

## res\summary.js

```
1 // Link html file with script -----[]
2 // internal js :
3 ```
4     <script>
5     // code
6     </script>
7 ```
8     //external js file :
9 ```
10    <script src="path"></script>
11 ```
12
13 // comments : -----
14 // single line comments :
15 ```
16     // comment
17 ```
18 //multiple line comments :
19 ```
20     /*
21     */
22 ```
23
24 // vs code option shortcut :
25 ```
26     ctrl + / : comment selected content
27 ```
28 //output to screen : -----
29 //show msg in a pop up window L
30 ```
31     window.alert("content");
32     // or adding window not required :
33     alert("content")
34 ```
35 //add a content to the html page :
36 ```
37     document.write("content");
38 ```
39 //Log a content to the console window :
40 ```
41     console.log("content");
42 ```
43 //console methods : -----
44 //show normal content :
45 ```
46     console.log("content");
47 ```
48 // show error msg :
49 ```
50     console.error("errorContent")
51 ```
52 // show objects key : value in a table
53 ````
```

```
54     console.table(varName);
55     ``
56 // add styling to log content %c :
57     ``
58     console.log("%cContent","styling");
59     //example :
60     console.log("Hello world ","color :red; font-size:50px","color : gray; font-
size:40px");
61     ``
62 // Data Types : -----
63     ``
64     string
65     boolean
66     objects [arrays | objects]
67     function
68     Number [ integer | float ]
69     undefined
70     NaN
71     ``
72 // know the data type of a content :
73     ``
74     console.log(typeof content);
75     ``
76 // Variables : -----
77 // what's a variable :
78 /*
79     an amount of space reserved to store a value
80     and access it using a variable name
81 */
82 /*
83     javaScript is a Loosely Typed Language : Dynamic Typed
84     unlike
85     c++ that's a Strongly Typed Language
86 */
87
88 //declare a variable :
89     ``
90     //using var
91     var varName[=value];
92     //using let
93     let varName[=value];
94     ``
95 // declare a constant variable:
96     ``
97     constant varName=value;
98     ``
99 //the different between var && let :
100 /*
101     var :
102         1-the oldest one
103         2- function scope (limited just by function scope access)
104         2-a- 5-variable scope Drama(Added To window) if is not scoped by a function
105         3- provide hoisting behavior (use variable before declared)
106         4- you can redeclare it multiple type
107     let :
108         1- cams with ES6
```

```

109   2- block scope (Limited by all scope access (if ,else , function ... ) )
110   3- don't support hoisting behavior you can't use it before declare it it invokes an
111     error
112     4- you can't redeclare variable of it
113   /*
114   // id attribute in html
115   /*
116     it's considered as global variable you access it :
117   */
118   //example --[
119   ``
120     // html :
121     <h2 id="nice">nice</h2>
122     // js :
123     console.log(nice);
124   ``
125   // \ Character : -----
126   /*
127     -it used to tell to the compiler that's not a command and I want you to considered as a
128     text
129     - used a line continue
130   */
131   //syntax :--[
132   ``
133     console.log(' text\specialCharacters ')
134   ``
135   //example 1 :--[ Escape
136   ``
137     console.log('that\'s it ')
138   ``
139   //example 2 :--[ line continue
140   ``
141     console.log('hello world \
142       my name is ayoub ');
143   ``
144   // add new Line: -----
145   console.log("\n");
146   // concatenation: -----
147   // concat two string s : ---[
148   ``
149     //method 1 :
150     console.log("string1" + "string2");
151     //method 2 :
152     console.log("string1","string2");
153   ``
154   // concat string with variables :
155   ``
156     //method 1 :
157     console.log("string1" + varName);
158     //method 2 :
159     console.log("string1",VarName);
160     //method 3 : ES6 Template Literals
161     console.log(`string1 ${varName}`)
162   ``
163   // Arithmetic Operators: -----
164   ``
165     +

```

```

164      -
165      *
166      /
167      % : modulo
168      ** : power
169      ``
170 // postIncrement :--[
171
172      ``
173      // return then increment
174      varName++;
175      ``
176 // preIncrement :--[
177      ``
178      // increment then return
179      ++varName;
180      ``
181 // Unary Plus And Negation Operators: -----
182 // Unary Plus :
183      ``
184      // convert value to number
185      +value
186      ``
187 // Negation :
188      ``
189      // convert value to number then make it negative
190      -value
191      ``
192 // Type Coercion / casting : -----
193 /*
194     Type coercion, also known as type conversion or casting, is the process of converting a
195     value from one data type
196     to another in JavaScript. This happens automatically when an operation is performed
197     between values of different
198     types. JavaScript performs type coercion in various situations, such as:
199 */
200 // 1. **String Concatenation**: When using the `+` operator with strings and other data
201 //     types, JavaScript converts non-string values to strings.
202      ``
203      const num = 10;
204      const str = "The number is: " + num; // Type coercion: num is converted to a string
205      console.log(str); // Output: The number is: 10
206      ``
207 // 2. **Comparison Operators**: When using comparison operators like `==` or `!=`,
208 //     JavaScript may coerce values to a common type before comparing them.
209      ``
210      console.log(5 == "5"); // true - Type coercion: "5" is converted to a number for
211      // comparison
212      ``
213 // 3. **Logical Operators**: The logical operators `&&` and `||` return the value of one of
214 //     the specified operands, but not necessarily of type `true` or `false`.
215      ``
216      console.log(5 && "Hello"); // "Hello" - Type coercion: Both values are considered "
217      // truthy"
218      ``

```

```
215 // 4. **Explicit Type Conversion**: You can also perform explicit type conversion using
216 // functions like `String()`, `Number()`, `Boolean()`, etc.
217 ```
218     const numStr = "10";
219     const num = Number(numStr); // Explicit conversion from string to number
220     console.log(num); // Output: 10
221 ```
222 //Assignment Operators : -----
223 ```
224     varName+=value;
225     varName-=value;
226     varName*=value;
227     varName%=value;
228     varName/=value;
229     varName**=value;
230 ```
231 // Numbers : -----
232 /*
233 -- Double Precision: JavaScript uses double-precision floating-point format to represent
234 numeric values.
235 -- Syntactic Sugar "_": Underscores can be used in numeric literals for better readability.
236 -- e Notation: Allows representing large or small numbers using exponent notation.
237 -- ** Operator: Exponentiation operator raises the left operand to the power of the right
238 operand.
239 */
240
241 console.log(1000000);           // Output: 1000000
242 console.log(1_000_000);        // Output: 1000000 (underscore for readability)
243 console.log(1e6);             // Output: 1000000 (e notation)
244 console.log(10**6);           // Output: 1000000 (exponentiation)
245
246 // Numbers functions : -----
247 /*
248 varName.toString() : convert from number to string
249 varName.toFixed() : Format number with fixed decimal places
250 parseInt(value) parse value to int
251 parseFloat(value) parse value to float
252 Number.isInteger(value) [ES6] Check if a number is an integer
253 note : you can use Number methods without put Number. before them (not all functions )
254 */
255 //example : --[
256 const num = 10;
257
258 // Convert number to string
259     console.log(num.toString()); // Output: "10"
260
261 // Format number with fixed decimal places
262     console.log(num.toFixed(2)); // Output: "10.00"
263
264 // Parse string to integer
265     console.log(parseInt("10")); // Output: 10
266
267 // Parse string to float
```

```
268     console.log(parseFloat("10.5")); // Output: 10.5
269
270 // Check if a number is an integer [ES6]
271     console.log(Number.isInteger(4)); // Output: true
272
273 // Check if a value is NaN [ES6]
274     console.log(Number.isNaN("Hello")); // Output: false
275     console.log(Number.isNaN(NaN)); // Output: true
276
277 //Math Object: -----[]
278 /*
279 . round()
280 . ceil()
281 . floor()
282 . min()
283 . max()
284 . pow()
285 . random()
286 . trunc() [Es6]
287 */
288 //examples:--[
289 // Math.round() - Rounds a number to the nearest integer.
290     console.log(Math.round(4.7)); // Output: 5
291
292 // Math.ceil() - Rounds a number up to the nearest integer.
293     console.log(Math.ceil(4.3)); // Output: 5
294
295 // Math.floor() - Rounds a number down to the nearest integer.
296     console.log(Math.floor(4.7)); // Output: 4
297
298 // Math.min() - Returns the smallest of zero or more numbers.
299     console.log(Math.min(4, 7, 2, 9)); // Output: 2
300
301 // Math.max() - Returns the largest of zero or more numbers.
302     console.log(Math.max(4, 7, 2, 9)); // Output: 9
303
304 // Math.pow() - Returns the base to the exponent power.
305     console.log(Math.pow(2, 3)); // Output: 8
306
307 // Math.random() - Returns a random number between 0 (inclusive) and 1 (exclusive).
308     console.log(Math.random()); // Output: a random number between 0 and 1
309
310 // Math.trunc() [ES6] - Returns the integer part of a number by removing any fractional digits.
311     console.log(Math.trunc(4.7)); // Output: 4
312
313 // -String Methods -----[]
314 /*
315     -Access With Index
316     -Access With charAt()
317     -length
318     -trim()
319     -toUpperCase()
320     -toLowerCase()
321     -Chain .Methods
322 */
```

```
323 // -String Methods -----[]
324
325 /*
326     -Access With Index
327 */
328 // You can access individual characters in a string using square brackets [] and the index
329 // of the character. Indexing in JavaScript starts from 0.
330     let str = "hello";
331     console.log(str[0]); // Output: "h"
332     console.log(str[1]); // Output: "e"
333
334 /*
335     -Access With charAt()
336 */
337 // Returns the character at the specified index in a string. If the index is out of range,
338 // an empty string is returned.
339     console.log(str.charAt(0)); // Output: "h"
340     console.log(str.charAt(4)); // Output: "o"
341
342 /*
343     -Length
344 */
345 // Returns the Length of a string, i.e., the number of characters in the string.
346     console.log(str.length); // Output: 5
347
348 /*
349     -trim()
350 */
351 // Removes whitespace from both ends of a string.
352     let strWithWhitespace = "    hello    ";
353     console.log(strWithWhitespace.trim()); // Output: "hello"
354
355 /*
356     -toUpperCase()
357 */
358 // Returns the string with all its characters converted to uppercase.
359     console.log(str.toUpperCase()); // Output: "HELLO"
360
361 /*
362     -toLowerCase()
363 */
364 // Returns the string with all its characters converted to Lowercase.
365     let strUpperCase = "HELLO";
366     console.log(strUpperCase.toLowerCase()); // Output: "hello"
367
368 /*
369     -Chain .Methods
370 */
371 // You can chain multiple string methods together, with each method being called on the
372 // result of the previous one.
373     console.log(strWithWhitespace.trim().toUpperCase()); // Output: "HELLO"
374
375 /*
376     -indexOf(Value)
377     -lastIndexOf(Value)
378 */
```

```

377     -slice(Start. [start,end)
378     -repeat(Times) . [ES6]
379     -split(Separator [Opt])
380     -substring(start,end)
381     -substr(start ,characters To Extract)
382     -includes(value)
383     -startWidth(value)
384     -endWidth(value)
385 */
386
387 // ✨ -indexOf(Value, startIndex)
388 /*
389     Returns the index within the calling String object of the first occurrence of the
390     specified value,
391     starting the search at fromIndex. Returns -1 if the value is not found.
392
393     Parameters:
394         - Value: The substring to search for within the string.
395 */
396     str = "hello world";
397 console.log(str.indexOf('o')); // Output: 4
398 console.log(str.indexOf('l')); // Output: 2
399
400 // ✨ -lastIndexOf(Value, startIndex)
401 /*
402     Returns the index within the calling String object of the last occurrence of
403     the specified value, searching backwards from fromIndex. Returns -1 if the value is not
404     found.
405
406     Parameters:
407         - Value: The substring to search for within the string.
408 */
409 console.log(str.lastIndexOf('o')); // Output: 7
410 console.log(str.lastIndexOf('l')); // Output: 9
411
412 // ✨ -slice(Start, [end])
413 /*
414     Extracts a section of a string and returns it as a new string. You specify the start
415     index and optionally the end index (exclusive).
416
417     Parameters:
418         - Start: The index where to begin the extraction (inclusive).
419         - End: The index where to end the extraction (exclusive). If omitted, slice extracts
420             to the end of the string.
421 */
422 console.log(str.slice(6)); // Output: "world"
423 console.log(str.slice(0, 5)); // Output: "hello"
424
425 // ✨ -repeat(Times) . [ES6]
426 /*
427     Returns a new string containing the specified number of copies of the string on which it
428     was called.
429
430     Parameters:
431         - Times: The number of times to repeat the string.
432 */
433 let repeatedStr = "abc";
434 console.log(repeatedStr.repeat(3)); // Output: "abcabcabc"
435

```

```

429 // ⚡ -split(Separator [Opt])
430 /*
431     Splits a string into an array of substrings based on the specified separator and returns
432     the array.
433     Parameters:
434         - Separator (optional): Specifies the character(s) to use for separating the string.
435         If omitted, the entire string will be returned as the only element in the array.
436 */
437 let sentence = "This is a sentence";
438 console.log(sentence.split(' ')); // Output: ["This", "is", "a", "sentence"]
439 /*
440 - ⚡substring(start,end)
441 */
442 // Returns a new string that includes characters from the original string between the
443 // specified start and end indexes (end index not included).
444 str = "hello world";
445 console.log(str.substring(1, 4)); // Output: "ell"
446 /*
447 - ⚡substr(start ,characters To Extract)
448 */
449 // Returns the characters in a string beginning at the specified location through the
450 // specified number of characters.
451 console.log(str.substr(1, 4)); // Output: "ello"
452 /*
453 - ⚡includes(value)
454 */
455 // Determines whether a string contains the specified value. Returns true if the string
456 // contains the value, otherwise false.
457 console.log(str.includes('world')); // Output: true
458 /*
459 - ⚡startsWith(value)
460 */
461 // Determines whether a string begins with the characters of a specified string. Returns
462 // true if the string starts with the value, otherwise false.
463 console.log(str.startsWith('hello')); // Output: true
464 /*
465 - ⚡endsWith(value)
466 */
467 // Determines whether a string ends with the characters of a specified string. Returns true
468 // if the string ends with the value, otherwise false.
469 console.log(str.endsWith('world')); // Output: true
470 //assignment : --[
471 let a="Elzero Web School";
472 //1 Zero
473 console.log(a.slice(2,6).charAt(0).toUpperCase() + a.slice(3,6));
474 // 2 HHHHHHHH
475 console.log((('H').repeat(8)));
476 //3 ['Elzero']
477 console.log([a.substr(0,6)])
478 //4 Elzero School
479 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
```

```
535     break;
536 }
537 //Arrays : -----
538 /*
539 -Create Arrays [Two Methods ] new Array () | []
540 -Access Arrays Elements
541 -Nested Arrays
542 -Change Arrays Elements
543 -Check For Array Array isArray(arr);
544 */
545 //Arrays : -----
546
547 /*
548 -Create Arrays [Two Methods ] new Array () | []
549 */
550 // You can create arrays in JavaScript using two methods: using the Array constructor or
551 // using array literal notation.
552 let arr1 = new Array(); // Using Array constructor
553 let arr2 = []; // Using array literal notation
554 /*
555 -Access Arrays Elements
556 */
557 // You can access individual elements of an array using square brackets [] and the index of
558 // the element. Indexing in arrays starts from 0.
559 let fruits = ["apple", "banana", "orange"];
560 console.log(fruits[0]); // Output: "apple"
561 console.log(fruits[1]); // Output: "banana"
562 /*
563 -Nested Arrays
564 */
565 // Arrays can also contain other arrays, creating nested arrays.
566 let nestedArray = [[1, 2], [3, 4], [5, 6]];
567 /*
568 -Change Arrays Elements
569 */
570 // You can change the value of an element in an array by assigning a new value to the
571 // corresponding index.
572 fruits[0] = "pear"; // Changing "apple" to "pear"
573 /*
574 -Check For Array Array.isArray(arr);
575 */
576 // You can check if a variable is an array using the Array.isArray() method. It returns true
577 // if the variable is an array, otherwise false.
578 console.log(Array.isArray(fruits)); // Output: true
579 console.log(Array.isArray("apple")); // Output: false
580 //Arrays Methods : -----
581 // add element to the first :
582 arrName.unshift(value1,value2);
583
584 // add element to the end :
585 arrName.push();
586
587 // remove the first element L
```

```
588 arrName.shift();
589
590 // remove the last element
591 arrName.pop();
592
593 // get the index of a value from first index to last :
594 arrName.indexOf(value,startIndex=0);
595
596 //get the index of a value from last index to first :
597 arrName.lastIndexOf(value,startIndex=0);
598
599 // check if value exist :
600 arrName.includes(value);
601
602 // sort Array :
603 arrName.sort();
604 // reverse an array :
605 arrName.reverse();
606
607 // get slice array ;
608 result=arrName.slice(start,end);
609
610 // remove || add elements :
611 arrName.splice(startIndex,numberElementsToRemove,replacedValue1,replacedValueN);
612
613 // concat Arrays :
614 newArr=arr1.concat(arr2,[arr3]);
615
616 // convert from array to string :
617 str=arr.join(separator=',');
618
619 //assignment :
620 let zero = 0;
621 let counter = 3;
622
623 let my = ["Ahmed", "Mazero", "Elham", "Osama", "Gamal", "Ameer"];
624
625 // ["Osama", "Elham", "Mazero", "Ahmed"];
626 console.log(my.reverse());
627
628 console.log(my.slice(-++counter));
629
630 // ["Elham", "Mazero"]
631 console.log(my.slice(--counter, ++zero + ++counter));
632
633 // "Elzero"
634 console.log(my[--counter].substr(--zero, --counter) + my.reverse()[++zero].substring(++zero));
635 //my.reverse()[zero].substring(++zero)
636 console.log(counter, zero);
637 // r0;
638 console.log(my[--zero][my[zero].length - ++zero] + my[++zero].substring(counter - zero, --counter));
639
640 // Loops : -----
641 // for Loop :
642 // [label :]
```

```

643     for(let i=start ; i<end; iteration){
644         // code
645         // [continue | break Label];
646     }
647 //for in : index
648 for (let element of elements){
649     //code
650     // [continue | break]
651 }
652 // for of : element
653 for(let elementIndex of elements){
654     //code
655     // [continue | break]
656 }
657
658 // while Loop :
659 while(condition){
660     // code
661     // [continue | break]
662 }
663 // do while Loop :
664 do{
665     //code
666     // [continue | break]
667 }while(loop);
668
669 // foreach
670 elements.foreach((element,index)=>{
671     // code
672     // return
673 })
674
675 //assignment :-----[]
676 // optimized version :
677 const body = document.body;
678 let myAdmins = ["Ahmed", "Osama", "Sayed", "Stop", "Samera"];
679 let myEmployees = ["Amged", "Samah", "Ameer", "Omar", "Othman", "Amany", "Samia"];
680
681 let adminNumber = myAdmins.slice(0, myAdmins.indexOf("Stop") == -1 ? 0 : myAdmins.indexOf("Stop")).length;
682 body.innerHTML = `<h3>we have ${adminNumber} Admins</h3>`;
683 for (i = 0; i < adminNumber; i++) {
684     body.innerHTML += `<hr> <h4> the Admin For Team is <span style='color:red;'> ${myAdmins[i]} </span> </h4>`;
685     body.innerHTML += `<h2>Team Members :</h2>`;
686
687     let counter = 0;
688     myEmployees.forEach((element) => {
689         if (element.startsWith(myAdmins[i][0])) body.innerHTML += `<h4> -${++counter} ${element} </h4>`;
690     });
691 }
692
693 //functions :-----[]
694 function functionName(args){
695
696     // [return value]

```

```
697 }
698 //function with variables
699 let functionName=function(args){
700
701 }
702
703
704 // arrow function : (this = window always)--[
705 let functionName=()=>{
706
707 }
708 // call a function :--[
709 functionName(v1,v2,vn);
710 // default value
711 function functionName(arg1,arg2=defaultValue){ // if the user did not enter the value of the
second parameter :
712
713
714 }
715
716 // reset Parameters :--[
717 function functionName(...args){
718 // code :
719 }
720 // example :
721 function sum(...values){
722 return values.reduce((prev,current)=>prev+current);
723 }
724 // assignment: -----[]
725 function showDetails(name,age,available){
726
727     let strContent="Hello ";
728     strContent+=(typeof name ==="string" ? name : typeof age =="string" ? age : typeof
available =="string" ? available : "unknown") + ", Your Age Is ";
729     strContent+=(typeof name ==="number" ? name : typeof age =="number" ? age : typeof
available =="number" ? available : "unknown") + " Your Are ";
730     let availableStatus= typeof name ==="boolean" ? name : typeof age =="boolean" ? age : typeof
available =="boolean" ? available : "unknown";
731     strContent+=(!availableStatus ? "Not " : "") + "Available For Hire"
732     console.log(strContent);
733 }
734
735 /*
736 Hello amine, Your Age Is 20 Your Are Not Available For Hire
737 Hello amine, Your Age Is 20 Your Are Available For Hire
738 Hello amine, Your Age Is 38 Your Are Not Available For Hire
739 */
740 showDetails("amine",20,false);
741 showDetails(20,"amine",true);
742 showDetails(false,"amine",38);
743
744 // function inside function example : -----
745 function sayMessage(fName, lName) {
746     let message = `hello`;
747     function concatMsg() {
748         function getFullName() {
749             return fName + " " + lName;
```

```

750     }
751
752         return `${message} ${getFullName()}`;
753     }
754
755     return concatMsg();
756 }
757 console.log(sayMessage("ayoub", "km"));
758
759 // add a function to Array type : -----
760 // [ ]
761     Array.prototype.functionName= function(){
762         //code
763     };
764
765 // map function : return new array with specific values : -----
766 // [ ]
767     newArr=arr.map( (element,index,arr)=>{
768         //code
769     })
770
771 // map function implementation :--{
772     let mapTest=function(callBack){
773         const newArray=[];
774         this.forEach(element => {
775
776             newArray.push(callBack(element))
777         });
778         return newArray;
779     }
780     Array.prototype.mapTest = mapTest;
781
782 // or --[
783
784     Array.prototype.mapFunction=function(callback){
785         var newArray=[];
786         for(element of this){
787             newArray.push(callback(element));
788         }
789         return newArray;
790     }
791
792 arr1=[1,2,3,4,6,7,8];
793 arr2=arr1.mapFunction(function(element){
794     return element*10;
795 });
796
797 // example : return newArr=arr*10 --[
798     newArr=arr.map(element=> element *10);
799
800 // example : inverted case :
801     str="Good job";
802     console.log(str.split("").map(ele=>ele === ele.toUpperCase() ? ele.toLowerCase() : ele.toUpperCase()).join(""));
803
804 // example : inverted numbers :
805     let numbers=[1,-1,5,-20];
806     console.log(numbers.map(ele=> -ele ));
807
808 // filter function : -----
809 // this function help us to copy oldArray to new array
810 // with specific condition :

```

```

805  newArr=arr.filter( (element,index,arr)=>{
806      return condition;
807  })
808 // implementation :--[
809  Array.prototype.filterFunction = function (callback) {
810      let newArray = [];
811      this.forEach(function (element) {
812          if (callback(element)) newArray.push(element);
813      });
814
815      return newArray;
816  };
817 //example :
818  let arr3=[1,3,4,5,6,7,8];
819  let newArray1=arr1.filterFunction(function(element){
820      return element>5 ? true :false;
821
822 });
823
824 // reduce : -----
825 /*
826     this function explore element by element
827     in the end the function will return a Number :
828
829 */
830  var total=arr.reduce(function(preValue,currentValue){
831      // code
832  });
833
834 //example : --[
835  var arr=[1,3,4,5,6,8];
836  var total=arr.reduce(function(preValue,currentValue){
837
838      return preValue+currentValue;
839  });
840 // assignment: -----
841 //1 :--[
842 let theBiggest=["Bla","Propaganda","other","AAA","Battery","Test"];
843 let max=theBiggest.reduce((acc,current)=> acc.length > current.length ? acc:current);
844 console.log(max);
845 let removeChars=[ 'E','@','@','L','Z','@','@','E','R','@','0'];
846 let mot=removeChars.filter(ele=> ele != '@').reduce( (acc,current)=> `${acc}${current}`);
847 console.log(mot);
848 //2 :--[
849 let myString="1,2,3,EE,l,z,e,r,o,_W,e,b,_S,c,h,o,o,l,2,0,z";
850 let newStr= myString.split(",").filter((ele,index)=> ele!=',' && index !=myString.split(",") .length -1)
851 .map(ele=> ele == '_' ? " " : isNaN(ele) ? ele.charAt(0) : "" )
852 .reduce((acc,current)=>acc+current);
853 console.log(newStr); // Elzero web School :
854
855 // objects : -----
856 /*
857     In JavaScript, an object is a collection of key-value pairs, where each key is a string
858     (or Symbol)
859     and each value can be any data type, including arrays, functions, and other objects.

```

```

860 */
861 // syntax : --[
862   objectName={
863     key:value,
864     key:value,
865     functionName:function(){
866       //code
867     }
868     // ....
869   }
870 // or
871 objectName= new Object{
872   key:value,
873   key:value,
874   functionName:function(){
875     //code
876   }
877   // ....
878 }
879
880
881 // access to object :--[
882 //Dot notation :
883   objectName.key;
884 // Bracket notation :
885   objectName['key'];
886 // example a : --[
887 const person1 = {
888   firstName: "John",
889   lastName: "Doe",
890   age: 30,
891 };
892
893 //example b :--[
894   let student = {
895     FirstName: "ayoub",
896     lastName: "majid",
897     printObject: function() {
898       console.log("the name ", this.FirstName);
899       console.log("the lastName ",this.lastName);
900     }
901   };
902   student.printObject();
903
904 // example c :--[
905   // we can add a default value to an argument
906   const obj = {
907     name: "John",
908     sayHelloArrow: () => {
909       console.log("Hello, " + this.name); // `this` is lexically scoped (undefined in this
case)
910     },
911     sayHelloNormal: function() {
912       console.log("Hello, " + this.name); // `this` is dynamically scoped (refers to obj)
913     },
914   };

```

```

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"

```

```
971 console.log(person["address"]["zipCode"]); // Output: "10001"
972
973 // create a prototype object :
974 /*
975     The Object.create() method in JavaScript is used to create a new object with a specified
976     prototype object and optionally,
977     with specified properties. It allows you to create an object that inherits properties
978     and methods from another object,
979     known as the prototype object.
980 */
981 //syntax : --[
982     Object.create(prototypeObject);
983 //example :--[
984     person = {
985         name: "John",
986         age: 30,
987         address: {
988             street: "123 Main St",
989             city: "New York",
990             zipCode: "10001"
991         }
992     };
993
994     obj = Object.create(person);
995     console.log(obj);
996     console.log(obj.name);
997 // Object.assign method: -----
998 //--- Purpose:
999 /*
1000     The `Object.assign()` method is used to copy the values of all enumerable own properties
1001     from
1002     one or more source objects to a target object.
1003     It returns the modified target object.
1004 */
1005
1006 // Syntax:--[
1007     Object.assign(target, ...sources)
1008 // Parameters: --[
1009 /*
1010     - target: The target object to which the properties will be copied.
1011     - sources: One or more source objects whose properties will be copied to the target
1012     object.
1013 */
1014
1015 // Return value:--[
1016 // The modified target object after copying the properties from the source object(s).
1017
1018 // Example: --[
1019     let target = { a: 1, b: 2 };
1020     let source = { b: 3, c: 4 };
1021
1022     // Copy properties from source to target
1023     Object.assign(target, source);
1024
1025     console.log(target); // Output: { a: 1, b: 3, c: 4 }
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
```

```
1024 //      Notes:  
1025 /*  
1026     1. If the target object already has a property with the same name, its value will be  
1027        overwritten by the value from the source object.  
1028     2. If a source object is `null` or `undefined`, it will be ignored.  
1029     3. `Object.assign()` only copies enumerable own properties from the source objects. It  
1030        does not copy properties from the  
1031           source's prototype chain.  
1032     4. The properties are copied in the order they are listed in the `sources` parameter,  
1033        with properties from later  
1034           sources overwriting properties from earlier sources.  
1035 */  
1036 // get object keys :  
1037   Object.keys(obj);  
1038  
1039 // get object values :  
1040   Object.values(obj);  
1041  
1042 // DOM: -----[]  
1043 // Purpose:  
1044 /*  
1045    The DOM, or Document Object Model, is a programming interface for web  
1046    documents. It represents the structure of an HTML document in a tree-like  
1047    structure, where each node represents a part of the document, such as  
1048    elements, attributes, and text.  
1049 */  
1050  
1051 // Document:  
1052 /*  
1053    The document is the root node of the DOM tree. It represents the entire  
1054    HTML document and provides methods and properties for interacting with its  
1055    contents.  
1056 */  
1057  
1058 // Object:  
1059 /*  
1060    Each element in an HTML document, such as <div>, <p>, <h1>, etc., is  
1061    represented as an object in the DOM. These objects can be accessed and  
1062    manipulated using JavaScript.  
1063 */  
1064  
1065 // Model:  
1066 /*  
1067    The DOM provides a structured representation of the document, allowing  
1068    developers to programmatically access, modify, and manipulate the content,  
1069    structure, and style of the document.  
1070 */  
1071  
1072 //  
1073 /*  
1074    The DOM is essential for dynamic web development because it allows  
1075    JavaScript to interact with and modify the HTML content of a webpage in  
1076    real-time. It enables tasks such as adding or removing elements, changing  
1077    element attributes and styles, and responding to user events like clicks  
1078    and keypresses.  
1079 */  
1080  
1081 /*
```

```
1078     In summary, the DOM is a programming interface that allows JavaScript to
1079     interact with and manipulate the structure and content of HTML documents,
1080     making web pages dynamic and interactive.
1081 */
1082 // get page title :
1083     console.log(document.title);
1084 // get page body :
1085     console.log(document.body);
1086 // get page forms collection :
1087     console.log(document.forms);
1088 // get value of an input that has a name='one' from form 1 in the page
1089     console.log(document.forms[0].one.value);
1090 // get page links(a) collection
1091     console.log(document.links);
1092 // get href of link number 1 in the page :
1093     console.log(document.links[0].href);
1094 // get page imgs
1095     console.log(document.images);
1096 // change the src of the first img in the page :
1097     document.images[0].src="newSource";
1098
1099 // selectors: -----
1100 // get element by id :
1101     var element=document.getElementById('idName');
1102
1103 // get elements by tag name :
1104     var elements = document.getElementsByTagName("TagName");
1105
1106 // get elements by class name :
1107     var elements=document.getElementsByClassName("className");
1108
1109 // get element by id or class or tag:
1110     var element = document.querySelector("[TagName] or [#idName] or [.className]");
1111
1112 // get elements by id or class or tag:
1113     var elements = document.querySelectorAll("[TagName] or [#idName] or [.className]");
1114
1115 // inner text as html to an element (you can insert html element also ) : -----
1116     -----
1117     element.innerHTML="innerHTMLWanted";
1118
1119 //inner text to an element : -----
1120     element.innerText="innerHTMLWanted";
1121
1122 // get content of element : -----
1123     console.log(element.innerHTML);
1124
1125 // get input value : -----
1126     var input=document.querySelector("input");
1127     console.log(input.value);
1128
1129 // set attribute : -----
1130     element.setAttribute("attributeName","value");
1131     -- or
1132     element.attribute=value;
1133
1134 // example :
```

```
1133 let h1=document.querySelector("h2");
1134 let input=document.querySelector("input");
1135 let content=h1.innerHTML="hello world";
1136 input.setAttribute("value",content);
1137
1138 // get value of an attribute : -----
1139 element.getAttribute("attributeName");
1140 --or
1141 element.attributeName;
1142
1143 // get all attributes : -----
1144 element.attributes;
1145
1146 // check if element has an attribute : -----
1147 element.hasAttribute('attributeName');
1148
1149 // check if element has any attribute : -----
1150 element.hasAttributes();
1151
1152 // remove attribute : -----
1153 element.removeAttribute("attributeName");
1154
1155 // style : -----
1156 // set styling :
1157 element.style.styleName="value";
1158 -- or
1159 element.setProperty("propertyName", 'value', "priority");
1160
1161 // remove style :
1162 element.style.styleName="unset";
1163 -- or
1164 element.removeProperty("propertyName");
1165
1166 //example 1:--[
1167   element.style.color="red";
1168 //example 2:--{
1169   element.style.backgroundColor="black";
1170
1171 // cssText :
1172   element.style.cssText=
1173   `styling...
1174
1175 //example : --[
1176   h1=document.querySelector("h1");
1177
1178   h1.style.cssText=
1179   `background-color: red;
1180   color : green;
1181   padding :20px;
1182   text-align : center;
1183
1184   `
1185 // edit styling of style css file : -- [
1186   document.styleSheets[ccsFileName].rules[StylingBlockNumber].style.removeProperty("property")
1187 /*
```

```
1188     document.styleSheets : object contains all css files
1189     rules : contains all css styling block of a specific file :
1190     style access to the of a specific block :
1191 */
1192 //example : --[
1193     document.styleSheets[0]
1194     // html : <link rel="stylesheet" href="style.css">
1195
1196     // css :
1197     ...
1198     // rules[0]
1199     h1{
1200         background-color: red;
1201         //Property
1202         color : green;
1203         padding :20px;
1204         text-align : center;
1205
1206     }
1207     // rules[1]
1208     h2{
1209         //Property
1210         color: green;
1211     }
1212     ...
1213     console.log(document.styleSheets[0].rules[0].style.removeProperty("color"))
1214     console.log(document.styleSheets[0].rules[1].style.removeProperty("color"))
1215
1216 // classes : -----
1217 // add class List :
1218 element.classList='cls1 cls2 clsN';
1219 // --or:
1220     element.classList.add("cls1","cls2","clsN");
1221
1222 //replace a class :
1223     element.classList.replace("oldName", "newName")
1224
1225 // remove a class :
1226     element.classList.remove("className");
1227
1228 // get the number of classes of an element :
1229     element.classList.length;
1230
1231 // check if a class is exist
1232     element.classList.contains("className");
1233
1234 // get className using order :
1235     element.classList.item(index);
1236
1237 // add & remove in the same time , if class exist it will remove otherwise it will add it
1238 element.classList.toggle("className");
1239
1240 //create element : -----
1241 // create element :
1242     let newElement=document.createElement('tagName');
1243 // create textContent:
```

```
1244     var textContent=document.createTextNode("TextWanted");
1245 // create attribute :
1246     var attribute=document.createAttribute("name");
1247 // change value of attribute element :
1248     attribute.value='value';
1249 // create comment :
1250     var comment=document.createComment("comment");
1251
1252 // append to an html element :
1253     element.appendChild(element1);
1254 // prepend child to an html element :
1255     element.prepend(element1)
1256
1257 // add an element before an element
1258     parentElement.insertBefore(newElement,referenceElement);
1259 // ref : the element you wanna insert before :
1260 // or:
1261     referenceElement.before(element);
1262
1263 // insert element after element :
1264     referenceElement.after(element);
1265 // insert element in a position :
1266     refElement.insertAdjacentElement("position", newElement);
1267 // insert html content :
1268     refElement.insertAdjacentHTML("position", HtmlContent);
1269 // insert Text content :
1270     refElement.insertAdjacentText("position", HtmlContent);
1271 // position _____:
1272     "beforebegin" //Inserts immediately before the target element.
1273     "afterbegin"//Inserts as the first child of the target element.
1274     "beforeend"//Inserts as the last child of the target element.
1275     "afterend" //Inserts immediately after the target element.
1276
1277 // clone child :
1278     copy=element.cloneNode([true]); // pass true if want to clone also the content
1279
1280 // Notes -- [
1281 /*
1282     content : (include spaces && comments)
1283     Element : (exclude spaces && comments)
1284 */
1285
1286 // know the parent of an element :
1287     element.parentNode;
1288     -- or
1289     element.parentElement;
1290
1291 // get the first content in a div :
1292     element.firstChild;
1293 // get the first element in a div
1294 element.firstElementChild;
1295
1296 // get the last content in a div :
1297     element.lastChild
1298 // get the last content in a div :
1299     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
1360     its descendants
1361     let titleClone = title.cloneNode(true); // Clone title and its descendants
1362     let detailsClone = details.cloneNode(true); // Clone details and its descendants
1363
1364     // Set unique content for title element
1365     titleClone.textContent = `Product ${i}`;
1366
1367     // Append cloned elements to productContainer clone
1368     productClone.classList.add(`products${i}`);
1369     productClone.appendChild(titleClone);
1370     productClone.appendChild(detailsClone);
1371
1372     // Append productContainer clone to document body
1373     document.body.appendChild(productClone);
1374 }
1375
1376 document.head.innerHTML += `
1377 >
1378 .product{
1379     <style>
1380         .product{
1381             padding: 10px;
1382             padding-left: 15px;
1383             border-radius: 12px;
1384             box-shadow: 1px 5px 10px rgba(0,0,0 ,0.3);
1385             margin-bottom: 20px;
1386             width: 80%;
1387             margin: 30px auto;
1388             border-left: 5px solid #262643;
1389         }
1390     </style>
1391 };
1392 // assignment video 100 : -----
1393 // js ---[ :
1394 let body = document.querySelector("body");
1395
1396 //start upDown animation -----
1397 function getAnimation() {
1398     let parentDiv = document.createElement("div");
1399     parentDiv.classList.add("parent");
1400     let h3Anim = document.createElement("h3");
1401     h3Anim.textContent = "Build By Ayoub Majid";
1402     let pointsDiv = document.createElement("div");
1403     pointsDiv.classList.add("points");
1404
1405     for (let i = 1; i <= 3; i++) {
1406         let tempPointDiv = document.createElement("div");
1407         tempPointDiv.classList.add(`point${i}`);
1408         pointsDiv.appendChild(tempPointDiv);
1409     }
1410

```

```

1411     // append elements to parent div :
1412     parentDiv.appendChild(h3Anim);
1413     parentDiv.appendChild(pointsDiv);
1414
1415     return parentDiv;
1416 }
1417
1418 function pushAnimationToDom() {
1419     let startPointsComment = document.createComment("Start upDown animation ");
1420     let endPointsComment = document.createComment("End upDown animation ");
1421
1422     body.appendChild(startPointsComment);
1423     body.appendChild(getAnimation());
1424     body.appendChild(endPointsComment);
1425 }
1426
1427 pushAnimationToDom();
1428
1429 var parent = document.querySelector(".parent");
1430
1431 function delay(milliseconds) {
1432     return new Promise((resolve) => setTimeout(resolve, milliseconds));
1433 }
1434 function scrollToTop() {
1435     document.documentElement.scrollTop = 0; // For modern browsers
1436     document.body.scrollTop = 0; // For older browsers
1437 }
1438
1439 async function upDownAnim() {
1440     scrollToTop();
1441     parent.style.display = "flex";
1442     for (let i = 1; i >= 0; i -= 0.01) {
1443         parent.style.opacity = i;
1444         await delay(24);
1445     }
1446     parent.style.display = "none";
1447 }
1448 upDownAnim();
1449
1450 // start header ----- :
1451 function getHeader() {
1452     let header = document.createElement("header");
1453     let nav = document.createElement("nav");
1454     nav.classList.add("navBar");
1455     let h1Header = document.createElement("h1");
1456     h1Header.classList.add("title");
1457     h1Header.textContent = "ELZERO";
1458
1459     let ulHeader = document.createElement("ul");
1460     let arrLiList = [];
1461     let arrLiContent = ["Home", "About", "Services", "Contact"];
1462     for (let i = 0; i < 4; i++) {
1463         arrLiList.push(document.createElement("li"));
1464         arrLiList[i].textContent = arrLiContent[i];
1465     }
1466     arrLiList.forEach((li) => {

```

```
1467     ulHeader.appendChild(li);
1468 });
1469 nav.appendChild(h1Header);
1470 nav.appendChild(ulHeader);
1472
1473 header.appendChild(nav);
1474
1475 return header.cloneNode(true);
1476 }
1477 function pushHeader() {
1478     let startComment = document.createComment("start header");
1479     let endHeader = document.createComment("end header");
1480     body.appendChild(startComment);
1481     body.appendChild(getHeader());
1482     body.appendChild(endHeader);
1483 }
1484 pushHeader();
1485 // start main content ----- :
1486
1487 function getMainContent() {
1488     let main = document.createElement("main");
1489     let products = document.createElement("div");
1490     products.classList.add("products");
1491
1492     let product = document.createElement("div");
1493
1494     product.classList.add("product");
1495     let h1Product = document.createElement("h1");
1496     h1Product.classList.add("ProductNumber");
1497     let pProduct = document.createElement("p");
1498     pProduct.textContent = "Product";
1499     product.appendChild(h1Product);
1500     product.appendChild(pProduct);
1501
1502     for (let i = 1; i <= 50; i++) {
1503         let tempProduct = product.cloneNode(true);
1504         tempProduct.firstChild.textContent = i;
1505         products.appendChild(tempProduct);
1506     }
1507     return products;
1508 }
1509 function pushProductsToDom() {
1510     let startMainContentComment = document.createComment("start main Content");
1511     let endMainContentComment = document.createComment("end main Content");
1512     body.appendChild(startMainContentComment);
1513     body.appendChild(getMainContent());
1514     body.appendChild(endMainContentComment);
1515 }
1516 pushProductsToDom();
1517
1518 // start footer ----- :
1519
1520 function getFooter() {
1521     let footerDom = document.createElement("footer");
1522 }
```

```
1523 let h2Footer = document.createElement("h2");
1524 h2Footer.innerHTML = "Copyright &copy 2023";
1525
1526 footerDom.appendChild(h2Footer);
1527
1528 return footerDom;
1529 }
1530
1531 function pushFooterToDom() {
1532     let startFooterComment = document.createComment("start Footer");
1533     let endFooterComment = document.createComment("end Footer");
1534
1535     body.appendChild(startFooterComment);
1536     body.append(getFooter());
1537     body.appendChild(endFooterComment);
1538 }
1539 pushFooterToDom();
1540
1541 // css : --[
1542 `````
1543     /* start default parameters */
1544     *
1545     padding: 0;
1546     margin: 0;
1547     box-sizing: border-box;
1548 }
1549
1550 ul, li {
1551     list-style: none;
1552 }
1553
1554 a {
1555     text-decoration: none;
1556 }
1557
1558 input, button {
1559     outline: none;
1560 }
1561
1562 /* end default parameters */
1563
1564 /* start components */
1565 .container {
1566     padding-left: 10px;
1567     padding-right: 10px;
1568     margin-left: auto;
1569     margin-right: auto;
1570 }
1571
1572 /* small */
1573 @media (min-width:768px) {
1574     .container {
1575         width: 750px;
1576     }
1577 }
```

```
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: #eeeeef24;
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;
```

```
1635     display: flex;
1636     transition: opacity 0.3s;
1637     flex-wrap: wrap;
1638     z-index: 99999;
1639 }
1640
1641 .points {
1642     display: flex;
1643     justify-content: center;
1644     gap: 30px;
1645 }
1646
1647 .parent h3 {
1648     color: #2c4755;
1649     margin-left: 20px;
1650     font-size: 2.3em;
1651     font-weight: 800;
1652 }
1653
1654
1655 @media (max-width:768px) {
1656
1657     .parent h3 {
1658         font-size: 2em;
1659         text-align: center;
1660     }
1661 }
1662
1663 .points div {
1664     width: 50px;
1665     height: 50px;
1666     border-radius: 50%;
1667     background-color: var(--primaryColor);
1668     animation-name: upDown;
1669     animation-duration: 2s;
1670     animation-iteration-count: 10;
1671     animation-direction: alternate;
1672
1673 }
1674
1675
1676 .parent .point2 {
1677     animation-delay: 0.3s;
1678 }
1679
1680
1681 .parent .point3 {
1682     animation-delay: 0.6s;
1683 }
1684
1685 @keyframes upDown {
1686
1687     to {
1688         opacity: 0.6;
1689         transform: translateY(-40px);
1690     }
1691 }
```

```
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
```

```
1747     ul li:hover::after {
1748         width: 90%;
1749     }
1750
1751     /* end header */
1752     /* start main content */
1753     main {
1754         display: flex;
1755         flex-direction: column;
1756         justify-content: space-between;
1757         margin-bottom: 60px;
1758
1759     }
1760
1761     /* start products */
1762     .products {
1763         margin-top: 50px;
1764         padding: 20px;
1765         display: grid;
1766         grid-template-columns: repeat(auto-fit, minmax(240px, 1fr));
1767         row-gap: 50px;
1768         column-gap: 30px;
1769         background-color: var(--primaryBg);
1770
1771     }
1772
1773     .products .product {
1774         background-color: var(--secondaryBg);
1775         text-align: center;
1776         padding: 26.5px 20px;
1777         border-radius: 13px;
1778         font-size: 1.2rem;
1779         box-shadow: 0px 10px 10px rgba(3, 36, 13, 0.045);
1780         transition: 0.4s;
1781     }
1782
1783     .products .product:hover {
1784
1785         transform: translateY(-4px);
1786     }
1787
1788     .product .ProductNumber {
1789         margin-bottom: 16px;
1790         color: var(--primaryColor);
1791     }
1792
1793     .product p {
1794         color: #3e3e3e;
1795     }
1796
1797     /* end products */
1798     /* start footer */
1799     footer {
1800         position: fixed;
1801         width: 100%;
1802         bottom: 0;
```

```

1803         background-color: rgb(31, 114, 23);
1804         padding: 3px;
1805         text-align: center;
1806         color: white;
1807     }
1808
1809     footer h2 {
1810         padding: 7px;
1811         text-shadow: 1px 2px 10px rgba(0, 0, 0, 0.503);
1812     }
1813
1814     /* end footer */
1815
1816     /* end main content */
1817     ```
1818 // html : --[
1819     ```
1820     <!DOCTYPE html>
1821     <html lang="en">
1822         <head>
1823             <meta charset="UTF-8" />
1824             <meta name="viewport" content="width=device-width, initial-scale=1.0" />
1825             <title>Document</title>
1826
1827             <!-- google fonts -->
1828             <link rel="preconnect" href="https://fonts.googleapis.com" />
1829             <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
1830             <link href="https://fonts.googleapis.com/css2?family=Rubik:ital,wght@0,300..900;1,300..900&display=swap" rel="stylesheet" />
1831
1832             <link rel="stylesheet" href="style.css" />
1833         </head>
1834
1835         <body>
1836             <script src="script.js"></script>
1837
1838             <style></style>
1839         </body>
1840     </html>
1841
1842     ```
1843 // Events : -----
1844 // type of events :
1845     // 1. **Mouse Events:** *
1846     /*
1847         ✓ - `click` - Occurs when the mouse is clicked.
1848         ✓ - `dblclick` - Occurs when the mouse is double-clicked.
1849         ✓ - `mousedown` - Occurs when a mouse button is pressed down.
1850         ✓ - `mouseup` - Occurs when a mouse button is released.
1851         ✓ - `mousemove` - Occurs when the mouse pointer moves.
1852         ✓ - `mouseover` - Occurs when the mouse pointer enters an element.
1853         ✓ - `mouseout` - Occurs when the mouse pointer leaves an element.
1854     */
1855     // 2. **Keyboard Events:** *
1856     /*
1857         ✓ - `keydown` - Occurs when a key is pressed down.

```

```

1858     ✓ - `keyup` - Occurs when a key is released.
1859     ✓ - `keypress` - Occurs when a key is pressed.
1860   */
1861 // 3. **Form Events:**
1862 /*
1863     ✓ - `submit` - Occurs when a form is submitted.
1864     ✓ - `reset` - Occurs when a form is reset.
1865     ✓ - `change` - Occurs when the value of an input element changes.
1866     ✓ - `input` - Occurs when the content of an input element changes.
1867   */
1868 // 4. **Window Events:**
1869 /*
1870     ✓ - `Load` - Occurs when a webpage has finished Loading.
1871     ✓ - `unload` - Occurs when a user leaves a page.
1872     ✓ - `resize` - Occurs when the browser window is resized.
1873     ✓ - `scroll` - Occurs when the user scrolls in the window.
1874   */
1875 // 5. **Document Events:**
1876 /*
1877     ✓ - `DOMContentLoaded` - Occurs when the HTML document has been completely loaded
and
1878           parsed without waiting for stylesheets, images, and subframes to finish Loading.
1879     ✓ - `readystatechange` - Occurs when the readyState property of the document
changes.
1880   */
1881 // 6. **Focus Events:**
1882 /*
1883     ✓ - `focus` - Occurs when an element gains focus.
1884     ✓ - `blur` - Occurs when an element loses focus.
1885   */
1886 // 7. **Touch Events:**
1887 /*
1888     ✓ - `touchstart` - Occurs when a touch is initiated on a touch-enabled device.
1889     ✓ - `touchmove` - Occurs when a touch point is moved along the touch surface.
1890     ✓ - `touchend` - Occurs when a touch point is removed from the touch surface.
1891   */
1892
1893 // add event :
1894   element.onEvent=function(){
1895     //code
1896   }
1897 -- or
1898   element.addEventListener("event",function(){
1899     //code
1900
1901   });
1902 -- or
1903   //html code :
1904   /*
1905     <tagName onEvent="functionName() or JavaScriptCode" >
1906   */
1907   //js code :
1908   function FunctionName(){
1909     // code
1910   }
1911 //event object -----
1912 /*

```

1913     The event object in JavaScript is a special object that contains information about an  
1914     event when it occurs, such as a user  
1915     action (e.g., a mouse click, a key press, or a form submission). The event object  
1916     is automatically passed as an argument  
1917     to the event handler function when an event occurs, and it provides details about the  
1918     event, such as the type of the event,  
1919     the target element, and additional information specific to the event type.  
1920     \*/  
1921     // ⚡1. \*\*`event.type`\*\*:  
1922         // - Returns a string representing the type of the event ("click", "keydown", ...).  
1923  
1924     // ⚡2. \*\*`event.target`\*\*:  
1925         // - Returns the DOM element that triggered the event.  
1926  
1927     // ⚡3. \*\*`event.currentTarget`\*\*:  
1928         // - Returns the current DOM element that is handling the event.  
1929     //example : --[  
1930         

1931             

1932                 Click me!  
1933

  
1934

  
1935         document.getElementById('outerDiv').addEventListener('click', function(event) {  
1936             // currentTarget refers to the element that the event listener is attached to  
1937             console.log('Current Target: ' + event.currentTarget.id);  
1938  
1939             // target refers to the actual element that triggered the event  
1940             console.log('Target: ' + event.target.id);  
1941         });  
1942  
1943     // ⚡4. \*\*`event.preventDefault()`\*\*:  
1944     /\*  
1945         - Prevents the default behavior associated with the event. For example,  
1946         preventing the default action of a form submission or a hyperlink click.  
1947     \*/  
1948     // ⚡5. \*\*`event.stopPropagation()`\*\*:  
1949     /\*  
1950         - Stops the event from propagating up or down the DOM hierarchy. It prevents the event  
1951         from reaching other event listeners on ancestor or descendant elements.  
1952     \*/  
1953     //example  
1954         

1955             

1956                 Click me!  
1957

  
1958

  
1959         document.getElementById('innerDiv').addEventListener('click', function(event) {  
1960             // This prevents the event from reaching the outer div  
1961             event.stopPropagation();  
1962  
1963             alert('Inner div clicked!');  
1964         });  
1965  
1966         document.getElementById('outerDiv').addEventListener('click', function() {  
1967             alert('Outer div clicked!');  
1968         });

```

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';

```

```

2025     newElement.classList.add('dynamic-element');
2026     document.getElementById('container').appendChild(newElement);
2027 });
2028
2029 // Add event listener to a parent element that will contain dynamically added content
2030 document.getElementById('container').addEventListener('click', function(event) {
2031     // Check if the clicked element matches a specific selector
2032     if (event.target.matches('.dynamic-element')) {
2033         // Your event handling logic here
2034         console.log('Dynamic element clicked');
2035     }
2036 });
2037 //event.target.matches :
2038 /*
2039     you can use any valid CSS selector string with the matches() method to check if an
2040     element matches the specified selector.
2041     This includes class selectors (.className), ID selectors (#idName), attribute selectors
2042     ([attributeName="value"]),
2043     tag selectors (tagName), pseudo-class selectors (:pseudo-class), and more.
2044 */
2045 //Example : -----
2046 let myP=document.querySelector("h1");
2047
2048 myP.addEventListener('click' ,function () {
2049     let newP = myP.cloneNode(true);
2050     newP.className = "clone";
2051     document.body.appendChild(newP);
2052 });
2053
2054 document.addEventListener("click", function (e) {
2055     if (e.target.className === "clone") {
2056         e.target.addEventListener('click',function(){
2057             console.log("clone Element ");
2058         })
2059     }
2060 });
2061 // remove event listener : -----
2062 element.removeEventListener('click', callbackFunction)
2063 // example : --[
2064     // Define the event listener function
2065     function clickHandler() {
2066         let newP = myP.cloneNode(true);
2067         newP.className = "clone";
2068         document.body.appendChild(newP);
2069     }
2070
2071     // Add the event listener to myP
2072     myP.addEventListener('click', clickHandler);
2073
2074     // Later, when you want to remove the event listener
2075     myP.removeEventListener('click', clickHandler);
2076
2077 // BOM [Browser Object Modal] : -----
2078 // show a pop up window with a message  need No response only ok Available
2079 ```;

```

```
2080     [window.]alert("message")
2081     ``
2082 //show pop up window with a message : need response and return a boolean ok=true &
cancel=false
2083     var status=confirm("message :");
2084
2085 // show pop up window with a message : collect Data
2086     var data=prompt('msg','defaultMsg');
2087
2088 // *setTimeout -----
2089 /*
2090     this function pause the execution of the code
2091 until we get a specific time (arrow function ) :
2092 */
2093 setTimeout(() => {
2094     // code
2095 }, timeout); // time with ms
2096 //clear interval :
2097 clearInterval(intervalName);
2098
2099 // BOM :[browser object modal]-----[]
2100
2101 // window.location object : -----
2102
2103 // set / get the link of the page with saving session history :
2104 /*
2105     if you pass the short link it will just change the page name
2106     oldPat/newHrefLink
2107 */
2108     window.location.href;
2109
2110 //set / get the hostname (ip address of the server )
2111     window.location.hostname;
2112 //set /get the host= [hostname + portNumber] (ip of server + port)
2113     window.location.host;
2114
2115 // get /set the protocol of the page (http / https)
2116     location.protocol;
2117
2118 // reload the page :
2119     location.reload();
2120
2121 // replace the link of the page with removing the old Link from
2122 //session history :
2123     location.replace("newLink")
2124
2125 // set the link of the page with saving session history :
2126     location.assign("newLink");
2127 // window functions : -----
2128 // open a window :
2129 window.open("url","_self | _blank","window style(put values without px)",
wannaSaveThePreviousInHistory);
2130 // example :
2131     setInterval(() => {
2132
2133         window.open('https://google.com','','width:200px;height:200px');
2134     }, 200);
```

```
2135 // close the window
2136     window.close();
2137
2138 // history object  -----
2139     // show history link :
2140         history.length;
2141
2142     // get back to previous history page :
2143         history.back();
2144
2145     // get forward to next history page :
2146         history.forward();
2147
2148     // get back/forward base on the position :
2149     /*
2150         position 0 reload the page :
2151     */
2152         history.go(position);
2153
2154 // print the page :
2155 window.print();
2156
2157 // focus at a window :
2158 window.focus();
2159 //example :
2160 let newWindow=document.open('www.google.com','_blank','width :300;height 500; ')
2161 newWindow.focus();
2162
2163 //scroll to a position :
2164 scrollToTop(xValue,yValue);
2165 scroll(xValue,yValue);
2166
2167 // scroll to a position with base on the same position
2168 //in the next scroll
2169 scrollBy(xValue,yValue);
2170
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.pageYOffset // 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 // check if get the end of the page :
2192     if (scrollTop + windowHeight >= totalHeight);
2193
2194 // get url params :
2195 // method 1 :
2196     // Function to get URL parameters
2197     function getUrlParams() {
2198         // Create a new URL object with the current URL
2199         const url = new URL(window.location.href);
2200
2201         // Return the searchParams object, which contains the parameters
2202         return Object.fromEntries(url.searchParams.entries());
2203     }
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);
```

```
2247
2248 // dataset : -----
2249 /*
2250   it's a way to set some information about an html an element
2251   to access getting using js dataset
2252 */
2253 // example :
2254 // html
2255 ```
2256   <button data-color="red" data-background="green">stop</button>
2257 ```
2258 // js
2259   let btn=document.querySelector('button');
2260   console.log(btn.dataset);
2261
2262
2263 // session storage :
2264 ```
2265   Local Storage and Session Storage are both mechanisms provided by web browsers to store
key-value
2266   pairs locally within the user's browser. However, they have some differences in terms of
scope,
2267   lifetime, and usage:
2268 ```
2269 // 1. **Scope:***
2270 /*
2271   - **Local Storage:** Data stored in local storage persists even after the browser is
closed
2272     and reopened. It is scoped to the origin (i.e., the combination of protocol, host,
2273     and port of the website). This means that data stored in local storage for one
2274     website is accessible to that same website even if the user navigates away and
2275     comes back later.
2276   - **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
2277     is opened. Once the tab or window is closed, the data is cleared and no longer
accessible.
2278 */
2279 // 2. **Lifetime:***
2280 /*
2281   - **Local Storage:** Data stored in local storage persists indefinitely until explicitly
removed
2282     by the web application or cleared by the user through browser settings.
2283   - **Session Storage:** Data stored in session storage persists only for the duration of
the page
2284     session. If the user navigates to another page within the same website or closes the
tab/window,
2285     the session storage data is cleared.
2286 */
2287 // 3. **Usage:***
2288 /*
2289   - **Local Storage:** Local storage is often used for storing data that needs to be
preserved across browser
2290     sessions, such as user preferences, cached data, or application state.
2291   - **Session Storage:** Session storage is useful for storing temporary data that is only
relevant
2292     for the current browsing session, such as data needed for multi-step processes or
data that should not
2293     persist beyond the current session.
```

```

2295 */
2296 /*
2297     In summary, Local storage provides persistent storage across browser sessions and is
2298     scoped to the origin,
2299     while session storage provides temporary storage for the duration of a page session and
2300     is scoped to the tab
2301     or window. The choice between them depends on the specific requirements of the web
2302     application and the
2303     desired behavior of the stored data.
2304 */
2305 /**
2306     info :
2307     -new tab = new session
2308     -duplicate Tab = copy session
2309     -new tab with same url = new session
2310 */
2311 // store primitive data type in session storage :
2312     sessionStorage.setItem('key',value);
2313     sessionStorage.key=value;
2314     sessionStorage['key']=value;
2315 // store no primitive data type in session storage :
2316     sessionStorage.setItem('key',JSON.stringify(value));
2317 //get primitive data type in session storage :
2318     result=sessionStorage.setItem('key');
2319 // get no primitive data type in session storage :
2320     result=JSON.parse(sessionStorage.setItem('key'));
2321
2322 //remove item from session storage :
2323     localStorage.removeItem('key');
2324     delete sessionStorage.key
2325
2326 //clear session storage :
2327     sessionStorage.clear();
2328
2329 // get key by index ;
2330     sessionStorage.key(index);
2331
2332 // Destructuring Assignment :-----[ ]
2333 /*
2334     Destructuring assignment is a feature introduced in ES6 (ECMAScript 2015) that allows you
2335     to
2336     extract values from arrays or properties from objects into distinct variables.
2337     It provides a concise and
2338     convenient way to unpack values from complex data structures like arrays and objects.
2339 */
2340 // Array Destructuring: -----[ ]
2341 /*
2342     In array destructuring, you can extract elements from an array and assign them to
2343     variables
2344     in a single line. The syntax uses square brackets [] on the left side of the assignment.
2345 */
2346     // Array with values
2347     numbers = [1, 2, 3, 4, 5];

```

```
2347 // Destructuring assignment
2348 const [first, second, , fourth] = numbers;
2349
2350 console.log(first); // Output: 1
2351 console.log(second); // Output: 2
2352 console.log(fourth); // Output: 4
2353
2354 // In object destructuring, you can extract specific properties
2355 // from an object and assign them to variables.
2356 // The syntax uses curly braces `{}` on the left side of the assignment.
2357
2358 // object Destructuring : -----
2359 /*
2360     Destructuring assignment is a powerful feature that makes working
2361     with arrays and objects more concise and expressive,
2362     especially when dealing with complex data structures.
2363     It is widely used in modern JavaScript applications.
2364 */
2365 var { firstName, age } = person1;
2366 console.log(firstName); // Output: "John"
2367 console.log(age); // Output: 30
2368
2369 // You can also provide default values for variables in case the property
2370 // or array element does not exist:
2371 const Numbers = [1, 2];
2372 var [A, b, c = 3] = Numbers;
2373
2374 console.log(A); // Output: 1
2375 console.log(b); // Output: 2
2376 console.log(c); // Output: 3 (default value)
2377
2378 // You can also rename variables during destructuring using a colon `:`:
2379 const person2 = {
2380     firstName: "John",
2381     lastName: "Doe",
2382     age: 30,
2383 };
2384
2385 var { firstName: fName, lastName: lName } = person2;
2386
2387 console.log(fName); // Output: "John"
2388 console.log(lName); // Output: "Doe"
2389 // you can use variables already declared :
2390 const user={
2391     firstName :"ayoub",
2392     lastName:"majid",
2393     age :20 ,
2394     skills : {
2395         html :"html",
2396         css :"css"
2397     }
2398 }
2399     var firstName,lastName;
2400
2401 ({firstName,lastName}=user);
2402 // function destructuring :
```

```

2403 user = {
2404     firstName: "ayoub",
2405     lastName: "majid",
2406     age: 20,
2407     skills: {
2408         html: "html",
2409         css: "css",
2410     },
2411 };
2412
2413 function showUser({ firstName, lastName, skills: { css } } = user) {
2414     console.log(`your name is ${firstName}`);
2415     console.log(`your lastName is ${lastName}`);
2416     console.log(`your skills is ${css}`);
2417 }
2418
2419 showUser(user);
2420
2421 // mix content destructuring :
2422 user = {
2423     firstName: "ayoub",
2424     lastName: "majid",
2425     age: 20,
2426     skills: ["html", "css"],
2427     address:{ 
2428         morocco :"Rabat",
2429         egypt:"cairo"
2430     }
2431 };
2432 const {firstName,lastName,age,skills:[html,css],address:{morocco,egypt}}=user;
2433
2434 // assignment : --[
2435 let chosen =3;
2436 let myFriends=[ 
2437     {title:"nice",age:30,available:true,skills:["html", "css"]},
2438     {title:"amine",age:23,available:false,skills:["python", "django"]},
2439     {title:"ayoub",age:20,available:true,skills:["node js", "express"]}
2440 ]
2441
2442 if(chosen==1){
2443     var [{title,age,available,skills:[,skill2]},,]=myFriends;
2444 }
2445
2446 if(chosen==2){
2447     var [,,{title,age,available,skills:[,skill2]}]=myFriends;
2448 }
2449
2450 if(chosen==3){
2451     var [,,{title,age,available,skills:[,skill2]}]=myFriends;
2452 }
2453     console.log(`title ${title}`);
2454     console.log(`age ${age}`);
2455     console.log(`${available ? "available" : "Not available"}`);
2456     console.log(`skills 2 ${skill2}`);
2457
2458 // set data type : -----

```

```
2459 // create new set :  
2460 const set = new Set([1, 2, 3]);  
2461 // or  
2462 let data=[1,2,3];  
2463 set= new Set(data);  
2464 // or  
2465 set= new Set().add(1).add(2).add(3);  
2466  
2467 // add new value :  
2468 set.add(5);  
2469 // you can add duplicate values :  
2470 set.add(5);  
2471  
2472 // list a set :  
2473 for (element of set) {  
2474     console.log(element);  
2475 }  
2476  
2477 // check if value exist :  
2478 console.log(set.has(4));  
2479  
2480 // delete a value of set : return [true : value exist /false : not exist]  
2481 set.delete(3);  
2482  
2483 // get the Length of the set :  
2484 console.log("the size of the set is : ", set.size);  
2485  
2486 // clear a set :  
2487 set.clear();  
2488  
2489 //List using foreach :  
2490 set.forEach((element) => {  
2491     console.log("the value: ", element);  
2492 });  
2493  
2494  
2495 // weakset dataType : works just with objects -----[]  
2496 // create a weakset :  
2497 var weak= new WeakSet([{ A: 1, B: 2 }]);  
2498 // add new value :  
2499 set.add(5);  
2500  
2501 // check if value exist :  
2502 console.log(set.has(4));  
2503  
2504 // delete a value of set : return [true : value exist /false : not exist]  
2505     set.delete(3);  
2506 // - Set vs WeakSet --[  
2507 /*  
2508     "  
2509         The WeakSet is weak,  
2510         meaning references to objects in a WeakSet are held weakly.  
2511         If no other references to an object stored in the WeakSet exist,  
2512         those objects can be garbage collected.  
2513     "  
2514     --
```

```

2515     Set      => Can Store Any Data Values
2516     WeakSet => Collection Of Objects Only
2517     --
2518     Set      => Have Size Property
2519     WeakSet => Does Not Have Size Property
2520     --
2521     Set      => Have Keys, Values, Entries
2522     WeakSet => Does Not Have clear, Keys, Values And Entries
2523     --
2524     Set      => Can Use forEach
2525     WeakSet => Cannot Use forEach
2526
2527     Usage: Store objects and removes them once they become inaccessible
2528 */
2529 // map data type : -----
2530
2531 // Create a new Map
2532 let myMap = new Map();
2533 // or
2534 myMap= new Map([
2535     [key,value],
2536     [key,value]
2537 ])
2538
2539 // Add key-value pairs to the Map
2540 myMap.set('name', 'John');
2541 myMap.set(1, 'One');
2542 myMap.set(true, 'True');
2543
2544 // Get values from the Map
2545 console.log(myMap.get('name')); // Output: John
2546 console.log(myMap.get(1)); // Output: One
2547 console.log(myMap.get(true)); // Output: True
2548
2549 // Iterate over key-value pairs using forEach
2550 myMap.forEach((value, key) => {
2551     console.log(` ${key} : ${value}`);
2552 });
2553 // Output:
2554 // name : John
2555 // 1 : One
2556 // true : True
2557
2558 // Get the number of key-value pairs
2559 console.log(myMap.size); // Output: 3
2560
2561 // Check if a key exists
2562 console.log(myMap.has('name')); // Output: true
2563
2564 // Delete a key-value pair
2565 myMap.delete(1);
2566 console.log(myMap.size); // Output: 2
2567
2568 // real using -- [
2569 /*
2570     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);
```

```
2624
2625 // notes : --[
2626 /*
2627     The behavior you're observing is expected when using WeakMap in JavaScript.
2628
2629     WeakMap holds weak references to its keys, which means that if there are no other
2630     references
2631     to a key object apart from the one stored in the WeakMap, the key object may still be
2632     garbage
2633     collected. However, the garbage collection process is non-deterministic,
2634     meaning that it's not guaranteed to happen immediately after an object becomes
2635     unreachable.
2636
2637     In your code, even though you set obj to null, the garbage collector may not immediately
2638     reclaim the
2639     memory occupied by obj and its associated key-value pair in the WeakMap. Garbage
2640     collection is
2641     typically performed by the JavaScript engine asynchronously, and the exact timing may
2642     vary
2643     depending on various factors such as memory pressure, garbage collection algorithms, and
2644     optimizations performed by the engine.
2645
2646     To test if the object is actually removed from the WeakMap, you would need to explicitly
2647     trigger
2648     garbage collection, but this isn't possible directly in JavaScript. However, you could
2649     try forcing
2650     garbage collection in some environments by using platform-specific tools or by running
2651     the script
2652     in a controlled environment that allows such actions.
2653
2654
2655     In summary, while WeakMap helps in managing weak references and allows objects to be
2656     garbage collected
2657     when they are no longer reachable from other parts of the program, the timing of garbage
2658     collection is
2659     not deterministic and may vary depending on the JavaScript engine and runtime
2660     environment.
2661 */
2662
2663 // convert from element to array : -----
2664 // Array.from(element,(item)={
2665 //     //code
2666 // })
2667
2668 // convert from string to array
2669 // Array.from("ayoub");
2670
2671 // get the sum of a string :
2672 // Array.from("12345",(item)=> +item + +item);
2673
2674 // convert from set to Array :
2675 // myArr=[1,1,1,2,2,4];
2676 // set= new Set(myArr);
2677
2678 // console.log(Array.from(set));
2679 // you can also use the spread operator ...
2680 // myArr=[1,1,1,2,2,4];
2681 // set= new Set(myArr);
2682 // console.log([...set]);
2683
```

```

2671 // copy a values from an array to another position
2672 myArr.copyWithin(targetIndex,startCopyIndex=0,EndCopyIndex=arr.length) // end not included
2673
2674 // example :
2675 let myArray = [10, 20, 30, 40, 50, "A", "B"];
2676
2677 myArray.copyWithin(3); // [10, 20, 30, 10, 20, 30, 40]
2678
2679 myArray.copyWithin(4, 6); // [10, 20, 30, 40, "B", "A", "B"]
2680
2681 myArray.copyWithin(4, -1); // [10, 20, 30, 40, "B", "A", "B"]
2682
2683 myArray.copyWithin(1, -2); // [10, "A", "B", 40, 50, "A", "B"]
2684
2685 myArray.copyWithin(1, -2, -1); // [10, "A", 30, 40, 50, "A", "B"]
2686
2687 // check if any value at least match the condition in an array : [true /false]
2688 arr.some(CallbackFunc(Element, Index, Array),ValueToCompareWith)
2689 /*
2690   - Array.some(CallbackFunc(Element, Index, Array))
2691   --- CallbackFunc => Function To Run On Every Element On The Given Array
2692   ----- Element => The Current Element To Process
2693   ----- Index => Index Of Current Element
2694   ----- Array => The Current Array Working With
2695   ----- ValueToCompareWith = it's represented by this inside the CallbackFunc()
2696   --
2697   Using
2698   - Check if Element Exists In Array
2699   - Check If Number In Range
2700 */
2701 // example : --[
2702   var arr = [1, 2, 3, 4, 5, 6];
2703
2704 // method 1 :
2705   var status = arr.some(function (element) {
2706     return element > this;
2707   }, 5);
2708
2709 // method 2 :
2710   var status = arr.some(element=>element >5);
2711
2712 // check if an element is exist in an array :
2713   var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2714   function checkValues(arr, val) {
2715     return arr.some(function (e) {
2716       return e === val;
2717     });
2718   }
2719
2720   console.log(checkValues(nums, 20));
2721   console.log(checkValues(nums, 5));
2722
2723 // check if at least one element in the array is in a range :
2724   var range = {
2725     min: 10,
2726     max: 20,

```

```

2727     };
2728
2729     var checkNumberInRange = nums.some(function (e) {
2730         return e >= this.min && e <= this.max;
2731     }, range);
2732
2733     console.log(checkNumberInRange);
2734
2735
2736
2737 // check if all values match the condition in an array : [true /false]
2738 arr.every(CallbackFunc(Element, Index, Array),ValueToCompareWith)
2739 /*
2740     - Array.some(CallbackFunc(Element, Index, Array))
2741     --- CallbackFunc => Function To Run On Every Element On The Given Array
2742     ----- Element => The Current Element To Process
2743     ----- Index => Index Of Current Element
2744     ----- Array => The Current Array Working With
2745     ----- ValueToCompareWith = it's represented by this inside the CallbackFunc()
2746     --
2747     Using
2748     - Check If all values match a condition
2749 */
2750 // example : --[
2751     var arr = [1, 2, 3, 4, 5, 6];
2752
2753 // method 1 :
2754     var status = arr.every(function (element) {
2755         return element > this;
2756     }, 5);
2757
2758 // method 2 :
2759     var status = arr.every(element=>element >5);
2760
2761
2762 // check if all values in the array is in a range :
2763 var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2764     var range = {
2765         min: 10,
2766         max: 20,
2767     };
2768
2769     var checkNumberInRange = nums.every(function (e) {
2770         return e >= this.min && e <= this.max;
2771     }, range);
2772
2773     console.log(checkNumberInRange);
2774
2775 // find an element using value : -----
2776 arr.find(CallbackFunc(Element, Index, Array),ValueToCompareWith)
2777 /*
2778     - Array.find(CallbackFunc(Element, Index, Array))
2779     --- CallbackFunc => Function To Run On Every Element On The Given Array
2780     ----- Element => The Current Element To Process
2781     ----- Index => Index Of Current Element
2782     ----- Array => The Current Array Working With

```

```

2783     ----- ValueToCompareWith = it's represented by this inside the CallbackFunc()
2784     --
2785     Using
2786     - get value of the first ArrayElement that match the value passed in parameter
2787
2788     return undefined if not / element if exist
2789 */
2790 // example : --[
2791     var arr = [1, 2, 3, 4, 5, 6];
2792
2793     // method 1 :
2794     var status = arr.find(function (element) {
2795         return element > this;
2796     }, 5);
2797
2798     // method 2 :
2799     var status = arr.find(element=>element >5);
2800
2801     // get the value of the first element equal value :
2802     var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2803     function getValue(arr, val) {
2804         return arr.find(function (e) {
2805             return e === val;
2806         });
2807     }
2808
2809     console.log(getValue(nums, 20));
2810     console.log(getValue(nums, 5));
2811
2812     // get the value of the first element in the range :
2813     var range = {
2814         min: 10,
2815         max: 20,
2816     };
2817
2818     var valueInRange = nums.find(function (e) {
2819         return e >= this.min && e <= this.max;
2820     }, range);
2821
2822     console.log(valueInRange);
2823
2824 // find an element using index : -----
2825 arr.findIndex(CallbackFunc(Element, Index, Array),ValueToCompareWith)
2826 /*
2827     - Array.find(CallbackFunc(Element, Index, Array))
2828     --- CallbackFunc => Function To Run On Every Element On The Given Array
2829     ----- Element => The Current Element To Process
2830     ----- Index => Index Of Current Element
2831     ----- Array => The Current Array Working With
2832     ----- ValueToCompareWith = it's represented by this inside the CallbackFunc()
2833     --
2834     Using
2835     - Check if Element Exists In Array
2836
2837     return -1 if not / index of element if exist
2838 */

```

```

2839 // example : --[
2840   var arr = [1, 2, 3, 4, 5, 6];
2841
2842   // method 1 :
2843   var status = arr.findIndex(function (element) {
2844     return element > this;
2845   }, 5);
2846
2847   // method 2 :
2848   var status = arr.find(element=>element >5);
2849
2850   // get the index of an element in an array :
2851   var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2852   function GetIndex(arr, val) {
2853     return arr.findIndex(function (e) {
2854       return e === val;
2855     });
2856   }
2857
2858   console.log(GetIndex(nums, 20));
2859   console.log(GetIndex(nums, 5));
2860
2861   // get the index of the first element in the range :
2862   var range = {
2863     min: 10,
2864     max: 20,
2865   };
2866
2867   var elementIndexInRange = nums.findIndex(function (e) {
2868     return e >= this.min && e <= this.max;
2869   }, range);
2870
2871   console.log(elementIndexInRange);
2872
2873 // Spread Operator : -----
2874 /*
2875   Spread Operator => ...Iterable
2876   "Allow Iterable To Expand In Place"
2877 */
2878
2879 // Spread With String => Expand String
2880 console.log("Osama");
2881 console.log(..."Osama"); // split to characters : O s a m a
2882 console.log([... "Osama"]); // split to array of characters : ['O', 's', 'a', 'm', 'a']
2883 console.log({... "Osama"}); // split to object: {0: 'O', 1: 's', 2: 'a', 3: 'm', 4: 'a'}
2884
2885 // Concatenate Arrays
2886 let myArray1 = [1, 2, 3];
2887 let myArray2 = [4, 5, 6];
2888 let allArrays = [...myArray1, ...myArray2]; //: [1, 2, 3, 4, 5, 6]
2889 console.log(allArrays);
2890
2891 // Copy Array
2892 let copiedArray = [...myArray1];
2893 console.log(copiedArray); //: [1, 2, 3]
2894

```

```

2895 // Push Inside Array
2896 let allFriends = ["Osama", "Ahmed", "Sayed"];
2897 let thisYearFriends = ["Sameh", "Mahmoud"];
2898 allFriends.push(...thisYearFriends);
2899 console.log(allFriends); //: ['Osama', 'Ahmed', 'Sayed', 'Sameh', 'Mahmoud']
2900
2901 // Use With Math Object
2902 let myNums = [10, 20, -100, 100, 1000, 500];
2903 console.log(Math.max(...myNums)); //: 1000
2904
2905 // Spread With Objects => Merge Objects
2906 let objOne = {
2907   a: 1,
2908   b: 2,
2909 };
2910 let objTwo = {
2911   c: 3,
2912   d: 4,
2913 };
2914 console.log({ ...objOne, ...objTwo, e: 5 }); //: {a: 3, b: 2, d: 4, e: 5}
2915
2916 // assignment : -----
2917 let n1 = [10, 30, 10, 20];
2918 let n2 = [30, 20, 10];
2919 // Result : 200 * 1.05 = 210
2920 console.log(
2921   ( n1[new Set([...n1]).size] * Math.min(...n2) ) // Result : 200
2922   //Result : 0.05 * 21 = 1.05
2923   * ( // Result = 5 /100 = 0.05
2924     (
2925       // Result : 20 / 4 = 5
2926       ( n1[new Set([...n1]).size] // n1[3]=20
2927         // Result : 2*2=4
2928         /
2929           ( Math.max( ... n1.join("").split("") ).slice((new Set([...n2])
2930 .size)) ) //2
2931           * ( Math.max( ... n1.join("").split("") ).slice((new Set([...n2])
2932 .size)) ) // 2
2933             )
2934             )
2935             // Result : 21
2936             *
2937               ( new Set([...n1]).size ) // Result : 3
2938               * ( [...n1,...n2].length ) // Result : 7
2939             )
2940           )
2941         );
2942
2943
2944 // explain
2945 /*
2946   1- ( n1[new Set([...n1]).size] * Math.min(...n2) ) = 20 * 10 = 200 ://R1
2947
2948   2-( n1[new Set([...n1]).size] ) = n1[3]=20 ://R2
2949

```

```

2950      3-( Math.max( ... n1.join("").split("").slice((new Set([...n2]).size)) ) ) = 2
2951      :///*R3
2952          4-( R2 / ( ( R2 ) * ( R2 ) ) ) = 20/(2*2)=5 :///*R4
2953          5-( Math.min(...n1) ) = 10 :///*R5
2954          6-( R4/(R5 * R5) ) = 5/100 = 0.05 :///*R6
2955          7-( new Set([...n1,]).size ) = 3 :///*R7
2956          8-( new Set([...n1, ...n2]).size ) = 7 :///*R8
2957          9-( R6 * ( R7 * R8 ) ) = 0.05 * (7*3) = 1.05 :///*R9
2958          /*result :
2959             console.log(R1*R9) =210
2960
2961         */
2962         // or :
2963         // Result : 200 * 1.05 = 210
2964         console.log(Math.max(...n2) * [...n1, ...n2].length);
2965
2966     // regular Expressions : -----
2967     /*
2968         Regular expressions, often abbreviated as regex or regexp, are sequences of characters
2969         that define
2970             a search pattern. They are widely used in computer science and programming for tasks
2971             such as
2972                 pattern matching, string manipulation, and data validation. Regex allows for the
2973             efficient
2974                 and flexible processing of text by enabling the identification and extraction of
2975             specific
2976                 patterns within strings.
2977     */
2978
2979     // 1. Literal Characters:
2980     /*
2981         - Literal characters match themselves within a string. For example,
2982             the regex pattern 'hello' matches the word 'hello' in a string.
2983     */
2984
2985     // 2. Metacharacters:
2986     /*
2987         - Metacharacters have special meanings in regular expressions and are used to define
2988             patterns
2989     */
2990
2991     /*
2992         - '.' (dot): Matches any single character except newline.
2993         - '^' (caret): Matches the start of a line.
2994         - '$' (dollar): Matches the end of a line.
2995         - '*' (asterisk): Matches zero or more occurrences of the preceding character.
2996         - '+' (plus): Matches one or more occurrences of the preceding character.
2997         - '?' (question mark): Matches zero or one occurrence of the preceding character.
2998         - '|' (pipe): Acts as an OR operator, allowing alternative matches.
2999         - '[' and ']' (square brackets): Define a character class, matching any character
within the brackets.
            - '\' (backslash): Escapes metacharacters to match them literally.
2994     */
2995     // 3. Character Classes:
2996     /*
2997         - Character classes represent groups of characters and allow for more concise pattern
definitions.
2998     */
2999     /*

```

```

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         /*

```

```

3055      - Replacing or modifying text based on predefined patterns or rules.
3056      - Formatting text for display or storage according to specific conventions.
3057      */
3058 // test your pattern [true: any matches word /false]
3059   var reg=/exp	option;
3060   console.log(reg.test())
3061
3062 //test your pattern at a string [matches words]
3063   var reg=/exp	option;
3064   var str1='text'
3065   /*
3066     1-matches a string against a regular expression pattern
3067     2-returns an array with the matches
3068     3-returns null if no math is found
3069   */
3070 console.log(str1.match(str));
3071
3072 // replace the matches words : -- []
3073   // replace the first match word :
3074   console.log(str1.replace(reg,replacedValue));
3075
3076   // replace all matches word :
3077   console.log(str1.replaceAll(reg,replacedValue));
3078
3079 //examples :
3080
3081 // example 1 : --[
3082 /*
3083   v1/v2 : or
3084   [0-9] : range of value from 0 to 9
3085   [^0-8] : values not in this range
3086   [^0-9] : get values not no a number
3087 */
3088 var myString1='AaBbcdefG123!234%^&*'
3089
3090 let SmallAToZ=/[a-z]/g;
3091 let NotSmallAToZ=/[^a-z]/g;
3092 let capitalAToZ=/[A-Z]/g;
3093 let NotCapitalAToZ=/[^A-Z]/g;
3094 let acd=/[acd]/g;
3095 let notAcd=/[^acd]/g;
3096
3097 let letters=/[aA-zZ]/g;
3098 let notLetters=/[^aA-zZ]/g;
3099 let special =/[aA0-zZ9]/g;
3100 console.log(myString1.match(SmallAToZ));
3101 console.log(myString1.match(NotSmallAToZ));
3102 console.log(myString1.match(capitalAToZ));
3103 console.log(myString1.match(NotCapitalAToZ));
3104 console.log(myString1.match(acd));
3105 console.log(myString1.match(notAcd));
3106 console.log(myString1.match(letters));
3107 console.log(myString1.match(notLetters));
3108 console.log(myString1.match(special));
3109 // example 2 :--[
3110 /*

```

```

3111 --character classes :
3112   . : matches any character , except newLine or other line terminators
3113   \w : matches word characters. [a-z,A-Z,0-9 and underscore]
3114   \W : matches Non word characters
3115   \d : matches digits from 0-9
3116   \s : matches whitespace character
3117 */
3118 let email ='O@@@g...com o@g.com o@g.net o@g.com o-g.com o@s.org 1@1.com'
3119 let dot=/./g;
3120 let word=/\w/g;
3121 let noWord=/\W/g;
3122 let valid=/\w@\w.(com|net)/g;
3123 console.log(email.match(dot));
3124 console.log(email.match(word));
3125 console.log(email.match(noWord));
3126 console.log(email.match(valid));
3127
3128 // example 3 : --[
3129 /*
3130   \b : matches at the beginning or end of word.
3131   \B : matches not at the beginning/end of a word.
3132 */
3133 var names="Sayed 1Spam 2Spam 2Spam spam4 spam5 Osama Ahmed Aspamo";
3134 let re=/(\bspam|spam\b)/ig;
3135
3136 console.log(names.match(re));
3137 console.log(re.test("spam"))
3138
3139 // example 4 : --[
3140 // Quantifiers :
3141 /*
3142   n+ ... => one or more
3143   n* ... => zero or more
3144   n? .. => zero or n
3145   n{x} : Number of n
3146   n{x,y} : range from x to y
3147   n{x,} : at Least x
3148   $ : en with something
3149   ^ : start with something
3150   ?= : followed by something
3151   ?! : not followed by something
3152 */
3153 let mails='o@nn.sa osama@gmail.com elzero@gmail.net osama@mail.ru';
3154 var numbers1='0110 10 150 05120 0560 350 00'
3155 let reg=/\w+\.\w+/ig;
3156 console.log(mails.match(reg));
3157 let resNumbers=/0\d+0/ig;
3158 console.log(numbers1.match(resNumbers));
3159
3160
3161 let urls='https://google.com http://www.website.net web.com https';
3162 let regUrl=/^(https?:\/\/)?(www.)?\w+\.\w+/ig;
3163 console.log(urls.match(regUrl));
3164
3165 let serials='s100s s3000s s50000s s950000s';
3166 let reg1=/s\d{4,}s/g;

```

```
3167 console.log(serializers.match(reg1));
3168
3169 let str2='we love Programming';
3170 let names='1OsamaZ 2AhmedZ 3Mohammed 4MoustafaZ 5GamalZ';
3171 console.log(/^[a-z]{2}\s/gi.test(str2));
3172 let reg2=/\d\w{5}z/ig;
3173 reg2=/\d\w{5}(?!z)/gi;
3174 console.log(names.match(reg2))
3175 // OOP [OBJECT ORIENTED PROGRAMMING ] : -----
3176 ``
3177 Object-Oriented Programming (OOP) Overview:
3178
3179 Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects,"
3180 which can contain data in the form of fields (attributes or properties) and code in the form of
3181 procedures (methods or functions). OOP emphasizes organizing code into reusable and modular components,
3182 fostering better code maintenance, scalability, and flexibility. Key concepts of OOP include:
3183
3184 1. Classes and Objects:
3185 - Classes are blueprints or templates for creating objects. They define the attributes (data)
3186 and behaviors (methods) that objects of the class will have.
3187 - Objects are instances of classes, representing specific entities in the program's domain.
3188 They encapsulate data and behavior related to that entity.
3189
3190 2. Encapsulation:
3191 - Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on
3192 the data into a single unit (class or object).
3193 - It promotes information hiding, where the internal details of an object are hidden from the
3194 outside world, and only a well-defined interface is exposed for interacting with the object.
3195
3196 3. Inheritance:
3197 - Inheritance allows a class (subclass or derived class) to inherit properties and
3198 behaviors from another class (superclass or base class).
3199 - Subclasses can extend the functionality of their superclass, adding new features or
3200 overriding existing ones, while maintaining the common interface.
3201
3202 4. Polymorphism:
3203 - Polymorphism enables objects to be treated as instances of their superclass,
3204 allowing for more generic code that can operate on objects of different types.
3205 - It includes concepts like method overriding (redefining a method in a subclass)
3206 and method overloading (defining multiple methods with the same name but different parameters).
3207
3208 5. Abstraction:
3209 - Abstraction focuses on modeling the essential aspects of real-world
3210 entities while hiding unnecessary complexity.
3211 - It allows developers to create simplified models of complex systems,
3212 emphasizing what an object does rather than how it does it.
3213
3214 Benefits of Object-Oriented Programming:
```

```
3215
3216    1. Modularity: OOP promotes code modularity by encapsulating related functionality into
3217        classes
3218            and objects, making code easier to understand, maintain, and reuse.
3219
3220    2. Reusability: Objects and classes can be reused in different parts of the program
3221        or in different programs, reducing redundancy and promoting code
3222        efficiency.
3223
3224    3. Scalability: OOP provides a scalable approach to software development, allowing for
3225        easy extension and modification of existing code without affecting other
3226        parts of the system.
3227
3228    4. Flexibility: OOP enables developers to build flexible and adaptable systems that can
3229        evolve
3230            over time to meet changing requirements and business needs.
3231
3232
3233    5. Understandability: OOP emphasizes a clear and intuitive design approach, making it
3234        easier for
3235            developers to understand and reason about the structure and behavior
3236        of the code.
3237
3238
3239 // general syntax : --[
3240     class className {
3241         constructor(args) {
3242             this.att1 = value1;
3243             // ...
3244         }
3245         f1 = function () {};
3246         f2 = () => {
3247             // this is window in this scope
3248         };
3249
3250         f3() {}
3251     }
3252
3253 // create new instance from a class : --[
3254     let instance1= new className(args);
3255
3256
3257 // check if an object is an instance of a specific class :--[
3258     console.log(obj instanceof className);
3259
3260 // example : --[
3261
3262 // old method
3263     function User(id, username, salary) {
3264         this.id = id;
3265         this.username = username;
3266         this.salary = salary + 1000;
3267     }
3268
3269
3270     var user1 = new User(1, "amine", 500);
3271     var user2 = new User(2, "kamal", 2500);
3272     var user3 = new User(3, "nice", 3500);
3273     console.log(user1, "\n", user2, "\n", user3);
3274
3275 // ES6 :
3276     class User {
3277         constructor(id, username, salary) {
```

```

3267         this.id = id;
3268         this.username = username;
3269         this.salary = salary + 1000;
3270     }
3271 }
3272 let user1 = new User(1, "amine", 500);
3273 let user2 = new User(2, "kamal", 2500);
3274 let user3 = new User(3, "nice", 3500);
3275 console.log(user1, "\n", user2, "\n", user3);
3276
3277 // static members :
3278 /*
3279     Static members in JavaScript classes are properties or methods that belong
3280     to the class itself rather than to instances of the class. They are shared
3281     among all instances of the class and can be accessed directly from the class
3282     without the need to create an instance. Static members are declared using
3283     the static keyword within a class definition.
3284 */
3285 //static members :
3286 class Circle {
3287     //Static properties are shared across all instances of the class and are accessed using
3288     //the class name itself.
3289     static PI = 3.14;
3290     static radius = 1;
3291
3292     static getArea() {
3293         return Circle.PI * Circle.radius * Circle.radius;
3294     }
3295     console.log(Circle.PI); // Output: 3.14
3296     console.log(Circle.getArea()); // Output: 3.14
3297 //static methods :
3298 class MathUtils {
3299     //Static methods are functions that belong to the class itself and do not operate on
3300     //instances of the class.
3301     static add(x, y) {
3302         return x + y;
3303     }
3304     static subtract(x, y) {
3305         return x - y;
3306     }
3307     console.log(MathUtils.add(5, 3)); // Output: 8
3308     console.log(MathUtils.subtract(5, 3)); // Output: 2
3309 // Static Factory Methods:
3310 class Point {
3311     //Static factory methods are static methods used to create instances of the class.
3312     constructor(x, y) {
3313         this.x = x;
3314         this.y = y;
3315     }
3316
3317     static createOrigin() {
3318         return new Point(0, 0);
3319     }
3320 }
3321

```

```
3322     const origin = Point.createOrigin();
3323     console.log(origin); // Output: Point { x: 0, y: 0 }
3324
3325 // private members :
3326 class className{
3327     #PrivateProperty
3328     constructor(args){
3329         this.#PrivateProperty=value;
3330     }
3331     #privateMethod(){
3332         // code
3333
3334     }
3335 }
3336 //example :
3337 class Counter {
3338     #count = 0;
3339
3340     #increment() {
3341         this.#count++;
3342     }
3343
3344     #getCount() {
3345         return this.#count;
3346     }
3347
3348     publicMethod() {
3349         this.#increment();
3350         return this.#getCount();
3351     }
3352 }
3353
3354 counter = new Counter();
3355 console.log(counter.publicMethod()); // Output: 1
3356 console.log(counter.#count); // Error: Private field '#count' is not defined
3357 console.log(counter.#increment()); // Error: Private method '#increment' is not defined
3358 // inheritance :
3359 class baseClass{
3360     constructor(arg1){
3361         this.p1=arg1;
3362         /// ...
3363     }
3364     getPass(){
3365         this.p1 *10;
3366     }
3367
3368 };
3369 class derivedClass extends baseClass{
3370     constructor(arg1,arg2){
3371         // access constructor of base class :
3372         super(arg1);
3373         this.p2=arg2;
3374
3375 }
3376
3377     getDerivedClass(){
```

```

3378 // access function of baseClass :
3379 return super.getPass() + this.arg2;
3380 }
3381 }
3382 // example :
3383 function Person(name, age) {
3384     this.name = name;
3385     this.age = age;
3386 }
3387 class Student extends Person {
3388     constructor(name, age, grade) {
3389         super(name, age);
3390         this.grade = grade;
3391     }
3392
3393     study() {
3394         console.log(` ${this.name} is studying.`);
3395     }
3396 }
3397
3398 const student2 = new Student('Emma', 18, 'A');
3399 student2.greet(); // Output: Hello, my name is Emma and I am 18 years old.
3400 student2.study(); // Output: Emma is studying.
3401
3402 // Prototypes :
3403 /*
3404 In JavaScript, each object has an associated prototype, which serves as a fallback
3405 mechanism for properties and methods that are not found directly on the object.
3406 Prototypes allow for property and method inheritance, where objects can inherit
3407 properties and methods from their prototype chain.
3408 */
3409 //example
3410 class cPerson {
3411     constructor(name, age) {
3412         this.name = name;
3413         this.age = age;
3414     }
3415
3416     syaHello() {
3417         return `Hello ${this.name}`;
3418     }
3419 }
3420
3421 cPerson.prototype.nice = function () {
3422     return `nice to meet you ${this.name}`;
3423 };
3424 console.log(cPerson.prototype);
3425 const person3 = new cPerson("Alice", 25);
3426 console.log(person3.nice());
3427
3428 // object meta data descriptor : -----
3429 /*
3430 -writable : change his value
3431 -enumerable : not appears in loops (Log the object , iterate object key [for in ])
3432 -configurable[cannot delete or reconfigure ]
3433 -value : default value of the property

```

```
3434 */
3435 // syntax :
3436 Object.defineProperty(object, 'propertyName', {
3437   writable : value,
3438   enumerable : value,
3439   configurable : value,
3440   value: value,
3441 })
3442
3443 // example :
3444 const myObject = {
3445   a: 1,
3446   b: 2,
3447   c: 3,
3448 };
3449
3450 Object.defineProperty(myObject, "c", {
3451   writable: false,
3452   enumerable: false,
3453   configurable: false,
3454   value: 500,
3455 });
3456
3457 myObject.c = 100;
3458
3459 console.log(delete myObject.c);
3460 console.log(myObject.c);
3461
3462 // change multiple property metadata:
3463 Object.defineProperties(myObject, {
3464   'a':{
3465     writable: value,
3466     enumerable: value,
3467     configurable: value,
3468     value:value,
3469   },
3470   'b':{
3471     writable: value,
3472     enumerable: value,
3473     configurable: value,
3474     value:value,
3475   },
3476   'c':{
3477     writable: value,
3478     enumerable: value,
3479     configurable: value,
3480     value:value,
3481   },
3482 });
3483
3484 // get property metadata :
3485 Object.getOwnPropertyDescriptor(myObject, 'propertyName');
3486
3487 // get all properties metadata :
3488 Object.getOwnPropertyDescriptors(myObject);
3489
```

```

3490 // Date And Time : -----
3491 // get current date info :
3492   console.log(new Date());
3493
3494 // get number of milliseconds From 1/1/1970 until now :
3495   let milliseconds = Date.now();
3496
3497 // converting unites :
3498 // number of seconds :
3499   let seconds = milliseconds / 1000;
3500   console.log(`seconds ${seconds}`);
3501
3502 // minutes :
3503   let minutes = seconds / 60;
3504   console.log(`minutes ${minutes}`);
3505
3506 // hours
3507   let hours = minutes / 60;
3508   console.log(`hours ${hours}`);
3509
3510 // days :
3511   let days = hours / 24;
3512   console.log(`days ${days}`);
3513
3514 // months :
3515   let months = days / 30;
3516   console.log(`months ${months}`);
3517
3518 // years :
3519   let years = months / 12;
3520   console.log(`years ${years}`);
3521
3522 // get difference between two date :
3523   var dateNow = new Date();
3524   console.log(dateNow);
3525   let birthDay = new Date("12/10/2003");
3526
3527   console.log((dateNow - birthDay) / 1000 / 60 / 60 / 24 / 365);
3528
3529 // Date functions :--[
3530 // get functions :
3531   let date= new Date("12/10/2003");
3532   console.log(date.getTime()); //get number of milliseconds From 1/1/1970 until Date:
3533   console.log(date.getDate()); // get day number in month
3534   console.log(date.getFullYear()); // get year
3535   console.log(date.getMonth()); // get month [0,11]
3536   console.log(date.getDay()); // get day index order in the week [From sunday To saturday]
3537     console.log(date.getHours()); // get current hours (24 format)
3538     console.log(date.getMinutes()); // get current minutes
3539     console.log(date.getSeconds()); // get current seconds
3540
3541 // set functions :
3542 /*
3543   - setTime(Milliseconds)
3544   - setDate() => Day Of The Month [Negative And Positive]

```

```

3545     - setFullYear(year, month => Optional [0-11], day => Optional [1-31])
3546     - setMonth(Month [0-11], Day => Optional [1-31]) [Negative And Positive]
3547     - setHours(Hours [0-23], Minutes => Optional [0-59], Seconds => Optional [0-59], MS =>
3548     - Optional [0-999])
3549     - setMinutes(Minutes [0-59], Seconds => Optional [0-59], MS => Optional [0-999])
3550     - setSeconds(Seconds => [0-59], MS => Optional [0-999])
3551 */
3552 // example :
3553   let dateNow = new Date();
3554   dateNow.setTime(0);
3555   console.log(dateNow); // Thu Jan 01 1970 00:00:00 GMT+0000 (GMT)
3556
3557   dateNow.setTime(10000);
3558   console.log(dateNow); // Thu Jan 01 1970 00:00:10 GMT+0000 (GMT)
3559
3560   dateNow.setDate(35);
3561   console.log(dateNow); // Wed Feb 04 1970 00:00:10 GMT+0000 (GMT)
3562
3563   dateNow.setFullYear(2020, 13);
3564   console.log(dateNow); // Wed Feb 04 1970 00:00:10 GMT+0000 (GMT)
3565
3566   dateNow.setMonth(15);
3567   console.log(dateNow); // Mon Apr 04 2022 00:00:10 GMT+0000 (GMT)
3568
3569 // Date Formatting : --[
3570 /*
3571   new Date(timestamp)
3572   new Date(Date String)
3573   new Date(Numeric Values)
3574
3575   Format
3576   - "Oct 25 1982"
3577   - "10/25/1982"
3578   - "1982-10-25" => ISO International Standard
3579   - "1982 10"
3580   - "1982"
3581   - "82"
3582   - 1982, 9, 25, 2, 10, 0 => year monthIndex day hours minutes seconds
3583   - 1982, 9, 25
3584   - "1982-10-25T06:10:00Z"
3585
3586   Date.parse("String") // Read Date From A String
3587 */
3588
3589   console.log(Date.parse("Oct 25 1982"));
3590
3591   let date1 = new Date(0);
3592   console.log(date1);
3593
3594   let date2 = new Date(404344800000);
3595   console.log(date2);
3596
3597   let date3 = new Date("10-25-1982");
3598   console.log(date3);
3599
3600   let date4 = new Date("1982-10-25");

```

```

3600 console.log(date4);
3601
3602 let date5 = new Date("1982-10");
3603 console.log(date5);
3604
3605 let date6 = new Date("82");
3606 console.log(date6);
3607
3608 let date7 = new Date(1982, 9, 25, 2, 10, 0);
3609 console.log(date7);
3610
3611 let date8 = new Date(1982, 9, 25);
3612 console.log(date8);
3613
3614 let date9 = new Date("1982-10-25T06:10:00Z");
3615 console.log(date9);
3616
3617 //Track Operations Time : ---[
3618 // Start Time
3619 let start = new Date();
3620
3621 // Operation
3622 for (let i = 0; i < 100; i++) {
3623 document.write(`<div>${i}</div>`);
3624 }
3625 // Time End
3626 let end = new Date();
3627
3628 // Operation Duration
3629 let duration = end - start;
3630
3631 console.log(`${duration}ms`);
3632
3633 // Generators : -----
3634 /*
3635 - Generator Function Run Its Code When Required.
3636 - Generator Function Return Special Object [Generator Object]
3637 - Generators Are Iterable
3638 */
3639 /*
3640
3641 Generators in JavaScript are special functions that can pause and resume their
execution.
3642 They are defined using the function* syntax and utilize the yield keyword to pause the
function's
3643 execution and return a value to the caller. Generators offer a powerful mechanism for
creating
3644 iterable sequences and asynchronous programming. Here's an overview of their usage:
3645 */
3646
3647 //1-Declaring a Generator Function: -- [
3648 /*
3649 Generator functions are declared using the function* syntax.
3650 Within the generator function, you can use the yield keyword
3651 to pause the execution and return a value to the caller.
3652 */
3653 //Example :

```

```
3654     function* generatorFunction() {
3655         yield 1;
3656         yield 2;
3657         yield 3;
3658     }
3659 // 2- Iterating Over Generator Values: -- [
3660 /*
3661     Generators produce iterable sequences that can be iterated over using loops,
3662     spread syntax, or built-in iterator methods like next().
3663 */
3664 //Example :
3665     const gen = generatorFunction();
3666     console.log(gen.next().value); // Output: 1
3667     console.log(gen.next().value); // Output: 2
3668     console.log(gen.next().value); // Output: 3
3669
3670
3671
3672 //3-Using for...of Loop with Generators: --[
3673 /*
3674     Generators can be used with the for...of Loop to iterate over all
3675     values produced by the generator.
3676 */
3677 //Example :
3678     for (const value of generatorFunction()) {
3679         console.log(value);
3680     }
3681     // Output:
3682     // 1
3683     // 2
3684     // 3
3685
3686 //4-Passing Values to Generators: --[
3687 /*
3688     In JavaScript, the next() method of an iterator returned by a generator function allows
3689     you to either
3690         retrieve the next value from the generator or to pass a value into the generator
3691         function. It returns
3692             an object with two properties: value and done.
3693
3694     When you call next() without passing a parameter, the generator function continues its
3695     execution until it
3696         encounters a yield statement, where it pauses and returns the yielded value as value.
3697         If the generator function completes execution without encountering a yield statement,
3698         the value
3699             property will be undefined, and the done property will be true.
3700         If you pass a parameter to next(value), the value passed as a parameter will be returned
3701         by
3702             the yield expression inside the generator function, and the generator will continue its
3703             execution
3704                 from where it was paused.
3705                 If an error occurs within the generator function, you can throw an exception using the
3706                 throw()
3707                     method of the iterator. This will cause the generator function to throw an error at the
3708                     point
3709                         where it's currently paused.
3710                         */
3711 //Example :
```

```

3704 | function* myGenerator() {
3705 |   try {
3706 |     const x = yield 1; // Pause execution and return 1 when next() is called
3707 |     console.log('Received:', x);
3708 |     const y = yield 2; // Pause execution and return 2 when next() is called
3709 |     console.log('Received:', y);
3710 |     yield 3; // Pause execution and return 3 when next() is called
3711 |   } catch (error) {
3712 |     console.log('Error caught:', error);
3713 |   }
3714 |
3715 |
3716 |   const iterator = myGenerator();
3717 |   console.log(iterator.next()); // Output: { value: 1, done: false }
3718 |   console.log(iterator.next('Hello')); // Output: Received: Hello, { value: 2, done: false }
3719 |   console.log(iterator.throw(new Error('Something went wrong'))); // Output: Error caught:
3720 |   Error: Something went wrong
3721 |
3722 //5- Return(): --[
3723 /*
3724   When return is called, the generator function exits early, and subsequent calls to
3725   next()
3726   will return an object with done set to true.
3727   The provided value (if any) is returned as the value property of the object returned by
3728   the next() method.
3729   If return is called with no argument, it's equivalent to return undefined.
3730 */
3731 //Example :
3732   function* myGenerator() {
3733     yield 1;
3734     yield 2;
3735     yield 3;
3736   }
3737
3738   iterator = myGenerator();
3739   console.log(iterator.next()); // Output: { value: 1, done: false }
3740   console.log(iterator.return('Stopped')); // Output: { value: 'Stopped', done: true }
3741   console.log(iterator.next()); // Output: { value: undefined, done: true }
3742
3743 //6- Asynchronous Generators: --[
3744 /*
3745   Generators can be used in asynchronous programming to produce asynchronous iterable
3746   sequences using asynchronous operations.
3747 */
3748 //Example :
3749   async function* asyncGenerator() {
3750     yield await Promise.resolve(1);
3751     yield await Promise.resolve(2);
3752     yield await Promise.resolve(3);
3753   }
3754
3755   (async () => {
3756     for await (const value of asyncGenerator()) {
3757       console.log(value);
3758     }
3759   })

```

```

3758     })();
3759     // Output:
3760     // 1
3761     // 2
3762     // 3
3763 //7- Delegate Generator Function: --[
3764     //1. `generateNums()`:
3765     /*
3766         - This generator function yields three numeric values: 1, 2, and 3.
3767         - It uses the `yield` keyword to pause execution and return each value one by one.
3768     */
3769     //2. `generateLetters()`:
3770     /*
3771         - This generator function yields three letter values: "A", "B", and "C".
3772         - Similar to `generateNums()`, it uses the `yield` keyword to pause execution and
3773         return each value one by one.
3774     */
3775     //3. `generateAll()`:
3776     /*
3777         - This generator function combines the values from `generateNums()`,
3778         `generateLetters()`, and an array `[4, 5, 6]`.
3779         - It uses the `yield*` syntax to delegate to other generator functions and an
3780         iterable (the array).
3781         - When `yield*` is used, it iterates over the values yielded by the delegated
3782         generator or iterable
3783             and yields each value in the current generator function.
3784             - So, `yield* generateNums();` yields the values from `generateNums()`, `yield*`
3785             `generateLetters();`
3786                 `yields the values from `generateLetters()`, and `yield* [4, 5, 6];` -
3787                 `yields the values from the array `[4, 5, 6]`.
3788     */
3789     iterator = generateAll();
3790
3791     console.log(iterator.next()); // Output: { value: 1, done: false }
3792     console.log(iterator.next()); // Output: { value: 2, done: false }
3793     console.log(iterator.next()); // Output: { value: 3, done: false }
3794     console.log(iterator.next()); // Output: { value: 'A', done: false }
3795     console.log(iterator.next()); // Output: { value: 'B', done: false }
3796     console.log(iterator.next()); // Output: { value: 'C', done: false }
3797     console.log(iterator.next()); // Output: { value: 4, done: false }
3798     console.log(iterator.next()); // Output: { value: 5, done: false }
3799     console.log(iterator.next()); // Output: { value: 6, done: false }
3800     console.log(iterator.next()); // Output: { value: undefined, done: true }
3801
3802 // Advance Real Example : -- [
3803     /*
3804         Let's consider a practical example of using generators for asynchronous programming.
3805         Imagine we have an API that returns data in chunks, and we want to fetch and process
3806         this
3807             data asynchronously. We'll use a generator function to handle the asynchronous
3808         fetching of
3809             data and processing it chunk by chunk.
3810     */
3811     // Simulated asynchronous API function to fetch data in chunks
3812     function fetchDataChunk() {
3813         return new Promise(resolve => {

```

```

3809     // Simulating delay for fetching data
3810     setTimeout(() => {
3811         // Simulating data chunk
3812         const chunk = [1, 2, 3, 4, 5];
3813         resolve(chunk);
3814     }, 1000);
3815 });
3816 }
3817
3818 // Generator function to asynchronously fetch and process data in chunks
3819 async function* processData() {
3820     let pageIndex = 1;
3821     while (pageIndex<=10) {
3822         // Fetch data chunk asynchronously
3823         const dataChunk = await fetchDataChunk();
3824         yield { pageIndex, dataChunk }; // Yield data chunk along with page index
3825         pageIndex++;
3826     }
3827 }
3828
3829 // Function to consume data from the generator asynchronously
3830 async function consumeData() {
3831     for await (const { pageIndex, dataChunk } of processData()) {
3832         console.log(`Page ${pageIndex}:`, dataChunk);
3833
3834         // Process data chunk here (e.g., save to database, perform calculations, etc.)
3835     }
3836 }
3837
3838 // Start consuming data
3839 consumeData();
3840
3841
3842 // Modules Import And Export : -----
3843 // you need add a type='module' to script:src (html file )
3844 ```
3845     <script src="main.js" type="module"></script>
3846     <script src="app.js" type="module"></script>
3847 ```
3848 /*
3849     type="module" attribute, indicating that the JavaScript files (app.js and main.js) are
3850     ES modules.
3851     When you use this attribute, the browser treats the scripts as ECMAScript modules.
3852 */
3853 // export a variable
3854     export let varName=value;
3855
3856 // export a function :
3857     export function functionName(){
3858
3859     };
3860
3861 // export a class :
3862     export class className{
3863

```

```
3864
3865 // export a collection :
3866   export {
3867     varName,
3868     functionName,
3869     className
3870   };
3871
3872 // import var from a file
3873 // alias optional :
3874 import {varName as alias , functionName as alias , className as alias } from 'path/file.js';
3875
3876 // Example : --[
3877   // index.html :
3878   ``
3879     <script src="main.js" type="module"></script>
3880     <script src="app.js" type="module"></script>
3881   ``
3882
3883 // main.js :
3884   let ak = 10;
3885   let arr = [1, 2, 3, 4, 5]; // Fix the array declaration
3886
3887   function saySomething() {
3888     return "nice to meet you";
3889   }
3890
3891   export { ak, arr, saySomething };
3892 // app.js :
3893   import {ak,arr as array,saySomething} from './main.js'
3894   console.log(ak);
3895   console.log(array)
3896   console.log(saySomething());
3897
3898 // import all export function and grouped in a single object :
3899 import * as containerName from 'path/source.js';
3900
3901 // Default export :
3902 /*
3903 you can export just one time
3904 */
3905
3906 // exportDefaultVariable :
3907 ``
3908   export default varName;
3909 ``
3910
3911 // export by default function :
3912 ``
3913   export default functionName;
3914 ``
3915 // export by default a class :
3916 ``
3917   export default className;
3918 ``
```

```
3919 // export by default a collection :
3920 /**
3921   export default {
3922     varName,
3923     functionName,
3924     className
3925   };
3926
3927 /**
3928 /**
3929 // export an anonymous collection :
3930 /**
3931   export default {
3932     // arrow function
3933     x: ()=>{
3934
3935       },
3936       // anonymous function :
3937       function(){
3938
3939         },
3940         // value
3941         v1:5
3942       };
3943 /**
3944 /**
3945 // import with default :
3946 /*
3947   defaultObjectName : set any name you want
3948 */
3949   import defaultObjectName,{varName as alias , functionName as alias , className as alias } from 'path/file.js';
3950
3951 // access default object when you import all :
3952   import * as containerName from 'path/source.js';
3953   containerName.default;
3954
3955 // example : ---[
3956 // html
3957 /**
3958   <script src="main.js" type="module"></script>
3959   <script src="app.js" type="module"></script>
3960 /**
3961
3962 // main.js
3963   a = 10;
3964   let arr = [1, 2, 3, 4, 5]; // Fix the array declaration
3965
3966   function saySomething() {
3967     return "nice to meet you";
3968   }
3969   let k = 10;
3970   export { a, arr, saySomething };
3971
3972   export default {
3973     hi() {
```

```
3974         console.log("hello world");
3975     },
3976     by() {
3977         console.log("good by");
3978     },
3979     v: 5,
3980     function() {
3981         console.log("anonymous function ");
3982     },
3983 };
3984
3985 // app.js :
3986     console.log("\nNamed export : ");
3987     import * as all from "./main.js";
3988     console.log(all.a);
3989     console.log(all.arr);
3990     console.log(all.saySomething());
3991
3992     console.log("\ndefault export : ");
3993     all.default.hi();
3994     all.default.by();
3995     console.log("the value : ", all.default.v);
3996     all.default.function();
3997
3998 // Json : -----
3999 /*
4000     What Is JSON ?
4001     - JavaScript Object Notation
4002     - Format For Sharing Data Between Server And Client
4003     - JSON Derived From JavaScript
4004     - Alternative To XML
4005     - File Extension Is .json
4006
4007     Why JSON ?
4008     - Easy To Use And Read
4009     - Used By Most Programming Languages And Its Frameworks
4010     - You Can Convert JSON Object To JS Object And Vice Versa
4011
4012     JSON vs XML
4013 =====
4014     = Text Based Format      = Markup Language      =
4015     = Lightweight           = Heavier             =
4016     = Does Not Use Tags    = Using Tags          =
4017     = Shorter               = Not Short            =
4018     = Can Use Arrays        = Cannot Use Arrays =
4019     = Not Support Comments = Support Comments   =
4020 =====
4021 */
4022 /*
4023     JSON Syntax
4024     - Data Added Inside Curly Braces { }
4025     - Data Added With Key : Value
4026     - Key Should Be String Wrapped In Double Quotes
4027     - Data Separated By Comma
4028     - Square Brackets [] For Arrays
4029     - Curly Braces {} For Objects
```

```

4030
4031 Available Data Types
4032 - String
4033 - Number
4034 - Object
4035 - Array
4036 - Boolean Values
4037 - null
4038 */
4039 //example : test.json
4040 ``
4041 {
4042     "string": "name",
4043     "number": 100,
4044     "object": {
4045         "EG": "Giza",
4046         "Mr": "Rabat"
4047     },
4048     "array": ["Html", "Css", "Js"],
4049     "boolean": true,
4050     "null": null
4051 }
4052 ``
4053 /*
4054 -- JSON.parse = > convert JSON to Js object
4055 -- JSON.stringify => convert JS object to JSON
4056 */
4057 const ServerObj = '{"username":"ayoub","age":"39"}';
4058 console.log(typeof ServerObj);
4059
4060 const JsObj = JSON.parse(ServerObj);
4061 console.log(JsObj);
4062
4063 JsObj["username"] = "amine";
4064
4065 console.log(JsObj);
4066
4067 let newUpdatedServerData = JSON.stringify(JsObj);
4068
4069 console.log(newUpdatedServerData);
4070 // Asynchronous vs Synchronous Programming : -----
4071 /*
4072 To Understand Ajax, Fetch, Promises
4073
4074 Asynchronous vs Synchronous Programming
4075 - Meaning
4076
4077 Synchronous
4078 - Operations Runs in Sequence
4079 - Each Operation Must Wait For The Previous One To Complete
4080 - Story From Real Life
4081
4082 Asynchronous
4083 - Operations Runs In Parallel
4084 - This Means That An Operation Can Occur while Another One Is Still Being Processed
4085 - 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 /*
```

```

4142 +-----+
4143 |   one()    |
4144 |   two()    |
4145 |   three()   |
4146 +-----+
4147 Each function (one(), two(), three()) is represented as a block.
4148 The blocks are stacked on top of each other, indicating the call stack.
4149 The topmost block represents the function at the top of the call stack (three()),
4150 and the bottommost block represents the function at the bottom of the call stack (one())
4151 .
4152     The arrows indicate the flow of execution, with three() calling two(), which in turn
4153 calls one().
4154 */
4155
4156 /*
4157     To Understand Ajax, Fetch, Promises
4158 Story
4159 - JavaScript Is A Single Threaded Language "ALL Operations Executed in Single Thread"
4160 - Call Stack Track All Calls
4161 - Every Function Is Done Its Popped Out
4162 - When You Call Asynchronous Function It Sent To Browser API
4163 - Asynchronous Function Like Settimeout Start Its Own Thread
4164 - Browser API Act As A Second Thread
4165 - API Finish Waiting And Send Back The Function For Processing
4166 - Browser API Add The Callback To Callback Queue
4167 - Event Loop Wait For Call Stack To Be Empty
4168 - Event Loop Get Callback From Callback Queue And Add It To Call Stack
4169 - Callback Queue Follow FIFO "First In First Out" Rule
4170 */
4171 // Example :
4172     console.log("One");
4173     setTimeout(() => {
4174         console.log("Three");
4175     }, 0);
4176
4177     setTimeout(() => {
4178         console.log("Four");
4179     }, 0);
4180
4181     console.log("Two");
4182     setTimeout(() => {
4183         console.log(myVar);
4184     }, 0);
4185
4186     let myVar = 100;
4187     myVar += 100;
4188 // Ajax : -----
4189 /*
4190
4191 - Ready State => Status Of The Request
4192 [0] Request Not Initialized
4193 [1] Server Connection Established
4194 [2] Request Received
4195 [3] Processing Request
4196 [4] Request Is Finished And Response Is Ready

```

```
4197 - Status
4198 [200] Response Is Successful
4199 [404] Not Found
4200 */
4201
4202 // create an object from XMLHttpRequest class :
4203 let request=new XMLHttpRequest();
4204
4205 // open function : to Prepare a request.: obj.open("type", "url");
4206 request.open("get", "url");
4207
4208 // specify the type of response :
4209 request.responseType="json";
4210
4211 // send the request :
4212 request.send();
4213
4214 // run function when the response arrived :
4215 request.onload=function(){
4216     // code
4217 }
4218
4219 /*
4220     Ajax
4221     - Ready State => Status Of The Request
4222         [0] Request Not Initialized
4223         [1] Server Connection Established
4224         [2] Request Received
4225         [3] Processing Request
4226         [4] Request Is Finished And Response Is Ready
4227     - Status
4228         [200 -300] Response Is Successful
4229         [404] Not Found [url error]
4230         [500 >] server error
4231 */
4232 // return the status of the response :
4233     request.status
4234 // return the readyStat of the request :
4235     request.readyState
4236 // run a function when the stat of request change : -- [
4237     request.onreadystatechange=function(){
4238         console.log("CHANGE ");
4239     };
4240
4241
4242 // add headers params :
4243     xhr.setRequestHeader("HeaderName1", "HeaderValue1");
4244     xhr.setRequestHeader("HeaderName2", "HeaderValue2");
4245 // Example :
4246     // Set the Content-Type header if you're sending JSON data
4247     xhr.setRequestHeader("Content-Type", "application/json");
4248
4249 // Create an object with the data you want to send
4250     var data = {
4251         key1: "value1",
4252         key2: "value2"
```

```
4253     };
4254
4255 // Convert the object to a JSON string
4256 var jsonData = JSON.stringify(data);
4257
4258 // Send the JSON data in the body of the request
4259 xhr.send(jsonData);
4260
4261 // set response type json in postman
4262 /*
4263     headers : =>"Accept:"application/json"
4264 */
4265
4266 // set headers params:
4267 request.setRequestHeader('key','value');
4268
4269 // set response type :
4270     request.setRequestHeader("Accept","application/json");
4271 // set request type :
4272     request.setRequestHeader("Content-Type","application/json");
4273
4274 // send request in post mode
4275 let bodyParams={
4276     "key1":"value1",
4277     "key2":"value2",
4278     "key3":"value3"
4279 }
4280
4281 // if you send json you will get stat=500
4282     request.send(bodyParams);
4283 // you need to convert it to string :
4284     request.send(JSON.stringify(bodyParams));
4285
4286 // type of request :
4287 /*
4288     get : get info
4289     post : send info
4290     put type : updated all information :
4291     patch : updated specific information :
4292 */
4293 // get with filtering :
4294     request.open("GET","url?queryParam=value");
4295
4296 //Example 1 : --[
4297 const xhr = new XMLHttpRequest();
4298 xhr.open('GET', 'https://api.example.com/data', true);
4299
4300 xhr.onreadystatechange = function () {
4301     if (xhr.readyState === 4) { // State 4: Done
4302         if (xhr.status === 200) { // HTTP status code 200 (OK)
4303             // Process the response data
4304             console.log(xhr.responseText);
4305         } else {
4306             // Handle errors or non-200 status codes
4307             console.error('Request failed with status:', xhr.status);
4308         }
4309     }
4310 }
```

```
4309         }
4310     );
4311
4312     xhr.send(); // Initiate the request
4313
4314 // Example 2 : -----
4315 function getPosts() {
4316     let request = new XMLHttpRequest();
4317
4318     request.open("GET", "https://jsonplaceholder.typicode.com/posts");
4319     request.responseType = "json";
4320     request.send();
4321
4322     request.onload = function () {
4323         if (request.status < 200 || request.status > 300) alert("server error");
4324         else {
4325             let posts = request.response;
4326
4327             for (element of posts) {
4328                 document.getElementById(
4329                     "content"
4330                 ).innerHTML += `<h2>${element.title}</h2>`;
4331             }
4332         }
4333     };
4334 }
4335
4336 //getPosts();
4337 function createNewPost() {
4338     let request = new XMLHttpRequest();
4339     request.open("POST", "https://jsonplaceholder.typicode.com/posts");
4340     request.setRequestHeader("Accept", "application/json");
4341     request.setRequestHeader("Content-Type", "application/json");
4342     request.responseType = "json";
4343     let bodyParams = {
4344         title: "my task",
4345         body: "go sleep",
4346         userId: 1,
4347     };
4348
4349     // send the request :
4350     request.send(JSON.stringify(bodyParams));
4351
4352     request.onload = function () {
4353         if (request.status < 200 || request.status > 300) alert("server error");
4354         else {
4355             let post = this.response;
4356             console.log(post);
4357             alert("the post has been created successfully ");
4358         }
4359     };
4360 }
4361
4362 function updatePost() {
4363     let request = new XMLHttpRequest();
4364     request.open("PUT", "https://jsonplaceholder.typicode.com/posts/1");
```

```
4365     request.setRequestHeader("Accept", "application/json");
4366     request.setRequestHeader("Content-Type", "application/json");
4367     request.responseType = "json";
4368     let bodyParams = {
4369       title: "hello world",
4370       body: "bar",
4371       userId: 1,
4372     };
4373 
4374     // send the request :
4375     request.send(JSON.stringify(bodyParams));
4376 
4377     request.onload = function () {
4378       if (request.status < 200 || request.status > 300) alert("server error");
4379       else {
4380         let post = this.response;
4381         console.log(post);
4382         alert("the post has been updated successfully ");
4383       }
4384     };
4385   }
4386 
4387   function deletePost() {
4388     let request = new XMLHttpRequest();
4389     request.open("DELETE", "https://jsonplaceholder.typicode.com/posts/1");
4390     request.setRequestHeader("Accept", "application/json");
4391 
4392     // send the request :
4393     request.send();
4394 
4395     request.onload = function () {
4396       if (request.status < 200 || request.status > 300) alert("server error");
4397       else alert("the post has been deleted successfully ");
4398     };
4399   }
4400 
4401   function getPostsWithFiltering() {
4402     let request = new XMLHttpRequest();
4403 
4404     request.open("GET", "https://jsonplaceholder.typicode.com/posts/?userId=1");
4405     request.responseType = "json";
4406     request.send();
4407 
4408     request.onload = function () {
4409       if (request.status < 200 || request.status > 300) alert("server error");
4410       else {
4411         let posts = request.response;
4412 
4413         for (element of posts) {
4414           document.getElementById(
4415             "content"
4416           ).innerHTML += `<h2>${element.title}</h2>`;
4417         }
4418       }
4419     };
4420   }

```

```

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");

```

```

4476     } else {
4477         rejectFunction(Error("Connection Failed"));
4478     }
4479 }).then(
4480     (resolveValue) => console.log(`Good ${resolveValue}`),
4481     (rejectValue) => console.log(`Bad ${rejectValue}`)
4482 );
4483
4484 // method 2 :
4485 myPromise = new Promise((resolveFunction, rejectFunction) => {
4486     let connect = true;
4487     if (connect) {
4488         resolveFunction("Connection Established");
4489     } else {
4490         rejectFunction(Error("Connection Failed"));
4491     }
4492 });
4493
4494 console.log(myPromise);
4495
4496 let resolver = (resolveValue) => console.log(`Good ${resolveValue}`);
4497 let rejecter = (rejectValue) => console.log(`Bad ${rejectValue}`);
4498
4499 myPromise.then(resolver, rejecter);
4500
4501 myPromise.then(
4502     (resolveValue) => console.log(`Good ${resolveValue}`),
4503     (rejectValue) => console.log(`Bad ${rejectValue}`)
4504 );
4505
4506 myPromise.then(
4507     (resolveValue) => console.log(`Good ${resolveValue}`),
4508     (rejectValue) => console.log(`Bad ${rejectValue}`)
4509 );
4510
4511 // then && catch && finally : --[
4512 /*
4513     Promise Training
4514
4515     We Will Go To The Meeting, Promise Me That We Will Find The 4 Employees
4516     .then(We Will Choose Two People)
4517     .then(We Will Test Them Then Get One Of Them)
4518     .catch(No One Came)
4519
4520     Then    => Promise Is Successful Use The Resolved Data
4521     Catch   => Promise Is Failed, Catch The Error
4522     Finally => Promise Successful Or Failed Finally Do Something
4523 */
4524 // example :
4525 const promise = new Promise((resolve, reject) => {
4526     let emp = ["amine", "ayoub", "kamal", "amina"];
4527     if (emp.length === 4) {
4528         resolve(emp);
4529     } else {
4530         reject(Error("Number of Employees Is Not 4 : "));
4531     }

```

```

4532    })
4533      .then((resolve) => {
4534        resolve.length = 2;
4535        return resolve;
4536      })
4537      .then((resolve) => {
4538        resolve.length = 1;
4539        return resolve;
4540      })
4541      .then((resolve) => console.log(`Final resolver : ${resolve}`))
4542      .catch((error) => {
4543        console.log(error);
4544      })
4545      .finally(() => console.log("Good by"));
4546
4547 // Promise And XHR : ---[
4548 const getData = (apiLink) => {
4549   return new Promise((resolve, reject) => {
4550     let myRequest = new XMLHttpRequest();
4551
4552     // receive the response :
4553     myRequest.onload = function () {
4554       if (this.status === 200) {
4555         resolve(this.response);
4556       } else {
4557         reject(Error("Status", this.status));
4558       }
4559     };
4560
4561     myRequest.open("GET", apiLink);
4562     myRequest.responseText = "json";
4563     // send the request to the server :
4564     myRequest.send();
4565   });
4566 };
4567
4568 const apiLink = "https://api.github.com/users/elzerewebschool/repos";
4569
4570 getData(apiLink)
4571   .then((response) => {
4572     response.length = 10;
4573     return response;
4574   })
4575   .then((response) => {
4576     for (let item of response) {
4577       let div = document.createElement("div");
4578       let text = document.createTextNode(item.full_name);
4579
4580       div.append(text);
4581       div.style.cssText =
4582         padding :20px;
4583         margin : 20px auto;
4584         width : 70%;
4585         min-width : 250px;
4586         text-align : center;
4587         border: 2px doted black;

```

```

4588     `;
4589         document.body.appendChild(div);
4590     }
4591   })
4592   .catch((error) => console.log(error));
4593
4594 // Fetch API : -----
4595 /*
4596   fetch is a JavaScript function that allows you to make network requests
4597   (typically HTTP requests) to
4598   fetch resources from a network, such as JSON data from a REST API, HTML
4599   from a website, or other types
4600   of data. It's widely used in modern web development for making asynchronous
4601   requests to web servers.
4602 */
4603   fetch(url, options)
4604 /*
4605   ✨url: The URL of the resource you want to fetch.
4606   options (optional): An object containing various options for the request,
4607   including the HTTP method, headers, request body, and more.
4608   ✨Creating a Request:
4609   When you call fetch, it creates and returns a Promise that represents the future
4610   response to the request.
4611   However, the request is not sent immediately; it's only prepared at this stage.
4612   ✨Configuring the Request:
4613   You can specify various options in the options object to configure the request:
4614
4615     ✨method: The HTTP method (e.g., 'GET', 'POST', 'PUT', 'DELETE') to use for the
4616   request.
4617     headers: An object containing the HTTP headers for the request, such as '
4618       Content-Type' and
4619       'Authorization'.
4620
4621     ✨body: The request body, typically used for sending data in POST or PUT requests.
4622     It should be a string or a FormData object.
4623
4624     ✨mode: The request mode (e.g., 'cors', 'no-cors', 'same-origin') that defines how
4625   cross-origin
4626   requests are handled.
4627
4628     ✨credentials: Indicates whether to include cookies or credentials with the request
4629   ('same-origin', 'include', 'omit').
4630
4631     ✨cache: The caching mode for the request ('default', 'no-store', 'reload', etc.).
4632
4633     ✨redirect: How to handle redirects ('follow', 'error', 'manual').
4634     And more.
4635 */
4636 // ✨Sending the Request: --[
4637 /*
4638   To actually send the request, you need to call .then() or use async/await on the
4639   returned Promise.
4640   This initiates the network request to the specified URL with the provided options.
4641 */
4642   fetch(url, options)

```

```

4640     .then(response => {
4641         // Handle the response here
4642     })
4643     .catch(error => {
4644         // Handle errors here
4645     });
4646
4647 // ⚡Handling the Response: --[
4648 /*
4649     Once the request is sent, the fetch function returns a Promise that resolves
4650     with a Response object representing the response from the server.
4651     You can then use methods and properties of this Response object to handle the response
4652     data.
4653
4654     Common methods and properties of the Response object include:
4655     .json(): Parses the response body as JSON.
4656     .text(): Reads the response body as text.
4657     .blob(): Returns the response body as a binary Blob.
4658     .headers: Access to the response headers.
4659     .status: HTTP status code (e.g., 200 for OK, 404 for Not Found).
4660     .statusText: HTTP status message (e.g., "OK", "Not Found").
4661 */
4662
4663 // ⚡Handling Errors: --[
4664 /*
4665     If the network request fails or encounters an error (e.g., due to a network issue,
4666     invalid URL, or server error), the Promise is rejected, and you can catch the error
4667     using .catch().
4668
4669     Here's an example of using fetch to make a GET request and handle the response:
4670 */
4671 // Example 1:
4672     fetch('https://jsonplaceholder.typicode.com/posts/1')
4673     .then(response => {
4674         if (!response.ok) {
4675             throw new Error('Network response was not ok');
4676         }
4677         return response.json(); // Parse response body as JSON
4678     })
4679     .then(data => {
4680         console.log(data); // Process the JSON data
4681     })
4682     .catch(error => {
4683         console.error('Fetch error:', error);
4684     });
4685
4686 //Example 2 :
4687 fetch("https://api.github.com/users/elzerewebschool/repos")
4688     .then((result) => {
4689         if (result.ok) return result.json();
4690
4691         throw new Error(result.statusText);
4692     })
4693     .then((myData) => {
4694         myData.length = 10;
4695         return myData;

```

```

4695 })
4696 .then((myData) => {
4697   for (let item of myData) {
4698     let div = document.createElement("div");
4699     let text = document.createTextNode(item.full_name);
4700
4701     div.append(text);
4702     div.style.cssText =
4703       padding :20px;
4704       margin : 20px auto;
4705       width : 70%;
4706       min-width : 250px;
4707       text-align : center;
4708       border: 2px doted black;
4709     `;
4710     document.body.appendChild(div);
4711   }
4712 });
4713
4714 // post Request steps : --[
4715 // Define the URL for the API endpoint
4716 const Url = 'https://api.example.com/endpoint';
4717
4718 // Create an object with the data you want to send in the request body
4719 const data = {
4720   param1: 'value1',
4721   param2: 'value2'
4722 };
4723
4724 // Create the request options, including method, headers, and body
4725 const requestOptions = {
4726   method: 'POST', // or 'GET', 'PUT', 'DELETE', etc.
4727   headers: {
4728     'Content-Type': 'application/json' // specify the content type if sending JSON data
4729     // Add any other headers if needed
4730   },
4731   body: JSON.stringify(data) // Convert the data object to a JSON string
4732 };
4733
4734 // Use the fetch function to make the request
4735 fetch(Url, requestOptions)
4736 .then(response => {
4737   // Check if the request was successful (status code 200-299)
4738   if (!response.ok) {
4739     throw new Error(`HTTP error! Status: ${response.status}`);
4740   }
4741   // Parse the response JSON
4742   return response.json();
4743 })
4744 .then(data => {
4745   // Do something with the data returned from the API
4746   console.log(data);
4747 })
4748 .catch(error => {
4749   // Handle errors
4750   console.error('Fetch error:', error);

```

```
4751 });
4752
4753 // Some useful Classes : --[
4754   "1. ** Headers` Class**:"
4755   // The `Headers` class represents a collection of HTTP headers associated with a
4756   // request or response. You can use it to manipulate headers before sending a request
4757   // or after receiving a response.
4758   headers = new Headers();
4759   headers.append('Content-Type', 'application/json');
4760   headers.set('Authorization', 'Bearer Token');
4761
4762
4763   "2. **`FormData` Class**:"
4764   // The `FormData` class allows you to create and manipulate form data that can be sent
4765   // in a
4766   // network request. You can use it to build and send form data in a POST request.
4767   formData = new FormData();
4768   formData.append('username', 'john_doe');
4769   formData.append('password', 'secure_password');
4770
4771
4772   "3. **`URL` and `URLSearchParams` Classes**: "
4773   // The `URL` class represents a URL, and the `URLSearchParams` class is used for
4774   // working
4775   // with URL query parameters. You can use these classes to parse and manipulate URLs.
4776   url = new URL('https://example.com/api');
4777   url.searchParams.append('param1', 'value1');
4778   url.searchParams.append('param2', 'value2');
4779
4780 // Example : --[
4781   // Create a new FormData object
4782   const formData = new FormData();
4783
4784   // Append form data fields
4785   formData.append('username', 'john_doe');
4786   formData.append('password', 'secretpassword');
4787
4788   // Create a new URL object with the endpoint
4789   const url = new URL('https://api.example.com/login');
4790
4791   // Create a new Headers object and set headers
4792   const headers = new Headers();
4793   headers.append('Content-Type', 'application/x-www-form-urlencoded'); // Example header
4794
4795   // Make a fetch request with form data, URL, and headers
4796   fetch(url, {
4797     method: 'POST', // Specify the HTTP method
4798     headers: headers, // Pass the Headers object with headers
4799     body: formData, // Pass the FormData object as the body
4800   })
4801   .then(response => {
4802     if (!response.ok) {
4803       throw new Error('Network response was not ok');
4804     }
4805     return response.json(); // Parse the JSON response
4806   })
4807   .then(data => {
```

```
4806     console.log('Login successful:', data);
4807   })
4808   .catch(error => {
4809     console.error('Error logging in:', error);
4810   });
4811
4812 // Async : -----
4813 /*
4814   - Async Before Function Mean This Function Return A Promise
4815   - Async And Await Help In Creating Asynchronous Promise Behavior With Cleaner Style
4816 */
4817 // method 1 : new Promise Method
4818 function getData() {
4819   return new Promise((res, rej) => {
4820     let users = [];
4821     if (users.length > 0) {
4822       res("Users Found");
4823     } else {
4824       rej("No Users Found");
4825     }
4826   });
4827 }
4828 getData().then(
4829   (resolvedValue) => console.log(resolvedValue),
4830   (rejectedValue) => console.log("Rejected " + rejectedValue)
4831 );
4832
4833 // method 2 : using Promise.[resolve / reject]
4834 function getData() {
4835   let users = ["Osama"];
4836   if (users.length > 0) {
4837     return Promise.resolve("Users Found");
4838   } else {
4839     return Promise.reject("No Users Found");
4840   }
4841 }
4842
4843 getData().then(
4844   (resolvedValue) => console.log(resolvedValue),
4845   (rejectedValue) => console.log("Rejected " + rejectedValue)
4846 );
4847
4848 //method 3 : using async function
4849 async function getData() {
4850   let users = [];
4851   if (users.length > 0) {
4852     return "Users Found";
4853   } else {
4854     throw new Error("No Users Found");
4855   }
4856 }
4857 getData().then(
4858   (resolvedValue) => console.log(resolvedValue),
4859   (rejectedValue) => console.log("Rejected " + rejectedValue)
4860 );
4861
```

```
4862 // Await : -----
4863 /*
4864     - Await Works Only Inside Async Functions
4865     - Await Make JavaScript Wait For The Promise Result
4866     - Await Is More Elegant Syntax Of Getting Promise Result
4867 */
4868 // examples :
4869 myPromise = new Promise((resolve, reject) => {
4870     setTimeout(() => {
4871         // resolve("Iam The Good Promise");
4872         reject(Error("Iam The Bad Promise"));
4873     }, 3000);
4874 });
4875
4876 async function readData() {
4877     console.log("Before Promise");
4878     // myPromise.then(resolvedValue) => console.log(resolvedValue));
4879     // console.log(await myPromise);
4880     console.log(await myPromise.catch((err) => err));
4881     console.log("After Promise");
4882 }
4883
4884 readData();
4885
4886 // Try, Catch And Finally With Fetch : -----
4887 // example :
4888     myPromise = new Promise((resolve, reject) => {
4889         setTimeout(() => {
4890             resolve("Iam The Good Promise");
4891         }, 3000);
4892     });
4893
4894     async function readData() {
4895         console.log("Before Promise");
4896         try {
4897             console.log(await myPromise);
4898         } catch (reason) {
4899             console.log(`Reason: ${reason}`);
4900         } finally {
4901             console.log("After Promise");
4902         }
4903     }
4904
4905 readData();
4906
4907 // "https://api.github.com/users/elzeroweb學校/repos"
4908
4909     async function fetchData() {
4910         console.log("Before Fetch");
4911         try {
4912             let myData = await fetch("https://api.github.com/users/elzeroweb學校/repos");
4913             console.log(await myData.json());
4914         } catch (reason) {
4915             console.log(`Reason: ${reason}`);
4916         } finally {
4917             console.log("After Fetch");
```

```

4918     }
4919   }
4920   fetchData();
4921
4922
4923 //axios in js :  get Users Using axios Library : -----
4924   const axios = require("axios"); // Import Axios in a Node.js environment
4925
4926 // Example: Making a POST request with request body parameters
4927   axios.post("https://example.com/api/resource", {
4928     key1: "value1",
4929     key2: "value2",
4930   }, {
4931     headers: {
4932       // Define your custom headers here
4933       "Content-Type": "application/json", // Set the appropriate content type
4934       Authorization: "Bearer your-access-token", // Optional: Include an authorization
4935       header
4936     },
4937   })
4938   .then((response) => {
4939     console.log(response.data); // Process the data from the response
4940   })
4941   .catch((error) => {
4942     console.error("Axios error:", error);
4943   });
4944
4945 // auto catch of errors :
4946 function getUsersAxios() {
4947   return new Promise((resolve, reject) => {
4948     axios
4949       .get(Users)
4950       .then((response) => {
4951         return response.data;
4952       })
4953       .then((Users) => {
4954         console.log(Users);
4955         resolve();
4956       })
4957       .catch((error) => {
4958         reject(error);
4959       });
4960   });
4961 }
4962
4963 function getPostsAxios(UserId) {
4964   let PostUrl = posts + UserId;
4965
4966   axios
4967     .get(PostUrl)
4968     .then((response) => {
4969       return response.data;
4970     })
4971     .then((Posts) => {
4972       console.log(Posts);

```

```

4973         })
4974         .catch((error) => {
4975             reject(error);
4976         });
4977     }
4978
4979     }
4980
4981     getUsersAxios()
4982         .then(() => {
4983             return getPostsAxios(1);
4984         })
4985         .catch((error) => {
4986             console.log("Error From :", error);
4987         })
4988
4989 // npm node package manager : -----
4990 /*
4991     ⚡ download node js : node -v to know the version
4992     ⚡ initialize the project => npm init
4993     ⚡ install library : npm install libraryName --save(save in in package.json)
4994     ⚡ package.json : information about library that you have installed :
4995     ⚡ node modules : contain the code of all library
4996     ⚡ Lock.json specify version of libraries :
4997 */
4998 // ⚡ important information : -----
4999 /*
5000     if you use just write : npm install
5001     the npm will take information library
5002     in the package file then it will install it and add to your
5003     project
5004     very helpful when you work in a team and you want to install the library
5005     of your team to work with, just you need to get the package file
5006     then write the npm install command then the npm will install
5007     all library in the package.json with the same specification :
5008 */
5009
5010 // ⚡ Last step import axios from node modules ✓ -----
5011 /**
5012 *
5013     you use the require just when you working with frameworks :(Angular,React...)
5014
5015     due to we work just with pure js we need to import the axios.js
5016     manually
5017     like this :
5018     <script src="../node_modules/axios/axios.js"></script>
5019     or :
5020     <script src="../node_modules/axios/dist/axios.min.js"></script>
5021 */
5022
5023
5024 // await and async : -----
5025 // simple way to get Users and Posts using simple fetch
5026 // without apply await ans async keyword
5027
5028 /*

```

```
5029 but firstly let's introduce the two fundamentals :  
5030 await: to keep the js await until an async code finished  
5031 it's veery useful when you handel with api setttimeout ..  
5032 and many foundations in js :  
5033 important notion about it :  
5034  
5035 be carefully because you can use await only in an async function :  
5036  
5037 let's freaking out the second concept :  
5038 async : this keyword using the define an async function :  
5039 and it's provide as to use the async keyword into functions  
5040 and make the function automatically return Promise  
5041 and the return keyword Represent the Resolve() function in the promise  
5042  
5043 async functionName(){  
5044     async code  
5045  
5046     return Anything // resolve(Anything)  
5047 }  
5048  
5049 /*  
5050 // version 1: -----[]  
5051 /*  
5052     in this version bellow : the code is writing  
5053     just with fetch functions without using await and async  
5054     keyword  
5055  
5056     how can see that's the implementation of it  
5057     it's a little bit difficult to Read an maintain :  
5058     specially when you work on multiple then in the same promise :  
5059     and for that the await an async founded to solve this problem exactly  
5060     features of await and async :  
5061  
5062     1-more readability of code :  
5063     2- easy the maintain  
5064  
5065 */  
5066 let usersUrl = "https://jsonplaceholder.typicode.com/users";  
5067 let postsUrl = "https://jsonplaceholder.typicode.com/posts/?userId=";  
5068  
5069 /*  
5070     now Let's develop this code : and make it more useful  
5071     by Provide to getUsers first then get Posts after it :  
5072     following the order :  
5073 */  
5074     function getUsers() {  
5075         // fetch the data from api placeholder api :  
5076         return new Promise((resolve, reject) => {  
5077             fetch(usersUrl)  
5078                 .then((response) => {  
5079                     // check status if ok or not :  
5080  
5081                     if (!response.ok) throw new Error(response.statusText);  
5082  
5083                     return response.json();  
5084             })
```

```

5085     .then((Users) => {
5086         console.log("All Users : ", Users);
5087         resolve("Success to get User Response : ");
5088     })
5089     .catch((error) => {
5090         console.log(error);
5091         reject("Failed to get User Response there has been an error");
5092     });
5093 }
5094
5095
5096 function getPosts(UserId) {
5097     // fetching data from basic PostsUrl +UserId;
5098     return new Promise((resolve, reject) => {
5099         fetch(postsUrl + UserId)
5100             .then((response) => {
5101                 // check status if ok or not :
5102
5103                 if (!response.ok) throw new Error(response.statusText);
5104
5105                 return response.json();
5106             })
5107             .then((posts) => {
5108                 console.log(`Posts Related to User[${UserId}]`, posts);
5109                 resolve("Success to get the API response ");
5110             })
5111             .catch((error) => {
5112                 console.log(error);
5113                 reject("there has been an error during fetching the API Response");
5114             });
5115     });
5116 }
5117
5118 getUsers()
5119     .then(() => getPosts(1))
5120     .catch((error) => {
5121         console.log("Error :", error);
5122     });
5123
5124 // In the provided code:
5125 async function fetchData() {
5126     try {
5127         let response = await fetch('https://api.example.com/data');
5128         let data = await response.json();
5129         return data;
5130     } catch (error) {
5131         console.error('Error fetching data:', error);
5132         throw error;
5133     }
5134 }
5135
5136 // version 2 with await and async functions : -----
5137 async function getUsers() {
5138     let response = await fetch('https://jsonplaceholder.typicode.com/users');
5139
5140     if (!response.ok) return response.statusText;

```

```

5141     let Users = await response.json();
5142
5143     console.log('Users', Users);
5144     return;
5145 }
5146
5147 let PostUrl = "https://jsonplaceholder.typicode.com/posts/?userId=";
5148
5149 async function getPosts(UserId) {
5150     let response = await fetch(PostUrl + UserId);
5151
5152     if (!response.ok) return response.statusText;
5153     let Posts = await response.json();
5154
5155     console.log(`Posts[${UserId}]`, Posts);
5156     return;
5157 }
5158
5159 async function getData(){
5160
5161     await getUsers();
5162         getPosts(1);
5163
5164 }
5165
5166 getData();
5167
5168
5169 /*
5170 The `throw error;` statement inside the `catch` block is throwing the error again
5171 after it has been Logged. When an error is
5172 thrown within a `catch` block, it propagates the error up the call stack. In the
5173 context of an `async` function like `fetchData()`,
5174     if you call `fetchData()`, and an error occurs during the execution of `fetchData()
5175 `,
5176     the function will reject with the thrown error.
5177
5178 Here's how it works:
5179
5180 1. The `fetch` API is used to make an HTTP request to 'https://api.example.com/data'.
5181 2. If the request fails (for example, due to network issues or an invalid URL),
5182     `fetch` will reject with an error.
5183
5184 3. The `await fetch(...)` expression inside the `try` block will throw an error.
5185 4. The code inside the `catch` block will execute, logging the
5186     error to the console using `console.error('Error fetching data:', error);`.
5187
5188 5. After Logging the error, `throw error;`
5189     re-throws the error, causing the `fetchData()` function to reject with this error.
5190
5191 When you call `fetchData()`, you can handle the rejection
5192     by using `.catch()` or `try/catch` blocks in the calling code. For example:
5193
5194 */
5195 fetchData()
5196     .then(data => {

```

```

5196     // Handle successful data retrieval
5197     console.log('Data:', data);
5198   })
5199   .catch(error => {
5200     // Handle the error from fetchData() here
5201     console.error('Error in fetchData():', error);
5202   });
5203 /*
5204   In this case, if there's an error during the execution of `fetchData()`,
5205   it will be caught in the `catch()` block, where you can handle
5206   it appropriately.
5207 */
5208
5209 // example 2: --[
5210 let titles = document.querySelectorAll("h1");
5211
5212 function changeVisibility(index) {
5213   return new Promise((resolve) => {
5214     setTimeout(() => {
5215       titles[index].style.visibility = "visible";
5216       resolve(index);
5217     }, 1000);
5218   });
5219 }
5220
5221 async function ChangeTitlesVisibility() {
5222   for (let i = 0; i < titles.length; i++) {
5223     await changeVisibility(i);
5224   }
5225 }
5226 }
5227
5228 ChangeTitlesVisibility();
5229
5230 //Authentication : -----
5231 //https://reqres.in/
5232 //type of token :
5233 /*
5234   Bearer token : (headers) Authorization =Bearer token
5235   normal token : (headers) Authorization = token
5236 */
5237 //Login :
5238 let loginUrl = "https://reqres.in/api/login";
5239 let UserUrl = "https://reqres.in/api/users";
5240
5241 let loginToken= localStorage.getItem("userToken") || "";
5242 function login() {
5243   axios
5244     .post(loginUrl, {
5245       email: "tracey.ramos@reqres.in",
5246       password: "cityslicka",
5247     })
5248     .then((response) => response.data)
5249     .then((Token) => {
5250       console.log(Token);

```

```
5252         loginToken = Token.token;
5253         localStorage.setItem("userToken",loginToken);
5254         createNewUser();
5255     })
5256     .catch((error) => {
5257         alert(error);
5258     });
5259 }
5260
5261
5262 function createNewUser() {
5263     let config = {
5264         headers: {
5265             "Authorization": "Bearer " + loginToken,
5266         },
5267     };
5268
5269     axios
5270         .post(
5271             UserUrl,
5272             {
5273                 name: "majid",
5274                 job: "leader",
5275             },
5276             config
5277         )
5278         .then((response) => response.data)
5279         .then((newUserInfo) => {
5280             console.log(newUserInfo);
5281         })
5282         .catch((error) => {
5283             console.log(error);
5284         });
5285     }
5286
5287 // using await and async function : -----
5288     loginUrl = "https://reqres.in/api/login";
5289     UserUrl = "https://reqres.in/api/users";
5290     let registerUrl = "https://reqres.in/api/register";
5291
5292     loginToken = localStorage.getItem("userToken") || "";
5293
5294     let bodyPrams = {
5295         email: "tracey.ramos@reqres.in",
5296         password: "cityslickda",
5297     };
5298
5299     async function login() {
5300         try {
5301             let response = await axios.post(loginUrl, bodyPrams);
5302
5303             let token = response.data;
5304             console.log(token);
5305             localStorage.setItem("userToken", token.token);
5306         } catch (error) {
5307             console.log("Error : ", error.message);
5308         }
5309     }
5310 }
```

```
5308     }
5309 }
5310
5311     headers = {
5312         Authorization: "Bearer " + loginToken,
5313     };
5314
5315     let UserINfo = {
5316         name: "majid",
5317         job: "leader",
5318     };
5319     async function createNewUser() {
5320         try {
5321             let response = await axios.post(UserUrl, UserINfo, headers);
5322
5323             let newUserInfo = response.data;
5324             console.log(newUserInfo);
5325         } catch (error) {
5326             console.log("Error : ", error.message);
5327         }
5328     }
5329
5330     async function register() {
5331         try {
5332             let response = await axios.post(registerUrl, bodyPrams);
5333
5334             let registerInfo = response.data;
5335             console.log(registerInfo);
5336             console.log("token : ", registerInfo.token);
5337             localStorage.setItem("userToken", registerInfo.token);
5338         } catch (error) {
5339             console.log("Error : ", error.message);
5340         }
5341     }
5342
5343     async function main() {
5344         console.log("\nRegister new user : ");
5345         await register();
5346
5347         console.log("\nlogin to my created account : ");
5348         await login();
5349
5350         console.log("\ncreate a new user using my token : ");
5351         await createNewUser();
5352     }
5353
5354     main();
5355     login();
5356
5357 // delay function : -----
5358 function delay(ms) {
5359     return new Promise(resolve => setTimeout(resolve, ms));
5360 }
5361
5362 /*
5363     by using the Login function we send request to api then api
```

```
5364 generate a token that we received then save in Local Storage
5365 to provide to user enter directly to his account
5366 without the need to login again by using CreateUser function :
5367 that take Token from Local Storage then created a new user
5368 */
5369 /*
5370 To navigate to another page using JavaScript, you can use the `window.Location` object.
5371 Here are a few common ways to achieve this:
5372
5373 #### 1. Using `window.Location.href`:
5374 You can set the `window.Location.href` property to the URL of the page you want to
5375 navigate to. For example:
5376
5377 ````javascript
5378 Navigate to a new page
5379 window.Location.href = "https://www.example.com/newpage.html";
5380 `````
5381
5382
5383 #### 2. Using `window.Location.assign()` method:
5384 The `assign()` method of the `window.Location` object is another way to navigate to a
5385 new page:
5386
5387 ````javascript
5388 Navigate to a new page
5389 window.Location.assign("https://www.example.com/newpage.html");
5390 `````
5391
5392 #### 3. Using `window.Location.replace()` method:
5393 The `replace()` method of the `window.Location` object can be used
5394 to navigate to a new page and replace the current
5395 page in the browser history. This means the user cannot navigate
5396 back to the original page using the browser's back button.
5397
5398 ````javascript
5399 Navigate to a new page and replace the current page in the browser history
5400 window.Location.replace("https://www.example.com/newpage.html");
5401 `````
5402 Choose the appropriate method based on your specific use case and whether you want the
5403 new
5404 page to be added to the browser history or replace the current page.
5405
5406 Authentication and tokens are fundamental concepts in the realm of security
5407 and identity management, especially in the context of web applications and
5408 APIs. Let's explore these concepts in detail:
5409
5410 #### Authentication:
5411
5412 Authentication is the process of verifying the identity of a user, application,
5413 or system. It ensures that the entity trying to access a resource is who it claims to be
5414 . There are various methods of authentication, each with its own strengths and use
5415 cases:
5416
5417 1. **Username/Password:**  

5418 - The user provides a username and password.
```

- 5417 - Common for web applications and traditional login systems.  
5418 - Vulnerable to various attacks, such as phishing.
- 5419
- 5420 2. **Multi-Factor Authentication (MFA):**  
5421 - Requires multiple forms of identification (e.g., password + SMS code or fingerprint).  
5422 - Enhances security by adding an additional layer of verification.
- 5423
- 5424 3. **Token-Based Authentication:**  
5425 - Uses tokens (e.g., JSON Web Tokens) for authentication.  
5426 - Reduces the need to store sensitive credentials on the client.  
5427 - Often used in modern web applications and APIs.
- 5428
- 5429 4. **OAuth and OpenID Connect:**  
5430 - Delegated authorization and authentication protocols.  
5431 - OAuth allows secure delegated access to resources.  
5432 - OpenID Connect is an identity layer on top of OAuth, providing authentication.
- 5433
- 5434 **Tokens:**
- 5435
- 5436 *Tokens play a crucial role in modern authentication and authorization systems.*  
5437 *They are used to represent the authenticated user and provide secure access*  
5438 *to protected resources. The most common types of tokens are:*
- 5439
- 5440 1. **Access Tokens:**  
5441 - Grants access to specific resources on behalf of the user.  
5442 - Short-lived and specific to the user and application.  
5443 - Used in OAuth for authorization.
- 5444
- 5445 2. **Refresh Tokens:**  
5446 - Used to obtain a new access token.  
5447 - Longer-lived than access tokens.  
5448 - Stored securely on the client.
- 5449
- 5450 3. **JSON Web Tokens (JWT):**  
5451 - A compact, URL-safe means of representing claims between two parties.  
5452 - Self-contained, containing information about the user or system.  
5453 - Often used as access tokens in token-based authentication.
- 5454
- 5455 **Token-Based Authentication Flow:**
- 5456
- 5457 1. **User Authentication:**  
5458 - The user provides credentials (e.g., username/password) to the authentication server.
- 5459
- 5460 2. **Token Issuance:**  
5461 - Upon successful authentication, the authentication server generates an access token  
5462 (and optionally a refresh token).
- 5463
- 5464 3. **Token Storage:**  
5465 - The access token is stored securely on the client (e.g., in a cookie or local storage)
- 5466
- 5467 4. **Token Usage:**  
5468 - The client includes the access token in the headers of API requests to access  
protected resources.
- 5469
- 5470 5. **Token Expiry and Refresh:**  
5471 - Access tokens have a limited lifespan. If they expire, the client can use the refresh

5472        token to obtain a new access token without requiring the user to re-enter credentials.  
5473  
5474        **#### Benefits of Token-Based Authentication:**  
5475  
5476        1. **\*\*Statelessness:\*\***  
5477        - No need to store user sessions on the server.  
5478        - Each request contains the necessary authentication information.  
5479  
5480        2. **\*\*Scalability:\*\***  
5481        - Stateless nature simplifies scaling, as there's no need to synchronize session state  
5482        across multiple servers.  
5483  
5484        3. **\*\*Security:\*\***  
5485        - Tokens can be encrypted and signed to ensure integrity and confidentiality.  
5486        - Reduced risk of Cross-Site Request Forgery (CSRF) and session hijacking.  
5487  
5488        4. **\*\*Decoupling Frontend and Backend:\*\***  
5489        - The frontend and backend can be developed independently, as long as they adhere to the  
token contract.  
5490  
5491        In summary, authentication verifies the identity of a user or application, while tokens  
play a crucial  
5492        role in securely representing and granting access to resources. Token-based  
authentication,  
5493        especially using technologies like OAuth and JWTs, has become a standard in modern web  
development  
5494        due to its security, scalability, and flexibility.  
5495        \*/