age}`);
    console.log(`${available ? "available" : "Not available"}`);
    console.log(`skills 2 ${skill2}`);

// set data type : -------------------------[]
```

```
    // create new set :
    const set = new Set([1, 2, 3]);
      // or
       let data=[1,2,3];
        set= new Set(data);
      // or
      set= new Set().add(1).add(2).add(3);

    // add new value :
    set.add(5);
    // you can add duplicate values :
    set.add(5);

    // list a set :
    for (element of set) {
        console.log(element);
    }

    // check if value exist :
    console.log(set.has(4));

    // delete a value of set : return [true : value exist /false : not exist]
    set.delete(3);

    // get the length of the set :
    console.log("the size of the set  is : ", set.size);

    // clear a set :
    set.clear();

    //list using foreach :
    set.forEach((element) => {
        console.log("the value: ", element);
    });


// weakset dataType :  works just with objects  -------------------------[]
// create a weakset :
    var  weak= new WeakSet([{ A: 1, B: 2 }]);
// add new value :
   set.add(5);

// check if value exist :
    console.log(set.has(4));

// delete a value of set : return [true : value exist /false : not exist]
    set.delete(3);
// - Set vs WeakSet --[
/*
    "
        The WeakSet is weak,
        meaning references to objects in a WeakSet are held weakly.
        If no other references to an object stored in the WeakSet exist,
        those objects can be garbage collected.
    "
    --
```

```
    Set     => Can Store Any Data Values
    WeakSet => Collection Of Objects Only
    --
    Set     => Have Size Property
    WeakSet => Does Not Have Size Property
    --
    Set     => Have Keys, Values, Entries
    WeakSet => Does Not Have clear, Keys, Values And Entries
    --
    Set     => Can Use forEach
    WeakSet => Cannot Use forEach

    Usage: Store objects and removes them once they become inaccessible
*/
// map data type : -------------------------[]

// Create a new Map
    let myMap = new Map();
    // or
    myMap= new Map([
        [key,value],
        [key,value]
    ])

// Add key-value pairs to the Map
    myMap.set('name', 'John');
    myMap.set(1, 'One');
    myMap.set(true, 'True');

// Get values from the Map
    console.log(myMap.get('name')); // Output: John
    console.log(myMap.get(1));      // Output: One
    console.log(myMap.get(true));   // Output: True

// Iterate over key-value pairs using forEach
    myMap.forEach((value, key) => {
    console.log(`${key} : ${value}`);
    });
    // Output:
    // name : John
    // 1 : One
    // true : True

// Get the number of key-value pairs
    console.log(myMap.size); // Output: 3

// Check if a key exists
    console.log(myMap.has('name')); // Output: true

// Delete a key-value pair
    myMap.delete(1);
    console.log(myMap.size); // Output: 2

// real using  -- [
/*
    One real-world scenario where Maps are commonly used is in web development, particularly
```

```
when dealing
    with data manipulation and management within applications. Here's an example of how Maps can be used in a web application:

    Let's consider a scenario where you're building a web application for managing user accounts.
    Each user has a unique user ID, and you need to store various information about each user, such as their
    name, email, and age.
*/
    // Create a Map to store user data
    let userData = new Map();

    // Add user data to the Map
    userData.set(1001, { name: 'Alice', email: 'alice@example.com', age: 30 });
    userData.set(1002, { name: 'Bob', email: 'bob@example.com', age: 25 });
    userData.set(1003, { name: 'Charlie', email: 'charlie@example.com', age: 35 });

    // Get user data by user ID
    user = userData.get(1002);
    console.log(user);
    // Output: { name: 'Bob', email: 'bob@example.com', age: 25 }

    // Update user data
    userData.set(1002, { name: 'Bob Smith', email: 'bob@example.com', age: 26 });

    // Check if a user exists
    console.log(userData.has(1003)); // Output: true

    // Delete a user
    userData.delete(1003);
    console.log(userData.size); // Output: 2

// - Map vs WeakMap : -------------------------[]

/*

    "
        WeakMap Allows Garbage Collector To Do Its Task But Not Map.
    "
    --
    Map     => Key Can Be Anything
    WeakMap => Key Can Be Object Only
    --
*/
// example : --[
    let weak = new WeakMap();

     obj = {
        firstName: "ayoub",
    };

    weak.set(obj, "value");
    console.log(weak);

    obj = null;
    console.log(weak);
```

```

// notes : --[
/*
    The behavior you're observing is expected when using WeakMap in JavaScript.

    WeakMap holds weak references to its keys, which means that if there are no other references
    to a key object apart from the  one stored in the WeakMap, the key object may still be garbage
    collected. However, the garbage collection process is non-deterministic,
    meaning that it's not guaranteed to happen immediately after an object becomes unreachable.

    In your code, even though you set obj to null, the garbage collector may not immediately reclaim the
    memory occupied by obj and its associated key-value pair in the WeakMap. Garbage collection is
    typically performed by the JavaScript engine asynchronously, and the exact timing may vary
    depending on various factors such as memory pressure, garbage collection algorithms, and
    optimizations performed by the engine.

    To test if the object is actually removed from the WeakMap, you would need to explicitly trigger
    garbage collection, but this isn't possible directly in JavaScript. However, you could try forcing
    garbage collection in some environments by using platform-specific tools or by running the script
    in a controlled environment that allows such actions.

    In summary, while WeakMap helps in managing weak references and allows objects to be garbage collected
    when they are no longer reachable from other parts of the program, the timing of garbage collection is
    not deterministic and may vary depending on the JavaScript engine and runtime environment.
*/

// convert from element  to array :  -------------------------[]
    Array.from(element,(item)={
        //code
    })

// convert from string to array
    Array.from("ayoub");

// get the sum of a string :
    Array.from("12345",(item)=> +item + +item);

// convert from set to Array :
      myArr=[1,1,1,2,2,4];
      set= new Set(myArr);

    console.log(Array.from(set));
// you can also use the spread operator ...
    myArr=[1,1,1,2,2,4];
    set= new Set(myArr);
    console.log([...set]);

```

```
// copy a  values from an array to another position
myArr.copyWithin(targetIndex,startCopyIndex=0,EndCopyIndex=arr.length) // end not included

// example :
    let myArray = [10, 20, 30, 40, 50, "A", "B"];

    myArray.copyWithin(3); // [10, 20, 30, 10, 20, 30, 40]

    myArray.copyWithin(4, 6); // [10, 20, 30, 40, "B", "A", "B"]

    myArray.copyWithin(4, -1); // [10, 20, 30, 40, "B", "A", "B"]

    myArray.copyWithin(1, -2); // [10, "A", "B", 40, 50, "A", "B"]

    myArray.copyWithin(1, -2, -1); // [10, "A", 30, 40, 50, "A", "B"]

// check if any value at least match the condition in an array : [true /false]
arr.some(CallbackFunc(Element, Index, Array),ValueToCompareWith)
/*
    - Array.some(CallbackFunc(Element, Index, Array))
    --- CallbackFunc => Function To Run On Every Element On The Given Array
    ------ Element => The Current Element To Process
    ------ Index => Index Of Current Element
    ------ Array => The Current Array Working With
    ------ ValueToCompareWith = it's represented by this inside the CallbackFunc()
    --
    Using
    - Check if Element Exists In Array
    - Check If Number In Range
*/
// example : --[
    var arr = [1, 2, 3, 4, 5, 6];

    // method 1 :
        var status = arr.some(function (element) {
            return element > this;
        }, 5);

    // method 2 :
    var status = arr.some(element=>element >5);

    // check  if an element  is exist in an array  :
        var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        function checkValues(arr, val) {
            return arr.some(function (e) {
            return e === val;
            });
        }

        console.log(checkValues(nums, 20));
        console.log(checkValues(nums, 5));

    // check if at least one element  in the array is in a range :
        var range = {
            min: 10,
            max: 20,
```

```
        };

        var checkNumberInRange = nums.some(function (e) {
            return e >= this.min && e <= this.max;
        }, range);

        console.log(checkNumberInRange);



// check if all values  match the condition in an array : [true /false]
arr.every(CallbackFunc(Element, Index, Array),ValueToCompareWith)
/*
    - Array.some(CallbackFunc(Element, Index, Array))
    --- CallbackFunc => Function To Run On Every Element On The Given Array
    ------ Element => The Current Element To Process
    ------ Index => Index Of Current Element
    ------ Array => The Current Array Working With
    ------ ValueToCompareWith = it's represented by this inside the CallbackFunc()
    --
    Using
    - Check If all  values match a condition
*/
// example : --[
    var arr = [1, 2, 3, 4, 5, 6];

    // method 1 :
        var status = arr.every(function (element) {
            return element > this;
        }, 5);

    // method 2 :
    var status = arr.every(element=>element >5);


    // check if all values   in the array is in a range :
    var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        var range = {
            min: 10,
            max: 20,
        };

        var checkNumberInRange = nums.every(function (e) {
            return e >= this.min && e <= this.max;
        }, range);

        console.log(checkNumberInRange);

// find  an element  using value  : -------------------------[]
arr.find(CallbackFunc(Element, Index, Array),ValueToCompareWith)
/*
    - Array.find(CallbackFunc(Element, Index, Array))
    --- CallbackFunc => Function To Run On Every Element On The Given Array
    ------ Element => The Current Element To Process
    ------ Index => Index Of Current Element
    ------ Array => The Current Array Working With
```

```
    ------ ValueToCompareWith = it's represented by this inside the CallbackFunc()
    --
    Using
    - get value of the first  ArrayElement that match  the value passed in parameter

    return undefined if not / element if exist
*/
// example : --[
    var arr = [1, 2, 3, 4, 5, 6];

    // method 1 :
        var status = arr.find(function (element) {
            return element > this;
        }, 5);

    // method 2 :
    var status = arr.find(element=>element >5);

      //  get the value  of the first element equal value    :
        var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        function getValue(arr, val) {
            return arr.find(function (e) {
            return e === val;
            });
        }

        console.log(getValue(nums, 20));
        console.log(getValue(nums, 5));

    //  get the value of the first element in the range   :
        var range = {
            min: 10,
            max: 20,
        };

        var valueInRange = nums.find(function (e) {
            return e >= this.min && e <= this.max;
        }, range);

        console.log(valueInRange);

// find  an element  using index  : -------------------------[]
arr.findIndex(CallbackFunc(Element, Index, Array),ValueToCompareWith)
/*
    - Array.find(CallbackFunc(Element, Index, Array))
    --- CallbackFunc => Function To Run On Every Element On The Given Array
    ------ Element => The Current Element To Process
    ------ Index => Index Of Current Element
    ------ Array => The Current Array Working With
    ------ ValueToCompareWith = it's represented by this inside the CallbackFunc()
    --
    Using
    - Check if Element Exists In Array

    return -1 if not / index of element  if exist
*/
```

```
// example : --[
    var arr = [1, 2, 3, 4, 5, 6];

    // method 1 :
    var status = arr.findIndex(function (element) {
            return element > this;
        }, 5);

    // method 2 :
    var status = arr.find(element=>element >5);

    //  get the index of  an element  in an array  :
        var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        function GetIndex(arr, val) {
            return arr.findIndex(function (e) {
            return e === val;
            });
        }

        console.log(GetIndex(nums, 20));
        console.log(GetIndex(nums, 5));

    //  get the index of the first element in the range   :
        var range = {
            min: 10,
            max: 20,
        };

        var elementIndexInRange = nums.findIndex(function (e) {
            return e >= this.min && e <= this.max;
        }, range);

        console.log(elementIndexInRange);

//  Spread Operator : -------------------------[]
/*
    Spread Operator => ...Iterable
    "Allow Iterable To Expand In Place"
*/

// Spread With String => Expand String
    console.log("Osama");
    console.log(..."Osama"); // split to characters : O s a m a
    console.log([..."Osama"]); // split to array of characters : ['O', 's', 'a', 'm', 'a']
    console.log({..."Osama"}); // split to object:  {0: 'O', 1: 's', 2: 'a', 3: 'm', 4: 'a'}

// Concatenate Arrays
    let myArray1 = [1, 2, 3];
    let myArray2 = [4, 5, 6];
    let allArrays = [...myArray1, ...myArray2]; //: [1, 2, 3, 4, 5, 6]
    console.log(allArrays);

// Copy Array
    let copiedArray = [...myArray1];
        console.log(copiedArray); //: [1, 2, 3]
```

```
// Push Inside Array
    let allFriends = ["Osama", "Ahmed", "Sayed"];
    let thisYearFriends = ["Sameh", "Mahmoud"];
    allFriends.push(...thisYearFriends);
    console.log(allFriends); //: ['Osama', 'Ahmed', 'Sayed', 'Sameh', 'Mahmoud']

// Use With Math Object
    let myNums = [10, 20, -100, 100, 1000, 500];
    console.log(Math.max(...myNums)); //: 1000

// Spread With Objects => Merge Objects
    let objOne = {
    a: 1,
    b: 2,
    };
    let objTwo = {
    c: 3,
    d: 4,
    };
    console.log({ ...objOne, ...objTwo, e: 5 });//: {a: 3, b: 2, d: 4, e: 5}

// assignment :  --------------------------[]
    let n1 = [10, 30, 10, 20];
    let n2 = [30, 20, 10];
    // Result : 200 * 1.05 = 210
    console.log(
            ( n1[new Set([...n1]).size] * Math.min(...n2) )  // Result : 200
            //Result : 0.05 * 21 = 1.05
            * ( // Result = 5 /100 = 0.05
                (
                    // Result : 20 / 4 = 5
                    (  n1[new Set([...n1]).size]  // n1[3]=20
                    // Result : 2*2=4
                    / (
                            (  Math.max( ... n1.join("").split("").slice((new Set([...n2])
.size)) )  )  //2
                        *  (  Math.max( ... n1.join("").split("").slice((new Set([...n2])
.size)) )  ) // 2
                    )
                    )
                / (Math.min(...n1) * Math.min(...n1)) // Result : 10*10 = 100
                )
                // Result : 21
            * (
                            ( new Set([...n1]).size ) // Result : 3
                        * ( [...n1,...n2].length ) //  Result : 7
                )

            )
            );

    // explain
    /*
        1- ( n1[new Set([...n1]).size] * Math.min(...n2) ) = 20 * 10 = 200 ://*R1

        2-( n1[new Set([...n1]).size] ) = n1[3]=20 ://*R2

```

```
        3-( Math.max( ... n1.join("").split("").slice((new Set([...n2]).size)) ) ) = 2 ://*R3
        4-( R2 / (   ( R2 ) * ( R2 )  )  = 20/(2*2)=5 ://*R4
        5-( Math.min(...n1) ) = 10 ://*R5
        6-( R4/(R5 * R5 ) ) = 5/100 = 0.05  ://*R6
        7-( new Set([...n1,]).size ) = 3 ://*R7
        8-( new Set([...n1, ...n2]).size )  = 7  ://*R8
        9-( R6 * ( R7 *  R8  ) ) = 0.05 * (7*3) = 1.05 ://*R9
        //*result :
        console.log(R1*R9) =210
    */

    // or :
        // Result : 200 * 1.05 = 210
        console.log(Math.max(...n2)  *  [...n1, ...n2].length);


// regular Expressions : ------------------------[]
/*
    Regular expressions, often abbreviated as regex or regexp, are sequences of characters that define
    a search pattern. They are widely used in computer science and programming for tasks such as
    pattern matching, string manipulation, and data validation. Regex allows for the efficient
    and flexible processing of text by enabling the identification and extraction of specific
    patterns within strings.
*/

// 1. Literal Characters:
/*
   - Literal characters match themselves within a string. For example,
    the regex pattern 'hello' matches the word 'hello' in a string.
*/
// 2. Metacharacters:
/*
   - Metacharacters have special meanings in regular expressions and are used to define patterns
*/
    /*
        - '.' (dot): Matches any single character except newline.
        - '^' (caret): Matches the start of a line.
        - '$' (dollar): Matches the end of a line.
        - '*' (asterisk): Matches zero or more occurrences of the preceding character.
        - '+' (plus): Matches one or more occurrences of the preceding character.
        - '?' (question mark): Matches zero or one occurrence of the preceding character.
        - '|' (pipe): Acts as an OR operator, allowing alternative matches.
        - '[' and ']' (square brackets): Define a character class, matching any character within the brackets.
        - '\' (backslash): Escapes metacharacters to match them literally.
    */
// 3. Character Classes:
/*
   - Character classes represent groups of characters and allow for more concise pattern definitions.
*/
    /*
```

```
3000
3001         - \d: Matches any digit (equivalent to [0-9]).
3002         - \w: Matches any word character (alphanumeric characters plus underscore).
3003         - \s: Matches any whitespace character.
3004         - \D, \W, \S: Negations of \d, \w, \s respectively.
3005         - [abc]: Matches any single character within the brackets.
3006         - [^abc]: Matches any single character not within the brackets.
3007     */
3008 // 4. Quantifiers:
3009 /*
3010     - Quantifiers specify the number of occurrences of a character or group in a pattern.
3011 */
3012  /*
3013     - {n}: Matches exactly n occurrences.
3014     - {n,}: Matches at least n occurrences.
3015     - {n,m}: Matches between n and m occurrences.
3016     - *: Matches zero or more occurrences.
3017     - +: Matches one or more occurrences.
3018     - ?: Matches zero or one occurrence.
3019 */
3020 // 5. Anchors:
3021 /*
3022    - Anchors assert positions in the string without consuming characters.
3023 */
3024     /*
3025         - \b: Matches a word boundary.
3026         - \B: Matches a non-word boundary.
3027         - ^: Matches the start of a string.
3028         - $: Matches the end of a string.
3029     */
3030
3031 // 6. Groups and Capturing:
3032 /*
3033     - Parentheses () are used to create groups within a regular expression. Groups can
3034      be captured for extraction or used for quantification and alternation.
3035 */
3036 // Applications of Regular Expressions: --[
3037     // 1. Text Search and Validation:
3038         /*
3039         - Validating email addresses, phone numbers, URLs, and other structured data
    formats.
3040         - Searching for specific patterns or keywords within text documents or strings.
3041         - Extracting information from unstructured text data.
3042         */
3043     // 2. Data Processing and Transformation:
3044         /*
3045         - Parsing and extracting data from logs, files, or structured documents.
3046         - Transforming data by replacing, removing, or reformatting specific patterns.
3047         */
3048     // 3. Input Validation and Sanitization:
3049         /*
3050         - Ensuring that user input adheres to specified formats or constraints before
    processing.
3051         - Sanitizing input to prevent injection attacks or malicious input.
3052         */
3053     // 4. Text Manipulation and Formatting:
3054         /*
```

```
        - Replacing or modifying text based on predefined patterns or rules.
        - Formatting text for display or storage according to specific conventions.
        */
// test your pattern [true: any matches word /false]
    var reg=/exp/option;
    console.log(reg.test())

//test your pattern at a string [matches words]
    var reg=/exp/option;
    var str1='text'
    /*
    1-matches a string against a regular expression pattern
    2-returns an array with the matches
    3-returns null if no math is found
     */
console.log(str1.match(str));

// replace the matches words :  -- []
    // replace the first match word  :
        console.log(str1.replace(reg,replacedValue));

    // replace all  matches word  :
        console.log(str1.replaceAll(reg,replacedValue));

//examples :

// example 1 : --[
/*
    v1|v2 : or
    [0-9] : range of value from 0 to 9
    [^0-8] : values not in this range
    [^0-9] : get values not no a number
*/
    var myString1='AaBbcdefG123!234%^&*'

    let SmallAToZ=/[a-z]/g;
    let NotSmallAToZ=/[^a-z]/g;
    let capitalAToZ=/[A-Z]/g;
    let NotCapitalAToZ=/[^A-Z]/g;
    let acd=/[acd]/g;
    let notAcd=/[^acd]/g;

    let letters=/[aA-zZ]/g;
    let notLetters=/[^aA-zZ]/g;
    let special =/[^aA0-zZ9]/g;
    console.log(myString1.match(SmallAToZ));
    console.log(myString1.match(NotSmallAToZ));
    console.log(myString1.match(capitalAToZ));
    console.log(myString1.match(NotCapitalAToZ));
    console.log(myString1.match(acd));
    console.log(myString1.match(notAcd));
    console.log(myString1.match(letters));
    console.log(myString1.match(notLetters));
    console.log(myString1.match(special));
// example 2 :--[
/*
```

```
--character classes :
    .  : matches any character , except newline or other line terminators
    \w : matches word characters. [a-z,A-Z,0-9 and underscore]
    \W : matches Non word characters
    \d : matches digits from 0-9
    \s : matches whitespace character
*/
    let email ='O@@@g...com o@g.com o@g.net o@g.com o-g.com o@s.org 1@1.com'
    let dot=/./g;
    let word=/\w/g;
    let noWord=/\W/g;
    let valid=/\w@\w.(com|net)/g;
    console.log(email.match(dot));
    console.log(email.match(word));
    console.log(email.match(noWord));
    console.log(email.match(valid));

// example 3 : --[
/*
    \b : matches at the beginning or end of word.
    \B : matches not at the  beginning/end of a word.
*/
    var names="Sayed 1Spam 2Spam 2Spam spam4 spam5 Osama Ahmed Aspamo";
    let re=/(\bspam|spam\b)/ig;

    console.log(names.match(re));
    console.log(re.test("spam"))

// example 4 : --[
// Quantifiers :
/*
    n+ ... => one or more
    n* ... => zero or more
    n? ..  =>  zero or n
    n{x}   : Number of n
    n{x,y} : range from x to y
    n{x,}  : at least x
    $ : en with something
    ^  : start with something
    ?= : followed by something
    ?! : not followed by something
*/
    let mails='o@nn.sa osama@gmail.com elzero@gmail.net osama@mail.ru';
    var numbers1='0110 10 150  05120 0560 350 00'
    let reg=/\w+.\w+/ig;
    console.log(mails.match(reg));
    let resNumbers=/0\d+0/ig;
    console.log(numbers1.match(resNumbers));


    let urls='https://google.com http://www.website.net web.com https';
    let regUrl=/(https?:\/\/)?(www.)?\w+.\w+/ig;
    console.log(urls.match(regUrl));

    let serials='s100s s3000s s50000s s950000s';
    let reg1=/s\d{4,}s/g;
```

```
    console.log(serials.match(reg1));

    let str2='we love Programming';
    let names='10osamaZ 2AhmedZ 3Mohammed 4MoustafaZ 5GamalZ';
    console.log(/^[a-z]{2}\s/ig.test(str2));
    let  reg2=/\d\w{5}z/ig;
    reg2=/\d\w{5}(?!z)/ig;
    console.log(names.match(reg2))
// OOP [OBJECT ORIENTED PROGRAMMING ] : ------------------------[]
```
    Object-Oriented Programming (OOP) Overview:

        Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects,"
        which can contain data in the form of fields (attributes or properties) and code in the form of
        procedures (methods or functions). OOP emphasizes organizing code into reusable and modular components,
        fostering better code maintenance, scalability, and flexibility. Key concepts of OOP include:

    1. Classes and Objects:
    - Classes are blueprints or templates for creating objects. They define the attributes (data)
        and behaviors (methods) that objects of the class will have.
    - Objects are instances of classes, representing specific entities in the program's domain.
        They encapsulate data and behavior related to that entity.

    2. Encapsulation:
    - Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on
        the data into a single unit (class or object).
    - It promotes information hiding, where the internal details of an object are hidden from the
        outside world, and only a well-defined interface is exposed for interacting with the object.

    3. Inheritance:
    - Inheritance allows a class (subclass or derived class) to inherit properties and
        behaviors from another class (superclass or base class).
    - Subclasses can extend the functionality of their superclass, adding new features or
        overriding existing ones, while maintaining the common interface.

    4. Polymorphism:
    - Polymorphism enables objects to be treated as instances of their superclass,
        allowing for more generic code that can operate on objects of different types.
    - It includes concepts like method overriding (redefining a method in a subclass)
        and method overloading (defining multiple methods with the same name but different parameters).

    5. Abstraction:
    - Abstraction focuses on modeling the essential aspects of real-world
        entities while hiding unnecessary complexity.
    - It allows developers to create simplified models of complex systems,
        emphasizing what an object does rather than how it does it.

    Benefits of Object-Oriented Programming:
```
```

```
    1. Modularity: OOP promotes code modularity by encapsulating related functionality into classes
                and objects, making code easier to understand, maintain, and reuse.

    2. Reusability: Objects and classes can be reused in different parts of the program
                    or in different programs, reducing redundancy and promoting code efficiency.

    3. Scalability: OOP provides a scalable approach to software development, allowing for
                    easy extension and modification of existing code without affecting other
                    parts of the system.

    4. Flexibility: OOP enables developers to build flexible and adaptable systems that can evolve
                    over time to meet changing requirements and business needs.

    5. Understandability: OOP emphasizes a clear and intuitive design approach, making it easier for
                        developers to understand and reason about the structure and behavior of the code.
```

```
// general syntax : --[
    class className {
        constructor(args) {
            this.att1 = value1;
            // ...
        }
        f1 = function () {};
        f2 = () => {
            // this is window in this scope
        };

        f3() {}
    }
// create new instance from a class : --[
    let instance1= new className(args);

// check if an object in an instance of a specific class :--[
    console.log(obj instanceof className);
// example : --[

// old method
    function User(id, username, salary) {
        this.id = id;
        this.username = username;
        this.salary = salary + 1000;
    }

    var user1 = new User(1, "amine", 500);
    var user2 = new User(2, "kamal", 2500);
    var user3 = new User(3, "nice", 3500);
    console.log(user1, "\n", user2, "\n", user3);
// ES6 :
    class User {
        constructor(id, username, salary) {
```

```
            this.id = id;
            this.username = username;
            this.salary = salary + 1000;
        }
    }
    let user1 = new User(1, "amine", 500);
    let user2 = new User(2, "kamal", 2500);
    let user3 = new User(3, "nice", 3500);
    console.log(user1, "\n", user2, "\n", user3);

// static members :
/*
    Static members in JavaScript classes are properties or methods that belong
    to the class itself rather than to instances of the class. They are shared
    among all instances of the class and can be accessed directly from the class
    without the need to create an instance. Static members are declared using
    the static keyword within a class definition.
*/
//static members :
    class Circle {
    //Static properties are shared across all instances of the class and are accessed using
the class name itself.
        static PI = 3.14;
        static radius = 1;

        static getArea() {
        return Circle.PI * Circle.radius * Circle.radius;
        }
    }
  console.log(Circle.PI); // Output: 3.14
  console.log(Circle.getArea()); // Output: 3.14
//static methods :
    class MathUtils {
    //Static methods are functions that belong to the class itself and do not operate on
instances of the class.
        static add(x, y) {
        return x + y;
        }

        static subtract(x, y) {
        return x - y;
        }
    }
    console.log(MathUtils.add(5, 3)); // Output: 8
    console.log(MathUtils.subtract(5, 3)); // Output: 2
// Static Factory Methods:
    class Point {
    //Static factory methods are static methods used to create instances of the class.
        constructor(x, y) {
          this.x = x;
          this.y = y;
        }

        static createOrigin() {
          return new Point(0, 0);
        }
      }
```

```
      const origin = Point.createOrigin();
      console.log(origin); // Output: Point { x: 0, y: 0 }

// private members :
    class className{
        #PrivateProperty
        constructor(args){
            this.#PrivateProperty=value;
        }
        #privateMethod(){
        // code

        }
    }
//example :
    class Counter {
        #count = 0;

        #increment() {
        this.#count++;
        }

        #getCount() {
        return this.#count;
        }

        publicMethod() {
        this.#increment();
        return this.#getCount();
        }
    }

    counter = new Counter();
    console.log(counter.publicMethod()); // Output: 1
    console.log(counter.#count); // Error: Private field '#count' is not defined
    console.log(counter.#increment()); // Error: Private method '#increment' is not defined
// inheritance :
class  baseClass{
 constructor(arg1){
 this.p1=arg1;
 /// ...
 }
 getPass(){
    this.p1 *10;
 }

};
class derivedClass extends baseClass{
constructor(arg1,arg2){
    // access constructor of base class :
    super(arg1);
    this.p2=arg2;

}

getDerivedClass(){
```

```
 // access function of baseClass :
return  super.getPass() + this.arg2;
}
}
// example :
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }
    class Student extends Person {
        constructor(name, age, grade) {
        super(name, age);
        this.grade = grade;
        }

        study() {
        console.log(`${this.name} is studying.`);
        }
    }

    const student2 = new Student('Emma', 18, 'A');
    student2.greet(); // Output: Hello, my name is Emma and I am 18 years old.
    student2.study(); // Output: Emma is studying.

// Prototypes :
/*
    In JavaScript, each object has an associated prototype, which serves as a fallback
    mechanism for properties and methods that are not found directly on the object.
    Prototypes allow for property and method inheritance, where objects can inherit
    properties and methods from their prototype chain.
*/
//example
    class cPerson {
        constructor(name, age) {
            this.name = name;
            this.age = age;
        }

        syaHello() {
            return `Hello ${this.name}`;
        }
    }

    cPerson.prototype.nice = function () {
        return `nice to meet you ${this.name}`;
    };
    console.log(cPerson.prototype);
    const person3 = new cPerson("Alice", 25);
    console.log(person3.nice());

// object meta data descriptor : -------------------------[]
/*
    -writable  : change his value
    -enumerable : not appears in loops (log the object , iterate object key [for in ])
    -configurable[cannot delete or reconfigure ]
    -value : default value of  the property
```

```
*/
// syntax  :
    Object.defineProperty(object,'propertyName',{
    writable : value,
    enumerable : value,
    configurable : value,
    value: value,
    })

// example :
    const myObject = {
        a: 1,
        b: 2,
        c: 3,
    };

    Object.defineProperty(myObject, "c", {
        writable: false,
        enumerable: false,
        configurable: false,
        value: 500,
    });

    myObject.c = 100;

    console.log(delete myObject.c);
    console.log(myObject.c);

// change multiple property metadata:
Object.defineProperties(myObject,  {
    'a':{
    writable: value,
    enumerable: value,
    configurable: value,
    value:value,
    },
    'b':{
        writable: value,
        enumerable: value,
        configurable: value,
        value:value,
    },
    'c':{
        writable: value,
        enumerable: value,
        configurable: value,
        value:value,
    },
});

// get property metadata :
    Object.getOwnPropertyDescriptor(myObject,'propertyName');

// get all properties metadata :
    Object.getOwnPropertyDescriptors(myObject);

```

```
//  Date And Time : ------------------------[]
// get current date info :
    console.log(new Date());

// get number of milliseconds  From 1/1/1970 until now :
    let milliseconds = Date.now();

//  converting unites :
// number of seconds :
    let seconds = milliseconds / 1000;
    console.log(`seconds ${seconds}`);

    // minutes :
    let minutes = seconds / 60;
    console.log(`minutes ${minutes}`);

    // hours
    let hours = minutes / 60;
    console.log(`hours ${hours}`);

    // days :
    let days = hours / 24;
    console.log(`days ${days}`);

    // months :
    let months = days / 30;
    console.log(`months ${months}`);

    // years :
    let years = months / 12;
    console.log(`years ${years}`);

// get difference between two date  :
    var dateNow = new Date();
    console.log(dateNow);
    let birthDay = new Date("12/10/2003");

    console.log((dateNow - birthDay) / 1000 / 60 / 60 / 24 / 365);

// Date functions :--[
// get functions :
    let date= new Date("12/10/2003");
    console.log(date.getTime()); //get number of milliseconds  From 1/1/1970 until Date:
    console.log(date.getDate()); // get day number  in month
    console.log(date.getFullYear()); // get year
    console.log(date.getMonth()); // get month [0,11]
    console.log(date.getDay()); // get  day  index order in the week [From sunday To
saturday]
    console.log(date.getHours()); // get  current hours (24 format)
    console.log(date.getMinutes()); // get  current minutes
    console.log(date.getSeconds()); // get  current seconds

// set functions :
/*
    - setTime(Milliseconds)
    - setDate() => Day Of The Month [Negative And Positive]
```

```
    - setFullYear(year, month => Optional [0-11], day => Optional [1-31])
    - setMonth(Month [0-11], Day => Optional [1-31]) [Negative And Positive]
    - setHours(Hours [0-23], Minutes => Optional [0-59], Seconds => Optional [0-59], MS => Optional [0-999])
    - setMinutes(Minutes [0-59], Seconds => Optional [0-59], MS => Optional [0-999])
    - setSeconds(Seconds => [0-59], MS => Optional [0-999])
*/
// example :
    let dateNow = new Date();
    dateNow.setTime(0);
    console.log(dateNow); // Thu Jan 01 1970 00:00:00 GMT+0000 (GMT)

    dateNow.setTime(10000);
    console.log(dateNow);//Thu Jan 01 1970 00:00:10 GMT+0000 (GMT)

    dateNow.setDate(35);
    console.log(dateNow); // Wed Feb 04 1970 00:00:10 GMT+0000 (GMT)

    dateNow.setFullYear(2020, 13);
    console.log(dateNow); //Wed Feb 04 1970 00:00:10 GMT+0000 (GMT)

    dateNow.setMonth(15);
    console.log(dateNow); //Mon Apr 04 2022 00:00:10 GMT+0000 (GMT)

// Date Formatting : --[
/*
    new Date(timestamp)
    new Date(Date String)
    new Date(Numeric Values)

    Format
    - "Oct 25 1982"
    - "10/25/1982"
    - "1982-10-25" => ISO International Standard
    - "1982 10"
    - "1982"
    - "82"
    - 1982, 9, 25, 2, 10, 0 => year monthIndex day hours minutes seconds
    - 1982, 9, 25
    - "1982-10-25T06:10:00Z"

    Date.parse("String") // Read Date From A String
*/

    console.log(Date.parse("Oct 25 1982"));

    let date1 = new Date(0);
    console.log(date1);

    let date2 = new Date(404344800000);
    console.log(date2);

    let date3 = new Date("10-25-1982");
    console.log(date3);

    let date4 = new Date("1982-10-25");
```

```
    console.log(date4);

    let date5 = new Date("1982-10");
    console.log(date5);

    let date6 = new Date("82");
    console.log(date6);

    let date7 = new Date(1982, 9, 25, 2, 10, 0);
    console.log(date7);

    let date8 = new Date(1982, 9, 25);
    console.log(date8);

    let date9 = new Date("1982-10-25T06:10:00Z");
    console.log(date9);

//Track Operations Time : ---[
    // Start Time
    let start = new Date();

    // Operation
    for (let i = 0; i < 100; i++) {
    document.write(`<div>${i}</div>`);
    }
    // Time End
    let end = new Date();

    // Operation Duration
    let duration = end - start;

    console.log(`${duration}ms`);

// Generators :  -------------------------[]
 /*
    - Generator Function Run Its Code When Required.
    - Generator Function Return Special Object [Generator Object]
    - Generators Are Iterable
*/
 /*

    Generators in JavaScript are special functions that can pause and resume their
execution.
    They are defined using the function* syntax and utilize the yield keyword to pause the
function's
    execution and return a value to the caller. Generators offer a powerful mechanism for
creating
    iterable sequences and asynchronous programming. Here's an overview of their usage:
*/

//1-Declaring a Generator Function: -- [
/*
    Generator functions are declared using the function* syntax.
    Within the generator function, you can use the yield keyword
    to pause the execution and return a value to the caller.
*/
//Example :
```

```
    function* generatorFunction() {
        yield 1;
        yield 2;
        yield 3;
    }
// 2- Iterating Over Generator Values: -- [
/*
    Generators produce iterable sequences that can be iterated over using loops,
     spread syntax,  or built-in iterator methods like next().
*/
//Example :
    const gen = generatorFunction();
    console.log(gen.next().value); // Output: 1
    console.log(gen.next().value); // Output: 2
    console.log(gen.next().value); // Output: 3



//3-Using for...of Loop with Generators: --[
/*
    Generators can be used with the for...of loop to iterate over all
    values produced by the generator.
*/
//Example :
    for (const value of generatorFunction()) {
        console.log(value);
    }
    // Output:
    // 1
    // 2
    // 3

//4-Passing Values to Generators: --[
/*
    In JavaScript, the next() method of an iterator returned by a generator function allows you to either
    retrieve the next value from the generator or to pass a value into the generator function. It returns
    an object with two properties: value and done.

    When you call next() without passing a parameter, the generator function continues its execution until it
    encounters a yield statement, where it pauses and returns the yielded value as value.
    If the generator function completes execution without encountering a yield statement, the value
    property will be undefined, and the done property will be true.
    If you pass a parameter to next(value), the value passed as a parameter will be returned by
    the yield expression inside the generator function, and the generator will continue its execution
    from where it was paused.
    If an error occurs within the generator function, you can throw an exception using the throw()
    method of the iterator. This will cause the generator function to throw an error at the point
    where it's currently paused.
    */
//Example :
```

```
function* myGenerator() {
    try {
      const x = yield 1; // Pause execution and return 1 when next() is called
      console.log('Received:', x);
      const y = yield 2; // Pause execution and return 2 when next() is called
      console.log('Received:', y);
      yield 3; // Pause execution and return 3 when next() is called
    } catch (error) {
      console.log('Error caught:', error);
    }
  }

  const iterator = myGenerator();
  console.log(iterator.next()); // Output: { value: 1, done: false }
  console.log(iterator.next('Hello')); // Output: Received: Hello, { value: 2, done: false }
  console.log(iterator.throw(new Error('Something went wrong'))); // Output: Error caught: Error: Something went wrong


//5- Return() : --[
/*
    When return is called, the generator function exits early, and subsequent calls to next()
    will return an object with done set to true.
    The provided value (if any) is returned as the value property of the object returned by the next() method.
    If return is called with no argument, it's equivalent to return undefined.
*/
//Example :
    function* myGenerator() {
        yield 1;
        yield 2;
        yield 3;
    }

    iterator = myGenerator();
    console.log(iterator.next()); // Output: { value: 1, done: false }
    console.log(iterator.return('Stopped')); // Output: { value: 'Stopped', done: true }
    console.log(iterator.next()); // Output: { value: undefined, done: true }


//6- Asynchronous Generators: --[
/*
    Generators can be used in asynchronous programming to produce asynchronous iterable
    sequences using asynchronous operations.
*/
//Example :
    async function* asyncGenerator() {
        yield await Promise.resolve(1);
        yield await Promise.resolve(2);
        yield await Promise.resolve(3);
    }

    (async () => {
        for await (const value of asyncGenerator()) {
        console.log(value);
        }
```

```
    })();
    // Output:
    // 1
    // 2
    // 3
//7-  Delegate Generator Function: --[
    //1. `generateNums()`:
      /*
        - This generator function yields three numeric values: 1, 2, and 3.
        - It uses the `yield` keyword to pause execution and return each value one by one.
    */
    //2. `generateLetters()`:
    /*
        - This generator function yields three letter values: "A", "B", and "C".
        - Similar to `generateNums()`, it uses the `yield` keyword to pause execution and return each value one by one.
    */
    //3. `generateAll()`:
    /*
         - This generator function combines the values from `generateNums()`, `generateLetters()`,
         and an array `[4, 5, 6]`.
         - It uses the `yield*` syntax to delegate to other generator functions and an iterable (the array).
         - When `yield*` is used, it iterates over the values yielded by the delegated generator or iterable
            and yields each value in the current generator function.
         - So, `yield* generateNums();` yields the values from `generateNums()`, `yield* generateLetters();
         ` yields the values from `generateLetters()`, and `yield* [4, 5, 6];`
         yields the values from the array `[4, 5, 6]`.
    */

     iterator = generateAll();

    console.log(iterator.next()); // Output: { value: 1, done: false }
    console.log(iterator.next()); // Output: { value: 2, done: false }
    console.log(iterator.next()); // Output: { value: 3, done: false }
    console.log(iterator.next()); // Output: { value: 'A', done: false }
    console.log(iterator.next()); // Output: { value: 'B', done: false }
    console.log(iterator.next()); // Output: { value: 'C', done: false }
    console.log(iterator.next()); // Output: { value: 4, done: false }
    console.log(iterator.next()); // Output: { value: 5, done: false }
    console.log(iterator.next()); // Output: { value: 6, done: false }
    console.log(iterator.next()); // Output: { value: undefined, done: true }

// Advance Real Example :  -- [
    /*
        Let's consider a practical example of using generators for asynchronous programming.
        Imagine we have an API that returns data in chunks, and we want to fetch and process this
        data asynchronously. We'll use a generator function to handle the asynchronous fetching of
        data and processing it chunk by chunk.
    */
    // Simulated asynchronous API function to fetch data in chunks
    function fetchDataChunk() {
        return new Promise(resolve => {
```

```
        // Simulating delay for fetching data
        setTimeout(() => {
            // Simulating data chunk
            const chunk = [1, 2, 3, 4, 5];
            resolve(chunk);
        }, 1000);
        });
    }

    // Generator function to asynchronously fetch and process data in chunks
    async function* processData() {
        let pageIndex = 1;
        while (pageIndex<=10) {
        // Fetch data chunk asynchronously
        const dataChunk = await fetchDataChunk();
        yield { pageIndex, dataChunk }; // Yield data chunk along with page index
        pageIndex++;
        }
    }

    // Function to consume data from the generator asynchronously
    async function consumeData() {
        for await (const { pageIndex, dataChunk } of processData()) {
        console.log(`Page ${pageIndex}:`, dataChunk);


        // Process data chunk here (e.g., save to database, perform calculations, etc.)
        }
    }

    // Start consuming data
    consumeData();

//  Modules Import And Export : -------------------------[]
// you need  add a type='module' to script:src (html file )
```
    <script src="main.js" type="module"></script>
    <script src="app.js" type="module"></script>
```
/*
    type="module" attribute, indicating that the JavaScript files (app.js and main.js) are ES modules.
    When you use this attribute, the browser treats the scripts as ECMAScript modules.
*/
// export a variable
    export let varName=value;

// export a function :
    export function functionName(){

    };

// export a class :
    export class className{

    };
```

```
// export a collection :
    export {
    varName,
    functionName,
    className
    };

// import var from a file
// alias optional :
import {varName  as alias , functionName  as alias   , className as alias } from '
path/file.js';

// Example :  --[
    // index.html :
    ```
        <script src="main.js" type="module"></script>
        <script src="app.js" type="module"></script>
    ```

    // main.js :
        let ak = 10;
        let arr = [1, 2, 3, 4, 5]; // Fix the array declaration

        function saySomething() {
            return "nice to meet you";
        }

        export { ak, arr, saySomething };
    // app.js  :
        import {ak,arr as array,saySomething} from './main.js'
        console.log(ak);
        console.log(array)
        console.log(saySomething());

    // import all export function and  grouped in a single object  :
    import * as containerName from 'path/source.js';

// Default export :
/*
you can export just one time
*/

// exportDefaultVariable :
```
    export default varName;
```

// export by default function :
```
    export default functionName;
```
// export by default a class :
```
    export default className;
```
```

```
// export  by default a collection  :
```
```
    export default {
        varName,
        functionName,
        className
        };

```
```
// export an anonymous collection :
```
```
    export default {
        // arrow function
        x:()=>{


        },
        // anonymous function :
        function(){

        },
        // value
        v1:5
    };
```
```
// import with default  :
/*
 defaultObjectName : set any name you want
*/
  import defaultObjectName,{varName  as alias , functionName  as alias   , className as alias } from 'path/file.js';

// access default object when you import all :
    import * as containerName from 'path/source.js';
    containerName.default;

// example : ---[
// html
```
```
    <script src="main.js" type="module"></script>
    <script src="app.js" type="module"></script>
```

```
// main.js
      a = 10;
    let arr = [1, 2, 3, 4, 5]; // Fix the array declaration

    function saySomething() {
        return "nice to meet you";
    }
    let k = 10;
    export { a, arr, saySomething };

    export default {
        hi() {
```

```
            console.log("hello world");
        },
        by() {
            console.log("good by");
        },
        v: 5,
        function() {
            console.log("anonymous function ");
        },
    };

// app.js :
    console.log("\nNamed export : ");
    import * as all from "./main.js";
    console.log(all.a);
    console.log(all.arr);
    console.log(all.saySomething());

    console.log("\ndefault export : ");
    all.default.hi();
    all.default.by();
    console.log("the value : ", all.default.v);
    all.default.function();

// Json : ------------------------[]
/*
    What Is JSON ?
    - JavaScript Object Notation
    - Format For Sharing Data Between Server And Client
    - JSON Derived From JavaScript
    - Alternative To XML
    - File Extension Is .json

    Why JSON ?
    - Easy To Use And Read
    - Used By Most Programming Languages And Its Frameworks
    - You Can Convert JSON Object To JS Object And Vice Versa

    JSON vs XML
    ================================================
    = Text Based Format      = Markup Language     =
    = Lightweight            = Heavier             =
    = Does Not Use Tags      = Using Tags          =
    = Shorter                = Not Short           =
    = Can Use Arrays         = Cannot Use Arrays   =
    = Not Support Comments   = Support Comments    =
    ================================================
*/
/*
    JSON Syntax
    - Data Added Inside Curly Braces {  }
    - Data Added With Key : Value
    - Key Should Be String Wrapped In Double Quotes
    - Data Separated By Comma
    - Square Brackets [] For Arrays
    - Curly Braces {} For Objects
```

```
    Available Data Types
    - String
    - Number
    - Object
    - Array
    - Boolean Values
    - null
*/
//example : test.json
```
```
{
    "string": "name",
    "number": 100,
    "object": {
        "EG": "Giza",
        "Mr": "Rabat"
    },
    "array": ["Html", "Css", "Js"],
    "boolean": true,
    "null": null
}
```
```
/*
    -- JSON.parse = > convert JSON to Js object
    -- JSON.stringify => convert JS object to JSON
*/
    const ServerObj = '{"username":"ayoub","age":"39"}';
    console.log(typeof ServerObj);

    const JsObj = JSON.parse(ServerObj);
    console.log(JsObj);

    JsObj["username"] = "amine";

    console.log(JsObj);

    let newUpdatedServerData = JSON.stringify(JsObj);

    console.log(newUpdatedServerData);
//  Asynchronous vs Synchronous Programming : ------------------------[]
/*
    To Understand Ajax, Fetch, Promises

    Asynchronous vs Synchronous Programming
    - Meaning

    Synchronous
    - Operations Runs in Sequence
    - Each Operation Must Wait For The Previous One To Complete
    - Story From Real Life

    Asynchronous
    - Operations Runs In Parallel
    - This Means That An Operation Can Occur while Another One Is Still Being Processed
    - Story From Real Life
```

```
4086
4087      - Facebook As Example
4088      - Simulation
4089
4090      Search
4091      - JavaScript Is A Single-Threaded
4092      - Multi Threaded Languages
4093  */
4094  // example : ---[
4095      // Synchronous
4096      console.log("1");
4097      console.log("2");
4098      window.alert("Operation");
4099      console.log("3");
4100
4101      // Asynchronous
4102      console.log("1");
4103      console.log("2");
4104      setTimeout(() => console.log("Operation"), 3000);
4105      console.log("3");
4106  // Call Stack || Stack Trace :  ------------------------[]
4107  /*
4108      To Understand Ajax, Fetch, Promises
4109      -- JavaScript Engine Uses A Call Stack To Manage Execution Contexts
4110      -- Mechanism To Make The Interpreter Track Your Calls
4111      -- When Function Called It Added To The Stack
4112      -- When Function Executed It Removed From The Stack
4113      -- After Function Is Finished Executing The Interpreter Continue From The Last Point
4114      -- Work Using LIFO Principle => Last In First Out
4115      -- Code Execution Is Synchronous.
4116      -- Call Stack Detect Web API Methods And Leave It To The Browser To Handle It
4117
4118      Web API
4119      -- Methods Available From The Environment => Browser
4120      -- js start with asynchronous  code
4121   */
4122  // example :
4123   // web api method (it will be called in  event loop queue later )
4124      setTimeout(() => {
4125          console.log("Web API");
4126      }, 0);
4127
4128      function one() {
4129          console.log("One");
4130      }
4131      function two() {
4132          one();
4133          console.log("Two");
4134      }
4135      function three() {
4136          two();
4137          console.log("Three");
4138      }
4139      three();
4140
4141  /*
```

```
    +--------------+
    |   one()    |
    |   two()    |
    |   three()  |
    +--------------+
    Each function (one(), two(), three()) is represented as a block.
    The blocks are stacked on top of each other, indicating the call stack.
    The topmost block represents the function at the top of the call stack (three()),
    and the bottommost block represents the function at the bottom of the call stack (one())
.
    The arrows indicate the flow of execution, with three() calling two(), which in turn
calls one().
*/

//  Event Loop + Callback Queue :  ------------------------[]

/*
    To Understand Ajax, Fetch, Promises
    Story
    - JavaScript Is A Single Threaded Language "All Operations Executed in Single Thread"
    - Call Stack Track All Calls
    - Every Function Is Done Its Poped Out
    - When You Call Asynchronous Function It Sent To Browser API
    - Asynchronous Function Like Settimeout Start Its Own Thread
    - Browser API Act As A Second Thread
    - API Finish Waiting And Send Back The Function For Processing
    - Browser API Add The Callback To Callback Queue
    - Event Loop Wait For Call Stack To Be Empty
    - Event Loop Get Callback From Callback Queue And Add It To Call Stack
    - Callback Queue Follow FIFO "First In First Out" Rule
*/
// Example :
    console.log("One");
    setTimeout(() => {
        console.log("Three");
    }, 0);

    setTimeout(() => {
        console.log("Four");
    }, 0);

    console.log("Two");
    setTimeout(() => {
        console.log(myVar);
    }, 0);

    let myVar = 100;
    myVar += 100;
//   Ajax :  ------------------------[]
/*

  - Ready State => Status Of The Request
  [0] Request Not Initialized
  [1] Server Connection Established
  [2] Request Received
  [3] Processing Request
  [4] Request Is Finished And Response Is Ready
```

```
  - Status
  [200] Response Is Successful
  [404] Not Found
*/

    // create an object from XMLHttpRequest class :
    let request=new XMLHttpRequest();

    //  open function :  to  Prepare a request.: obj.open("type","url");
        request.open("get","url");

    // specify  the type of  response :
        request.responseType="json";

    // send the request :
        request.send();

    // run function when the response arrived :
        request.onload=function(){
        // code
        }

    /*
        Ajax
        - Ready State => Status Of The Request
            [0] Request Not Initialized
            [1] Server Connection Established
            [2] Request Received
            [3] Processing Request
            [4] Request Is Finished And Response Is Ready
        - Status
            [200 -300] Response Is Successful
            [404] Not Found [url error]
            [500 >] server error
    */
    // return the status of the response :
        request.status
    // return the readyStat of the request :
        request.readyState
    // run a function when the  stat of request change  : -- [
        request.onreadystatechange=function(){
            console.log("CHANGE ");
        };


    // add headers params :
        xhr.setRequestHeader("HeaderName1", "HeaderValue1");
        xhr.setRequestHeader("HeaderName2", "HeaderValue2");
    // Example  :
        // Set the Content-Type header if you're sending JSON data
        xhr.setRequestHeader("Content-Type", "application/json");

    // Create an object with the data you want to send
        var data = {
            key1: "value1",
            key2: "value2"
```

```
        };

    // Convert the object to a JSON string
        var jsonData = JSON.stringify(data);

    // Send the JSON data in the body of the request
        xhr.send(jsonData);

    // set response type json in postman
    /*
        headers : =>"Accept:"application/json"
    */

    // set headers params:
        request.setRequestHeader('key','value');

    // set response type :
        request.setRequestHeader("Accept","application/json");
    // set request type :
        request.setRequestHeader("Content-Type","application/json");

    // send request in  post mode
        let bodyParams={
        "key1":"value1",
        "key2":"value2",
        "key3":"value3"
        }

    // if you send json you will get  stat=500
        request.send(bodyParams);
    // you need to convert it to string :
        request.send(JSON.stringify(bodyParams));

    // type of request :
    /*
        get  : get info
        post : send info
        put type : updated all  information :
        patch : updated specific  information :
    */
    // get with filtering :
        request.open("GET","url?queryParam=value");

    //Example 1 :  --[
        const xhr = new XMLHttpRequest();
        xhr.open('GET', 'https://api.example.com/data', true);

        xhr.onreadystatechange = function () {
            if (xhr.readyState === 4) { // State 4: Done
            if (xhr.status === 200) { // HTTP status code 200 (OK)
                // Process the response data
                console.log(xhr.responseText);
            } else {
                // Handle errors or non-200 status codes
                console.error('Request failed with status:', xhr.status);
            }
```

```
            }
        };

        xhr.send(); // Initiate the request

    // Example 2 : ---------------------------------[-]
    function getPosts() {
        let request = new XMLHttpRequest();

        request.open("GET", "https://jsonplaceholder.typicode.com/posts");
        request.responseType = "json";
        request.send();

        request.onload = function () {
        if (request.status < 200 || request.status > 300) alert("server error");
        else {
            let posts = request.response;

            for (element of posts) {
            document.getElementById(
                "content"
            ).innerHTML += `<h2>${element.title}</h2>`;
            }
        }
        };
    }

    //getPosts();
    function createNewPost() {
        let request = new XMLHttpRequest();
        request.open("POST", "https://jsonplaceholder.typicode.com/posts");
        request.setRequestHeader("Accept", "application/json");
        request.setRequestHeader("Content-Type", "application/json");
        request.responseType = "json";
        let bodyParams = {
        title: "my task",
        body: "go sleep",
        userId: 1,
        };

        // send the request :
        request.send(JSON.stringify(bodyParams));

        request.onload = function () {
        if (request.status < 200 || request.status > 300) alert("server error");
        else {
            let post = this.response;
            console.log(post);
            alert("the post has been created successfully ");
        }
        };
    }

    function updatePost() {
        let request = new XMLHttpRequest();
        request.open("PUT", "https://jsonplaceholder.typicode.com/posts/1");
```

```
        request.setRequestHeader("Accept", "application/json");
        request.setRequestHeader("Content-Type", "application/json");
        request.responseType = "json";
        let bodyParams = {
        title: "hello world",
        body: "bar",
        userId: 1,
        };

        // send the request :
        request.send(JSON.stringify(bodyParams));

        request.onload = function () {
        if (request.status < 200 || request.status > 300) alert("server error");
        else {
            let post = this.response;
            console.log(post);
            alert("the post has been updated successfully ");
        }
        };
    }

    function deletePost() {
        let request = new XMLHttpRequest();
        request.open("DELETE", "https://jsonplaceholder.typicode.com/posts/1");
        request.setRequestHeader("Accept", "application/json");

        // send the request :
        request.send();

        request.onload = function () {
        if (request.status < 200 || request.status > 300) alert("server error");
        else alert("the post has been deleted successfully ");
        };
    }

    function getPostsWithFiltering() {
        let request = new XMLHttpRequest();

        request.open("GET", "https://jsonplaceholder.typicode.com/posts/?userId=1");
        request.responseType = "json";
        request.send();

        request.onload = function () {
        if (request.status < 200 || request.status > 300) alert("server error");
        else {
            let posts = request.response;

            for (element of posts) {
            document.getElementById(
                "content"
            ).innerHTML += `<h2>${element.title}</h2>`;
            }
        }
        };
    }
```

```
4421
4422 //  Pyramid Of Doom || Callback Hell : ------------------------[]
4423 /*
4424     To Understand Ajax, Fetch, Promises
4425
4426     - What Is Callback
4427     - Callback Hell Example
4428
4429     What Is Callback
4430     - A Function That Is Passed Into Another One As An Argument To Be Executed Later
4431     - Function To Handle Photos
4432     --- [1] Download Photo From URL
4433     --- [2] Resize Photo
4434     --- [3] Add Logo To The Photo
4435     --- [4] Show The Photo In Website
4436 */
4437 // example : --[
4438     setTimeout(() => {
4439         console.log("Download Photo From URL");
4440         setTimeout(() => {
4441         console.log("Resize Photo");
4442         setTimeout(() => {
4443             console.log("Add Logo To The Photo");
4444             setTimeout(() => {
4445             console.log("Show The Photo In Website");
4446             }, 1000);
4447         }, 1000);
4448         }, 1000);
4449     }, 1000);
4450 // Promise : ------------------------[]
4451 /*
4452     - Promise In JavaScript Is Like Promise In Real Life
4453     - Promise Is Something That Will Happen In The Future
4454     - Promise Avoid Callback Hell
4455     - Promise Is The Object That Represent The Status Of An Asynchronous Operation And Its
    Resulting Value
4456
4457     - Promise Status
4458     --- Pending: Initial State
4459     --- Fulfilled: Completed Successfully
4460     --- Rejected: Failed
4461
4462     Story
4463     - Once A Promise Has Been Called, It Will Start In A Pending State
4464     - The Created Promise Will Eventually End In A Resolved State Or In A Rejected State
4465     - Calling The Callback Functions (Passed To Then And Catch) Upon Finishing.
4466
4467     - Then
4468     --- Takes 2 Optional Arguments [Callback For Success Or Failure]
4469 */
4470 // Example : --[
4471     // method 1
4472     const myPromise = new Promise((resolveFunction, rejectFunction) => {
4473         let connect = false;
4474         if (connect) {
4475             resolveFunction("Connection Established");
```

```
        } else {
            rejectFunction(Error("Connection Failed"));
        }
    }).then(
        (resolveValue) => console.log(`Good ${resolveValue}`),
        (rejectValue) => console.log(`Bad ${rejectValue}`)
    );

    // method 2 :
    myPromise = new Promise((resolveFunction, rejectFunction) => {
        let connect = true;
        if (connect) {
            resolveFunction("Connection Established");
        } else {
            rejectFunction(Error("Connection Failed"));
        }
    });

    console.log(myPromise);

    let resolver = (resolveValue) => console.log(`Good ${resolveValue}`);
    let rejecter = (rejectValue) => console.log(`Bad ${rejectValue}`);

    myPromise.then(resolver, rejecter);

    myPromise.then(
        (resolveValue) => console.log(`Good ${resolveValue}`),
        (rejectValue) => console.log(`Bad ${rejectValue}`)
    );

    myPromise.then(
        (resolveValue) => console.log(`Good ${resolveValue}`),
        (rejectValue) => console.log(`Bad ${rejectValue}`)
    );

// then && catch && finally : --[
/*
    Promise Training

    We Will Go To The Meeting, Promise Me That We Will Find The 4 Employees
    .then(We Will Choose Two People)
    .then(We Will Test Them Then Get One Of Them)
    .catch(No One Came)

    Then    => Promise Is Successful Use The Resolved Data
    Catch   => Promise Is Failed, Catch The Error
    Finally => Promise Successful Or Failed Finally Do Something
*/
// example :
    const promise = new Promise((resolve, reject) => {
        let emp = ["amine", "ayoub", "kamal", "amina"];
        if (emp.length === 4) {
            resolve(emp);
        } else {
            reject(Error("Number of Employees Is Not 4 : "));
        }
```

```
    })
        .then((resolve) => {
            resolve.length = 2;
            return resolve;
        })
        .then((resolve) => {
            resolve.length = 1;
            return resolve;
        })
        .then((resolve) => console.log(`Final resolver : ${resolve}`))
        .catch((error) => {
            console.log(error);
        })
        .finally(() => console.log("Good by"));;

// Promise And XHR  : ---[
    const getData = (apiLink) => {
        return new Promise((resolve, reject) => {
            let myRequest = new XMLHttpRequest();

            // receive the response :
            myRequest.onload = function () {
                if (this.status === 200) {
                    resolve(this.response);
                } else {
                    reject(Error("Status", this.status));
                }
            };

            myRequest.open("GET", apiLink);
            myRequest.responseType = "json";
            // send the request to the server :
            myRequest.send();
        });
    };

    const apiLink = "https://api.github.com/users/elzerowebschool/repos";

    getData(apiLink)
        .then((response) => {
            response.length = 10;
            return response;
        })
        .then((response) => {
            for (let item of response) {
                let div = document.createElement("div");
                let text = document.createTextNode(item.full_name);

                div.append(text);
                div.style.cssText = `
            padding :20px;
            margin : 20px auto;
            with : 70%;
            min-width : 250px;
            text-align : center;
            border: 2px doted black;
```

```
            `;
                document.body.appendChild(div);
            }
        })
        .catch((error) => console.log(error));

//  Fetch API : ------------------------[]
/*
    fetch is a JavaScript function that allows you to make network requests
    (typically HTTP requests) to
    fetch resources from a network, such as JSON data from a REST API, HTML
    from a website, or other types
    of data. It's widely used in modern web development for making asynchronous
    requests to web servers.
*/
  fetch(url, options)
/*
    ✨url: The URL of the resource you want to fetch.
    options (optional): An object containing various options for the request,
    including the HTTP method, headers, request body, and more.
    ✨Creating a Request:
    When you call fetch, it creates and returns a Promise that represents the future response to the request.
    However, the request is not sent immediately; it's only prepared at this stage.

    ✨Configuring the Request:
    You can specify various options in the options object to configure the request:

        ✨method: The HTTP method (e.g., 'GET', 'POST', 'PUT', 'DELETE') to use for the request.
            headers: An object containing the HTTP headers for the request, such as 'Content-Type' and
            'Authorization'.

        ✨body: The request body, typically used for sending data in POST or PUT requests.
            It should be a string or a FormData object.

        ✨mode: The request mode (e.g., 'cors', 'no-cors', 'same-origin') that defines how cross-origin
        requests are handled.

        ✨credentials: Indicates whether to include cookies or credentials with the request
        ('same-origin', 'include', 'omit').

        ✨cache: The caching mode for the request ('default', 'no-store', 'reload', etc.).

        ✨redirect: How to handle redirects ('follow', 'error', 'manual').
        And more.
*/

//✨Sending the Request: --[
/*
    To actually send the request, you need to call .then() or use async/await on the returned Promise.
    This initiates the network request to the specified URL with the provided options.
*/
fetch(url, options)
```

```
  .then(response => {
    // Handle the response here
  })
  .catch(error => {
    // Handle errors here
  });

//✨Handling the Response: --[
/*
    Once the request is sent, the fetch function returns a Promise that resolves
    with a Response object representing the response from the server.
    You can then use methods and properties of this Response object to handle the response data.

    Common methods and properties of the Response object include:
        .json(): Parses the response body as JSON.
        .text(): Reads the response body as text.
        .blob(): Returns the response body as a binary Blob.
        .headers: Access to the response headers.
        .status: HTTP status code (e.g., 200 for OK, 404 for Not Found).
        .statusText: HTTP status message (e.g., "OK", "Not Found").
*/


//✨Handling Errors: --[
/*
    If the network request fails or encounters an error (e.g., due to a network issue,
    invalid URL, or server error), the Promise is rejected, and you can catch the error using .catch().
    Here's an example of using fetch to make a GET request and handle the response:

*/
// Example 1:
    fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => {
        if (!response.ok) {
        throw new Error('Network response was not ok');
        }
        return response.json(); // Parse response body as JSON
    })
    .then(data => {
        console.log(data); // Process the JSON data
    })
    .catch(error => {
        console.error('Fetch error:', error);
    });

//Example 2 :
fetch("https://api.github.com/users/elzerowebschool/repos")
    .then((result) => {
        if (result.ok) return result.json();

        throw new Error(result.statusText);
    })
    .then((myData) => {
        myData.length = 10;
        return myData;
```

```
    })
    .then((myData) => {
        for (let item of myData) {
            let div = document.createElement("div");
            let text = document.createTextNode(item.full_name);

            div.append(text);
            div.style.cssText = `
    padding :20px;
    margin : 20px auto;
    with : 70%;
    min-width : 250px;
    text-align : center;
    border: 2px doted black;
    `;
            document.body.appendChild(div);
        }
    });

// post Request steps : --[
    // Define the URL for the API endpoint
    const Url = 'https://api.example.com/endpoint';

    // Create an object with the data you want to send in the request body
    const data = {
    param1: 'value1',
    param2: 'value2'
    };

    // Create the request options, including method, headers, and body
    const requestOptions = {
    method: 'POST', // or 'GET', 'PUT', 'DELETE', etc.
    headers: {
        'Content-Type': 'application/json' // specify the content type if sending JSON data
        // Add any other headers if needed
    },
    body: JSON.stringify(data) // Convert the data object to a JSON string
    };

    // Use the fetch function to make the request
    fetch(Url, requestOptions)
    .then(response => {
        // Check if the request was successful (status code 200-299)
        if (!response.ok) {
        throw new Error(`HTTP error! Status: ${response.status}`);
        }
        // Parse the response JSON
        return response.json();
    })
    .then(data => {
        // Do something with the data returned from the API
        console.log(data);
    })
    .catch(error => {
        // Handle errors
        console.error('Fetch error:', error);
```

```
    });

// Some useful Classes : --[
    "1. ** Headers` Class**:"
    //  The `Headers` class represents a collection of HTTP headers associated with a
    //  request or response. You can use it to manipulate headers before sending a request
    //   or after receiving a response.
      headers = new Headers();
    headers.append('Content-Type', 'application/json');
    headers.set('Authorization', 'Bearer Token');


    "2. **`FormData` Class**:"
    //  The `FormData` class allows you to create and manipulate form data that can be sent in a
    //  network request. You can use it to build and send form data in a POST request.
     formData = new FormData();
    formData.append('username', 'john_doe');
    formData.append('password', 'secure_password');


    "3. **`URL` and `URLSearchParams` Classes**: "
    //  The `URL` class represents a URL, and the `URLSearchParams` class is used for working
    //  with URL query parameters. You can use these classes to parse and manipulate URLs.
     url = new URL('https://example.com/api');
    url.searchParams.append('param1', 'value1');
    url.searchParams.append('param2', 'value2');

// Example : --[
    // Create a new FormData object
    const formData = new FormData();

    // Append form data fields
    formData.append('username', 'john_doe');
    formData.append('password', 'secretpassword');

    // Create a new URL object with the endpoint
    const url = new URL('https://api.example.com/login');

    // Create a new Headers object and set headers
    const headers = new Headers();
    headers.append('Content-Type', 'application/x-www-form-urlencoded'); // Example header

    // Make a fetch request with form data, URL, and headers
    fetch(url, {
    method: 'POST', // Specify the HTTP method
    headers: headers, // Pass the Headers object with headers
    body: formData, // Pass the FormData object as the body
    })
    .then(response => {
        if (!response.ok) {
        throw new Error('Network response was not ok');
        }
        return response.json(); // Parse the JSON response
    })
    .then(data => {
```

```
        console.log('Login successful:', data);
    })
    .catch(error => {
        console.error('Error logging in:', error);
    });

// Async :  ------------------------[]
/*
    - Async Before Function Mean This Function Return A Promise
    - Async And Await Help In Creating Asynchronous Promise Behavior With Cleaner Style
*/
// method 1 : new Promise Method
function getData() {
    return new Promise((res, rej) => {
        let users = [];
        if (users.length > 0) {
        res("Users Found");
        } else {
        rej("No Users Found");
        }
    });
    }
    getData().then(
    (resolvedValue) => console.log(resolvedValue),
    (rejectedValue) => console.log("Rejected " + rejectedValue)
    );

// method 2 : using Promise.[resolve | reject]
    function getData() {
    let users = ["Osama"];
    if (users.length > 0) {
        return Promise.resolve("Users Found");
    } else {
        return Promise.reject("No Users Found");
    }
    }

    getData().then(
    (resolvedValue) => console.log(resolvedValue),
    (rejectedValue) => console.log("Rejected " + rejectedValue)
    );

//method 3 : using async function
    async function getData() {
        let users = [];
        if (users.length > 0) {
        return "Users Found";
        } else {
        throw new Error("No Users Found");
        }
    }
    getData().then(
        (resolvedValue) => console.log(resolvedValue),
        (rejectedValue) => console.log("Rejected " + rejectedValue)
    );
```

```
//  Await : ------------------------[]
/*
    - Await Works Only Inside Asnyc Functions
    - Await Make JavaScript Wait For The Promise Result
    - Await Is More Elegant Syntax Of Getting Promise Result
*/
// examples :
    myPromise = new Promise((resolve, reject) => {
        setTimeout(() => {
        // resolve("Iam The Good Promise");
        reject(Error("Iam The Bad Promise"));
        }, 3000);
    });

    async function readData() {
        console.log("Before Promise");
        // myPromise.then((resolvedValue) => console.log(resolvedValue));
        // console.log(await myPromise);
        console.log(await myPromise.catch((err) => err));
        console.log("After Promise");
    }

    readData();

//  Try, Catch And Finally With Fetch : ------------------------[]
// example :
        myPromise = new Promise((resolve, reject) => {
            setTimeout(() => {
            resolve("Iam The Good Promise");
            }, 3000);
        });

        async function readData() {
            console.log("Before Promise");
            try {
            console.log(await myPromise);
            } catch (reason) {
            console.log(`Reason: ${reason}`);
            } finally {
            console.log("After Promise");
            }
        }

        readData();

        // "https://api.github.com/users/elzerowebschool/repos"

        async function fetchData() {
            console.log("Before Fetch");
            try {
            let myData = await fetch("https://api.github.com/users/elzerowebschool/repos");
            console.log(await myData.json());
            } catch (reason) {
            console.log(`Reason: ${reason}`);
            } finally {
            console.log("After Fetch");
```

```
            }
        }
        fetchData();


//axios in js :  get Users Using axios library :  ------------------------[]
    const axios = require("axios"); // Import Axios in a Node.js environment

// Example: Making a POST request with request body parameters
    axios.post("https://example.com/api/resource", {
    key1: "value1",
    key2: "value2",
    }, {
    headers: {
        // Define your custom headers here
        "Content-Type": "application/json", // Set the appropriate content  type
        Authorization: "Bearer your-access-token", // Optional: Include an authorization
header
    },
    })
    .then((response) => {
        console.log(response.data); // Process the data from the response
    })
    .catch((error) => {
        console.error("Axios error:", error);
    });


// auto catch of errors :  ------------------------[]
    function getUsersAxios() {
        return new Promise((resolve, reject) => {
        axios
            .get(Users)
            .then((response) => {
            return response.data;
            })
            .then((Users) => {
            console.log(Users);
            resolve();
            })
            .catch((error) => {
            reject(error);
            });
        });
    }

    function getPostsAxios(UserId) {
        let PostUrl = posts + UserId;

            axios
            .get(PostUrl)
            .then((response) => {
                return response.data;
            })
            .then((Posts) => {
                console.log(Posts);
```

```
            })
            .catch((error) => {
                reject(error);
            });

    }

    getUsersAxios()
        .then(() => {
        return getPostsAxios(1);
        })
        .catch((error) => {
        console.log("Error From :", error);
        })

// npm  node package manager :  -------------------------[]
/*
    ✨download node js :  node -v to know the version
    ✨initialize the project => npm init
    ✨ install library : npm install libraryName --save(save in in package.json)
    ✨package.json : information about library that you have installed :
    ✨node modules : contain the code of  all library
    ✨lock.json specify version  of  libraries :
*/
// ✨important information :  -------------------------[]
/*
    if you use just  write :  npn install
    the npm will take information library
    in the package file then it will install it and add  to your
    project
    very helpful when you work in a team and you want to install the library
    of  your team to work with,just  you  need to get the package file
    then write the npm install command then the npm will install
    all library  in the package.json with the same specification :
*/

//✨last step  import axios from node modules  ✔ -------------------------[]
/**
 *
    you use the require just when you  working with frameworks :(Angular,React...)

    due to we work just with pure js we need to import the axios.js
    manually
    like this :
    <script src="../node_modules/axios/axios.js"></script>
    or :
    <script src="../node_modules/axios/dist/axios.min.js"></script>
*/


// await and async : -------------------------[]
// simple way to get Users and Posts using simple fetch
// without apply await ans async keyWord

/*
```

```
    but firstly let's introduce the two fundamentals :
    await: to keep the js await until an async code finished
    it's veery useful when you handel with api settimeout ..
    and many foundations in js :
    important notion about it :

    be carefully because  you can use await only in an async function :

    let's   freaking out the second concept :
    async : this keyword using the define an async function :
    and it's provide as to use the async keyword into  functions
    and make the function automatically return   Promise
    and the return keyword Represent the Resolve() function in the promise

    async functionName(){
    async code

    return  Anything   // resolve(Anything)
    }

*/

// version 1:  ------------------------[]
/*
    in this version bellow  : the code is writing
    just with fetch functions without using  await and async
    keyword

    how can see that's the implementation of it
    it's a little bit difficult to Read an maintain :
    specially  when you work on multiple then in the same promise :
    and for that the await an async founded to solve this problem exactly
    features of await and async  :

    1-more readability of code :
    2- easy the maintain
*/
    let usersUrl = "https://jsonplaceholder.typicode.com/users";
    let postsUrl = "https://jsonplaceholder.typicode.com/posts/?userId=";

    /*
        now let's develop  this code :  and make it more useful
        by  Provide  to getUsers first then get  Posts after it :
        following the order :
    */
        function getUsers() {
            // fetch the data from api placeholder api :
            return new Promise((resolve, reject) => {
                fetch(usersUrl)
                    .then((response) => {
                        // check status if ok or not :

                        if (!response.ok) throw new Error(response.statusText);

                        return response.json();
                    })
```

```
                    .then((Users) => {
                        console.log("All Users : ", Users);
                        resolve("Success to get User Response : ");
                    })
                    .catch((error) => {
                        console.log(error);
                        reject("failed to get User Response there has been an error");
                    });
            });
        }

        function getPosts(UserId) {
            // fetching data from basic PostsUrl +UserId;
            return new Promise((resolve, reject) => {
                fetch(postsUrl + UserId)
                    .then((response) => {
                        // check status if ok or not :

                        if (!response.ok) throw new Error(response.statusText);

                        return response.json();
                    })
                    .then((posts) => {
                        console.log(`Posts  Related to User[${UserId}]`, posts);
                        resolve("Success to get the  API response  ");
                    })
                    .catch((error) => {
                        console.log(error);
                        reject("there has been an error during  fetching the API Response");
                    });
            });
        }

        getUsers()
            .then(() => getPosts(1))
            .catch((error) => {
                console.log("Error :", error);
            });

    // In the provided code:
        async function fetchData() {
            try {
            let response = await fetch('https://api.example.com/data');
            let data = await response.json();
            return data;
            } catch (error) {
            console.error('Error fetching data:', error);
            throw error;
            }
        }

// version 2 with await  and async functions  : -------------------------[]
    async function getUsers() {
        let response = await fetch('https://jsonplaceholder.typicode.com/users');

        if (!response.ok)  return response.statusText;
```

```
        let Users = await response.json();

        console.log('Users',Users);
        return;
    }

    let PostUrl = "https://jsonplaceholder.typicode.com/posts/?userId=";

    async function getPosts(UserId) {
        let response = await fetch(PostUrl + UserId);

        if (!response.ok)  return response.statusText;
        let Posts = await response.json();

        console.log(`Posts[${UserId}]`, Posts);
        return;
    }

    async function getData(){

    await  getUsers();
        getPosts(1);

    }

    getData();


  /*
    The `throw error;` statement inside the `catch` block is throwing the error again
    after it has been logged. When an error is
    thrown within a `catch` block, it propagates the error up the call stack. In the
    context of an `async` function like `fetchData()`,
        if you call `fetchData()`, and an error occurs during the execution of `fetchData()
`,
        the function will reject with the thrown error.

    Here's how it works:

    1. The `fetch` API is used to make an HTTP request to 'https://api.example.com/data'.
    2. If the request fails (for example, due to network issues or an invalid URL),
    `fetch` will reject with an error.

    3. The `await fetch(...)` expression inside the `try` block will throw an error.
    4. The code inside the `catch` block will execute, logging the
    error to the console using `console.error('Error fetching data:', error);`.

    5. After logging the error, `throw error;`
    re-throws the error, causing the `fetchData()` function to reject with this error.

    When you call `fetchData()`, you can handle the rejection
    by using `.catch()` or `try/catch` blocks in the calling code. For example:

  */
  fetchData()
    .then(data => {
```

```
      // Handle successful data retrieval
      console.log('Data:', data);
    })
    .catch(error => {
      // Handle the error from fetchData() here
      console.error('Error in fetchData():', error);
    });
/*
    In this case, if there's an error during the execution of `fetchData()`,
    it will be caught in the `.catch()` block, where you can handle
    it appropriately.
*/

// example 2:  --[
    let titles = document.querySelectorAll("h1");

    function changeVisibility(index) {
        return new Promise((resolve) => {
            setTimeout(() => {
                titles[index].style.visibility = "visible";
                resolve(index);
            }, 1000);
        });
    }

    async function ChangeTitlesVisibility() {
        for (let i = 0; i < titles.length; i++) {
            await changeVisibility(i);

        }
    }

ChangeTitlesVisibility();

//Authentication : -------------------------[]
//https://reqres.in/
//type of token :
/*
    Bearer token : (headers)  Authorization =Bearer token
    normal token : (headers)  Authorization = token
*/
//login :
    let loginUrl = "https://reqres.in/api/login";
    let UserUrl = "https://reqres.in/api/users";

    let loginToken= localStorage.getItem("userToken") || "";
    function login() {
        axios
            .post(loginUrl, {
                email: "tracey.ramos@reqres.in",
                password: "cityslicka",
            })
            .then((response) => response.data)

            .then((Token) => {
                console.log(Token);
```

```
                loginToken = Token.token;
                localStorage.setItem("userToken",loginToken);
                createNewUser();
            })
            .catch((error) => {
                alert(error);
            });
    }


    function createNewUser() {
        let config = {
            headers: {
                "Authorization": "Bearer " + loginToken,
            },
        };

        axios
            .post(
                UserUrl,
                {
                    name: "majid",
                    job: "leader",
                },
                config
            )
            .then((response) => response.data)
            .then((newUserInfo) => {
                console.log(newUserInfo);
            })
            .catch((error) => {
                console.log(error);
            });
    }

// using await and async  function :  ------------------------[]
       loginUrl = "https://reqres.in/api/login";
       UserUrl = "https://reqres.in/api/users";
    let registerUrl = "https://reqres.in/api/register";

     loginToken = localStorage.getItem("userToken") || "";

    let bodyPrams = {
        email: "tracey.ramos@reqres.in",
        password: "cityslickda",
    };

    async function login() {
        try {
            let response = await axios.post(loginUrl, bodyPrams);

            let token = response.data;
            console.log(token);
            localStorage.setItem("userToken", token.token);
        } catch (error) {
            console.log("Error : ", error.message);
```

```
        }
    }

     headers = {
        Authorization: "Bearer " + loginToken,
    };

    let UserINfo = {
        name: "majid",
        job: "leader",
    };
    async function createNewUser() {
        try {
            let response = await axios.post(UserUrl, UserINfo, headers);

            let newUserInfo = response.data;
            console.log(newUserInfo);
        } catch (error) {
            console.log("Error : ", error.message);
        }
    }

    async function register() {
        try {
            let response = await axios.post(registerUrl, bodyPrams);

            let registerInfo = response.data;
            console.log(registerInfo);
            console.log("token : ", registerInfo.token);
            localStorage.setItem("userToken", registerInfo.token);
        } catch (error) {
            console.log("Error : ", error.message);
        }
    }

    async function main() {
        console.log("\nRegister new user : ");
        await register();

        console.log("\nlogin to my created account  : ");
        await login();

        console.log("\ncreate a new user using my token : ");
        await createNewUser();
    }

    main();
    login();

// delay function :  ------------------------[]
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

/*
    by using the login function we send request to api then api
```

```
    generate a token that we received  then save in local Storage

    to provide to user enter directly to his account

    without the need to login again by using  CreateUser function :
    that take Token from local Storage then created a new user
*/
/*
    To navigate to another page using JavaScript, you can use the `window.location` object.
    Here are a few common ways to achieve this:

    ### 1. Using `window.location.href`:
    You can set the `window.location.href` property to the URL of the page you want to navigate to. For example:

    ```javascript
    Navigate to a new page
    window.location.href = "https://www.example.com/newpage.html";
    ```

    ### 2. Using `window.location.assign()` method:
    The `assign()` method of the `window.location` object is another way to navigate to a new page:

    ```javascript
    Navigate to a new page
    window.location.assign("https://www.example.com/newpage.html");
    ```

    ### 3. Using `window.location.replace()` method:
    The `replace()` method of the `window.location` object can be used
    to navigate to a new page and replace the current
    page in the browser history. This means the user cannot navigate
    back to the original page using the browser's back button.

    ```javascript
    Navigate to a new page and replace the current page in the browser history
    window.location.replace("https://www.example.com/newpage.html");
    ```

    Choose the appropriate method based on your specific use case and whether you want the new
    page to be added to the browser history or replace the current page.

    Authentication and tokens are fundamental concepts in the realm of security
    and identity management, especially in the context of web applications and
    APIs. Let's explore these concepts in detail:

    ### Authentication:

    Authentication is the process of verifying the identity of a user, application,
    or system. It ensures that the entity trying to access a resource is who it claims to be
    . There are various methods of authentication, each with its own strengths and use cases:

    1. **Username/Password:**
    - The user provides a username and password.
```

- Common for web applications and traditional login systems.
- Vulnerable to various attacks, such as phishing.

2. **Multi-Factor Authentication (MFA):**
- Requires multiple forms of identification (e.g., password + SMS code or fingerprint).
- Enhances security by adding an additional layer of verification.

3. **Token-Based Authentication:**
- Uses tokens (e.g., JSON Web Tokens) for authentication.
- Reduces the need to store sensitive credentials on the client.
- Often used in modern web applications and APIs.

4. **OAuth and OpenID Connect:**
- Delegated authorization and authentication protocols.
- OAuth allows secure delegated access to resources.
- OpenID Connect is an identity layer on top of OAuth, providing authentication.

### Tokens:

Tokens play a crucial role in modern authentication and authorization systems.
They are used to represent the authenticated user and provide secure access
to protected resources. The most common types of tokens are:

1. **Access Tokens:**
- Grants access to specific resources on behalf of the user.
- Short-lived and specific to the user and application.
- Used in OAuth for authorization.

2. **Refresh Tokens:**
- Used to obtain a new access token.
- Longer-lived than access tokens.
- Stored securely on the client.

3. **JSON Web Tokens (JWT):**
- A compact, URL-safe means of representing claims between two parties.
- Self-contained, containing information about the user or system.
- Often used as access tokens in token-based authentication.

### Token-Based Authentication Flow:

1. **User Authentication:**
- The user provides credentials (e.g., username/password) to the authentication server.

2. **Token Issuance:**
- Upon successful authentication, the authentication server generates an access token
(and optionally a refresh token).

3. **Token Storage:**
- The access token is stored securely on the client (e.g., in a cookie or local storage)
.

4. **Token Usage:**
- The client includes the access token in the headers of API requests to access
protected resources.

5. **Token Expiry and Refresh:**
- Access tokens have a limited lifespan. If they expire, the client can use the refresh

```
    token to obtain a new access token without requiring the user to re-enter credentials.

    ### Benefits of Token-Based Authentication:

    1. **Statelessness:**
    - No need to store user sessions on the server.
    - Each request contains the necessary authentication information.

    2. **Scalability:**
    - Stateless nature simplifies scaling, as there's no need to synchronize session state
        across multiple servers.

    3. **Security:**
    - Tokens can be encrypted and signed to ensure integrity and confidentiality.
    - Reduced risk of Cross-Site Request Forgery (CSRF) and session hijacking.

    4. **Decoupling Frontend and Backend:**
    - The frontend and backend can be developed independently, as long as they adhere to the token contract.

    In summary, authentication verifies the identity of a user or application, while tokens play a crucial
    role in securely representing and granting access to resources. Token-based authentication,
    especially using technologies like OAuth and JWTs, has become a standard in modern web development
    due to its security, scalability, and flexibility.
 */
```