

## Script.js

```
1          /* #. 🙋🙋 Value and Variable 🙋🙋
2
3 // 1.Naming Variable:Rules and Best Practices
4
5 // var my_firstName='Adarsh';   ✓
6
7 //var 123Name='Adarsh';   ✗
8
9 //var _myLastName='Rai'   ✓
10
11 //var $cityName='Bhiwandi'   ✓
12
13 //var my@Email='adarshrai@gmail.com'   ✗
14
15
16
17
18
19
20          /* #. 🙋🙋 Data Type 🙋🙋
21
22 //?🚫 Primitive Data Type
23
24 //1.String    var name='Adarsh'
25 //2.Number    var num=45
26 //3.Boolean    var isRaining =true
27 //4.BigInt    const bigInt=12345678912345+78945624892559962355n
28 //5.Undefined    var name;
29 //6.Null        var name=null
30 //7.Symbol        const mysymbol=symbol('description')
31
32
33 //?🚫 Object Data Type
34
35 //1.An Object
36 //2.An Array
37 //3. Date
38
39
40          /* #. ✗✗ InterView Question ✗✗
41
42
43 //!1.What is the perpose of typeof operator
44 //ans👉 use to find the datatype of a variable
45
46 //Example👉
47
48 // var num=56;
49 // console.log(typeof num);
50
51 // var myname='adarsh'
```

```

52 // console.log(typeof myname);
53
54
55
56 //!2.what is the result of typeof null
57 // console.log(typeof null); //result is object⟹ its a javascript bug
58
59
60
61 //!3.convert a string to number
62 //⟹ we just need to add "+" sign before the string
63 //Example 🙋
64
65 // var myString = '10'
66 // console.log(typeof +myString);
67 // console.log(typeof Number(myString)); using Number Function
68
69
70
71 //!4.convert a number to string
72 //⟹ we just need to add a empty string after the number
73 //Example 🙋
74 // var str=5;
75 // console.log(typeof (str + ''));
76 // console.log(typeof String(str)); //using string function
77
78
79
80
81 //!5.what are the truthy and falsy value
82
83 // 🙋 Truthy Value
84
85 // 🙋 true
86 // 🙋 any non-empty string('hello')
87 // 🙋 any non-zero number (25)
88 // 🙋 array and object
89
90
91
92 // 🙋 Falsy Value
93 // 🙋 False
94 // 🙋 any empty string('')
95 // 🙋 any zero number (0)
96 // 🙋 Undefined
97 // 🙋 Null
98 // 🙋 NaN(Not a Number)
99
100
101
102
103 // 🎁🎁 Bonus 🎁🎁
104 // ❤️❤️ ParseInt and ParseFloat
105

```

```

106 //👉👉👉 Both are the javaScript function used to convert string to number
107
108
109 //!1.❤️parseInt    ==> used to convert string to into Integer Value eliminate
decimal value
110
111 //Example👉
112 // var myString = '55'
113 // var myNumber=parseInt(myString)
114 // console.log(typeof myNumber);
115
116 // var myString=55.5
117 // console.log(parseInt(myString));    return only 55 not 55.5
118
119
120
121 //!2.❤️parseFloat ==> used to cnvert string to any number including float value
122 //Example👉
123 // var myString = '55.5'
124 // var myNumber=parseFloat(myString)
125 // console.log(myNumber);
126 // console.log(typeof myNumber);
127
128
129
130 //! 🤔🤔What will be the output of 🤔🤔🤔
131 // console.log(parseInt('077')); //==> 77 not print 0
132 // console.log(parseInt('-123')); //==> -123
133 // console.log(parseInt('@123')); //==> NaN
134 // console.log(parseInt('xyz')); //==> NaN(Nan Stand or Not a number return a
when a methamatical operation doesn't yield a valid number)
135
136
137 //To check a value a number or not use isNaN() function
138 //Example👉
139 // console.log(isNaN('xyz')); //==> true
140 // console.log(isNaN('56')); //==> false because it is a number ignore quotes
141
142
143
144
145
146
147
148 //*🔧🔧Expression & Opeartors🔧🔧
149
150 //?🤔🤔Type of Operator
151
152 //1.👉Assignment Operator    (=)
153 //2.👉Arithmetic Operator    (+,-,*,/,%)
154 //3.👉Comparison Operator    (=,==,<,>,≤,≥)
155 //4.👉Logical Operator        (AND(&&), OR(||),NOT(!))
156 //5.👉String Opeartor        (+)
157 //6.👉Unary Operator

```

```

158 //7. 🖱️ Ternary Operator (var result=a ≥ 18?'can vote':'can't vote')
159 //8. 🖱️ Typeof Operator (typeof) ⇒ check type of data type
160
161
162
163
164
165
166
167 /* 🖱️🖱️ Control Statement & Loops 🖱️🖱️
168
169 //1. 🖱️ IF-Else Statement
170 //2. 🖱️ Switch Statement
171 //3. 🖱️ While Loop
172 //4. 🖱️ Do-While Loop
173 //5. 🖱️ For-In /For-of Loop
174
175
176
177
178
179 //?1. 🖱️ IF-Else Statement
180
181 //Syntax 🖱️
182
183 //if(condition){ code .... }
184 // else{ code ... }
185
186
187
188 //Example 🖱️
189 // var temp=40
190 // if(temp>30){
191 //     console.log('lets go beach.. 🏖️🏖️');
192
193 // }
194 // else{
195 //     console.log('Watch Tv at Home 📺👁️');
196
197 // }
198
199
200 //we can also use an else if clause to check additional condition
201
202 // var temp=25;
203 // if(temp>30){
204 //     console.log('lets go beach.. 🏖️🏖️');
205 // }
206 // else if(temp>20 && temp<30){
207 //     console.log('Watch Tv at Home 📺👁️');
208
209 // }
210 // else{
211 //     console.log('thandi hai so jao');

```

```

212
213 // }
214
215
216
217
218 //!InterView Question 1 ? ? ?
219
220 //if the person is yonger than 18 , not a citizen , or not
221 // registered to vote display a message saying they are not eligible to vote
222
223 //if the person is 18 or older but not a citizen,display a message
224 // saying they are not eligible due to citizenship status
225
226 //if the person is 18 or older a citizen but not registerd to vote display
227 //a message they are not eligible due to registration status
228
229 //Ans 👉👉👉👉👉👉👉👉👉👉👉👉👉👉👉
230
231 // var age=17;
232 // var isCitizenOfIndia=false;
233 // var isRegistredForVote=false;
234
235 // if(age ≥ 18 && isCitizenOfIndia && isRegistredForVote){
236 //     console.log('Congratulation you can vote');
237
238 // }
239 // else if(age ≥ 18 && !isCitizenOfIndia && !isRegistredForVote){
240 //     console.log('you are not eligible');
241
242 // }
243 // else if(age ≥ 18 && !isCitizenOfIndia && isRegistredForVote){
244 //     console.log('you are not eligible due to citizenship status');
245
246 // }
247 // else if(age ≥ 18 && isCitizenOfIndia && !isRegistredForVote){
248 //     console.log('you are not eligible due to registration status');
249
250 // }
251 // else{
252 // console.log('you are not eligible due to age,citizenship status and
registration status');
253
254 // }
255
256
257
258
259 //Method 2 👉👉👉:
260
261 // var age=12;
262 // var isCitizenOfIndia=false;
263 // var isRegistredForVote=true;
264

```

```

265 // if(age ≥ 18){
266 //     if(isCitizenOfIndia){
267 //         if(isRegistredForVote){
268 //             console.log('Congratulation you can vote');
269 //         }
270 //         else{
271 //             console.log('you are not eligible due to registration status');
272 //         }
273 //     }
274 //     else{
275 //         console.log('you are not eligible due to citizenship status');
276 //     }
277 // }
278 // else{
279 //     console.log('You are not eligible to vote');
280 // }
281 // }
282
283
284
285 //!InterView Question 2 ? ? ?
286
287 // ? ? write a program to check if a number is even or odd
288
289 // var num=13;
290 // if(num%2==0){
291 //     console.log('num is even');
292 // }
293 // }
294 // else{
295 //     console.log('num is odd');
296 // }
297 // }
298
299
300
301 //! ? ? write a program to check if a number is Prime
302 // var num=13;
303 // var isPrime=true
304
305 // for(var i=2;i<num;i++){
306 //     if(num%i==0){
307 //         isPrime=false;
308 //         break;
309 //     }
310 // }
311 // if(isPrime){
312 //     console.log('num is prime');
313 // }
314 // }
315 // else{
316 //     console.log('num is not prime');
317 // }
318 // }

```

```

319
320 //! ?? write a program to check if a number is positive , negative or zero
321
322 // var num =-1
323 // if(num===0){
324 //     console.log('num is zero');
325 // }
326 // else if(num>0){
327 //     console.log('num is positive');
328 // }
329 // else{
330 //     console.log('num is negative');
331 // }
332
333
334
335 //?2.Switch Statement 🧠 🧠
336 //Syntax 👉 👉
337
338 // switch(expression){
339 //     case value1:
340 //         code
341 //         break;
342
343 //     case value1:
344 //         code
345 //         break;
346
347 //     default:
348 // }
349
350
351 //Example 👉 👉 👉 👉 👉
352
353 // var day='Monday';
354
355 // switch(day){
356 //     case 'Monday':
357 //         console.log('Today is Monday');
358 //         break
359
360
361 //     case 'Friday':
362 //         console.log('lets party 🎉 🎊 🎊');
363 //         break
364
365 //     case 'Sunday':
366 //         console.log('lets go for movie');
367 //         break
368
369 //     default:
370 //         console.log('no condition match');
371
372 // }

```

```

373
374
375
376
377 //?3.👀👀 While Loop👀👀
378
379 //syntax 🙋🙋🙋🙋
380 // while (condition) {
381
382 // }
383
384 //Example⇒ Print 1 to 10 number using while loop !? !? !?
385
386 // var i=1;
387 // while (i≤10) {
388 //     console.log(i);
389 //     i++;
390
391 // }
392
393 //!Create a Table of 5 using while loop
394
395 // var num=5;
396 // var i=1
397 // while (i≤10) {
398 //     console.log(num+"X"+i+'='+num*i);
399 //     i++;
400 // }
401
402
403
404
405
406
407
408 //?4.👀👀 do-While Loop👀👀
409
410 //syntax 🙋🙋🙋🙋
411 // do {
412
413     //code
414
415 // }while (condition)
416
417 //Example⇒ Print 1 to 10 number using do-while loop !? !? !?
418
419 // var i=1;
420 // do{
421 //     console.log(i);
422 //     i++;
423
424 // }while (i≤10)
425
426 //!Create a table of 19 using do-while loop

```



```

427
428 // var num=19
429 // var i=1
430 // do{
431 //     console.log(num+"X"+i+'='+num*i);
432 //     i++
433 // }while(i≤10)
434
435
436 //5.👀👀 For Loop👀👀
437
438 //syntax 🙋🙋🙋🙋
439 // for(initialization;condtion;iteration){
440 //     code
441 // }
442
443 //Example⇒ Print 1 to 10 number using For Loop !? !? !?
444
445 // for(var i=1;i≤10;i++){
446 //     console.log(i);
447
448 // }
449
450
451 //!Create a table of 29 using for loop
452 // var num=29
453 // for(let i=0;i≤10;i++){
454 //     console.log(num+"X"+i+'='+num*i);
455 // }
456
457
458 //!Create a infinit for loop
459 // for(;;){
460 //     console.log('Hello Adarsh');
461
462 // }
463
464
465
466 //!Practic : 🙋🙋🙋🙋
467 //!Calculate the sum of number from 1 to 10 using for loop
468
469 // var sumOfNumber=0;
470 // for(i=1;i≤10;i++){
471 //     sumOfNumber=sumOfNumber+i;
472
473 // }
474 // console.log(sumOfNumber);
475
476
477
478 //!Program to check if a year is a leap year
479

```

```

480 //condition for leap year  $\implies$  a year is divisible by 4 and 400 and not
divisible
481 // * by 100
482
483 // var year=2002
484
485 // if(year % 4 $\equiv$ 0){
486 //     if(year % 100  $\equiv$  0 ){
487
488 //         if(year % 400 $\equiv$ 0){
489 //             console.log(year,'is a leap year');
490
491 //         }
492 //         else{
493 //             console.log(year,'is not a leap year');
494
495 //         }
496 //     }
497 //     else{
498 //         console.log(year,'is a leap year');
499
500 //     }
501 // }
502 // else{
503 //     console.log(year,'is not a leap year');
504
505 // }
506
507
508
509 //!Draw a pattern
510
511 // *
512 // * *
513 // * * *
514 // * * * *
515 // * * * * *
516
517
518 // for(var i=1;i $\leq$ 5;i++){
519 //     var pattern="";
520 //     for(var j=1;j $\leq$ i;j++){
521 //         pattern=pattern+" *"
522 //     }
523 //     console.log(pattern);
524
525 // }
526
527
528
529
530
531
532

```

```

533          /* #. 🔄🔄Function In JavaScript 🔄🔄
534
535
536
537  //?1.Fucntion Decleration:- 📌📌📌
538  //?Declare the function using 'FUNCTION' keyword followed by the function name
539  //? parameter(if any) and the function bodey
540
541  //Example:- 📌📌
542
543  // function greet(){
544  //     console.log('hello i am a function');
545
546  // }
547
548
549  //?2.Fucntion Invocation (calling a function):- 📌
550
551  //? A function you can invoke or call it by using it's name followed by
552  //? parentheses if a function has parameter provide value(argument) for
553  //? those parameter inside the parantheses
554
555
556  //Example:- 📌📌
557
558  // greet()
559
560  //?Practice Time 🐼🐼🐼
561
562  //!write a function to find the sum of two number
563
564  // function sumOfTwoNumber(){
565  //     var a=5
566  //     var b=10
567  //     console.log(a+b);
568
569  // }
570  // sumOfTwoNumber()
571
572
573
574  //?3.Function Parameter 📌📌
575  //? It act as a placeholder for a value that will be provided when
576  //? the function is called
577
578  //Example:- 📌📌
579
580  // function greet(parameter1,parameter2,parameter13, ... ){
581  //     // code
582  // }
583  // greet()
584
585
586  //?4.Function Argument 📌📌

```

```

587
588 //? A function argument is a value that you provide when you call
589 //? a function. argument are passes into a function to fill parameter
590 //? defined in the function Decleration
591
592
593 //Example-: 🙋🙋
594 // greet(Argument1,Argument2,Argument3, ... )
595
596
597 //?Practice Time 🐼🐼🐼
598
599 //!write a function to find the sum of two number with parameter
600
601 // function sumOfTwoNumber(a,b){
602 //     console.log(a+b);
603
604 // }
605 // sumOfTwoNumber(78,89)
606
607
608 // var result=(function (a,b){
609 //     console.log(a+b)
610
611 // })(5,10);
612
613
614
615
616
617
618
619 //* #. 🐼🐼ECMAScript In JavaScript 🐼🐼
620
621 // ?. ECMAScript 2015/ES6
622
623 //todo 1.LET AND CONST
624 //todo 2.Template Strings
625 //todo 3.Default Arguments
626 //todo 4.Arrow Function
627 //todo 5.Destructuring
628 //todo 6.Object Properties
629 //todo 7.Rest Operators
630 //todo 8.Sperad Operators
631
632
633
634
635
636
637
638 //? 1.LET AND CONST
639
640 //*LET⇒It is used to declared variable with block-scoped.LET are Mutable

```

```

641  /*CONST⇒It is used to declared variable with block-scoped.CONST are immutable
642
643  //Example:-👉👉
644
645  // let myName='Adarsh';
646  // myName='vivek'
647  // console.log(myName);
648
649
650
651  // const Name='Adarsh';
652  // Name='vivek'
653  // console.log(Name);    //Assignment to constant variable. hence const are
654  immutable
655
656
657
658
659
660  /*how LET and Const are block-Scope Variable
661
662  //Example 2:-👉👉
663
664  // if(true){
665      let myName='Adarsh'
666      const mySurname='Rai'
667  // }
668
669  // console.log(myName);
670  // console.log(mySurname);    //we cant access the LET and CONST variable
671  outside the block scope
672
673
674
675
676  /*? 2.Template Strings
677  //In ES6,template string also known as template literals provide
678  // a convenient and flexible way to craete string in javascript.
679
680  //template string are enclsed in backticks (` `) rather than single or
681  // double quotes
682
683  //syntax:-👉👉
684
685  //`${}`
686
687
688  //Example:-👉👉
689
690  // let my_firstName='Adarsh';
691  // let my_lastName='Rai'
692

```

```
693 // let My_FullName=`${my_firstName} ${my_lastName}`
694 // console.log(My_FullName);
695
696
697
698 //Example 2:-👉👉
699
700 // const age=22;
701 // const message=`I am ${age} Years Old`
702 // console.log(message);
703
704
705
706
707 //? 3.Default Arguments
708 //default function parameter allow named parameter to be
709 // initialized with default value if no value or undefined or undefined is passed
710
711
712
713 //Example :-👉👉
714
715 //!Write a function to find sum of two number ?
716 //!what if during function call user only passed one arugumet
717
718
719
720 // function sum(a,b=20){ //here b set to the default parameter
721 //     return a+b
722 // }
723
724 // console.log(sum(10));
725
726
727
728
729
730
731 //? 4.Arrow Function
732
733 //Arrow function also known as fat arrow function were introduced
734 //as a concise way to write anonymous function.
735
736 //* Normal Function Expression
737
738 // const sum=function(a,b){
739 //     return a+b
740 // }
741 // console.log(sum(5,20));
742
743
744
745 //* convert Normal Function Expression to fat arrow function
746
```

```

747 // const sum=(a,b)⇒{
748 //     return a+b
749 // }
750 // console.log(sum(89,89));
751
752
753
754 // const sum=(a,b)⇒`the sum of two number is :⇒${a+b}`
755
756
757
758
759 //!Interview Questions 🍷🍷🍷🍷
760
761 /* 1. Write a javascript function calculator that take two number and operator
762 /*as parameter and return the result of two operator.the function support
763 /* addition,substraction,multiplication and division
764
765 // const calculator=(a,b,operator)⇒{
766 //     switch(operator){
767 //         case '+':
768 //             console.log(a+b);
769 //             break
770
771 //         case '-':
772 //             console.log(a-b);
773 //             break
774
775 //         case '*':
776 //             console.log(a*b);
777 //             break
778
779 //         case '/ ':
780 //             console.log(a/b);
781 //             break
782
783 //         default:
784 //             console.log('please enter valid operator');
785
786
787 //     }
788 // }
789
790 // calculator(400,4, '+')
791 // calculator(400,4, '-')
792 // calculator(400,4, '*')
793 // calculator(400,4, '/')
794
795
796
797
798
799 /*2.Write a Function to reserve a given string without using built-in
800 /* reverse methods

```

```

801
802 // const reverseString=(string)⇒{
803 //     debugger
804 //     let result="";
805 //     for(let i=string.length-1;i≥0;i--){
806 //         result=result+string[i]
807
808 //     }
809
810 //     return result
811
812 // }
813 // console.log(reverseString('Adarsh Rai'));
814
815
816
817 //*2.Write a Function to determine if a given string is a palinfrome
818 //* (read the same backward as forward)
819
820
821 // const findPalindrome=(string)⇒{
822 //     debugger
823 //     let result="";
824 //     for(let i=string.length-1;i≥0;i--){
825 //         result=result+string[i]
826
827 //     }
828
829 //     if(result===string){
830 //         return 'is a palindrome'
831 //     }
832 //     else{
833 //         return 'not a palindrome'
834 //     }
835
836 // }
837 // console.log(findPalindrome('oyo'));
838
839
840
841
842
843 //* #. 🐞🐞 JavaScript Arrays 🐞🐞
844
845
846 //*Array 🐞🐞🐞
847 //? ⇒JavaScript Array is an object that represent a collection of
848 //? similar type of elements
849
850 //? Each value will be called as an elements
851
852 //? In array each elements is represented by an inedx which start with zero
853
854

```



```

855 //Syntax 📌📌
856 //           0           1           2
857 // const person=['Ram','Adarsh','Hari']
858
859 //? We can access each element by using indexes
860
861 //person[0] ==>Ram
862 //person[1] ==>Adarsh
863
864
865
866 // const person=['Ram','Adarsh','Hari']
867 //           📌           📌
868 //           Lower index       Upper Index
869
870 /* First Elements or head ==>Refer to the element at index 0
871 /*Last Element or tail ==> refer to the element at the last which can be
obtained
872                                     /* using array.length-1
873
874 //?Es6 2022 also introduce new .at() method in array which help to
875 //? index from last elements too easily
876
877
878
879 // person[-1] ==>error
880 //person.at(-1) ==>Hari
881 //person.at(-2) ==> Adarsh
882
883
884
885
886
887 /* What we will cover 🎉🎉🎉
888
889 //todo 1. Creating Array / Accessing Element/Modifying Element
890 //todo 2. Array Traversal / Iterating over Array
891 //todo 3. How to insert , Add ,Replace and Delete Element in an Array(CURD)
892 //todo 4. Filter in an Array
893 //todo 5. Searching In Array
894 //todo 6. How to sort and complex an Array
895 //todo 7. Very Vary imp array Method
896
897
898 //? 1. Creating Array / Accessing Element/Modifying Element
899 //Array In javaScript can be created using the array constructor or with
900 //array literals (square bracket [])
901
902 /* Using Array Constructor
903
904 // let fruits=new Array('apple','orange','banana')
905 // console.log(fruits);
906
907

```

```

908  // *Using Array Literals
909
910  // let fruits=['apple','orange','banana']
911  // console.log(fruits);
912
913
914  // * How to create empty array
915
916  // let arr=[]
917  // console.log(arr);
918  // console.log(typeof arr);    // type of array is object
919
920
921
922
923  // * Accessing Elements
924
925  // Array elements are accessed using zero-based indices
926
927  // let fruits=['apple','orange','banana']
928  //           0         1         2
929
930  // console.log(fruits[0]);
931
932
933
934
935
936  // * Modifying Elements
937
938  // you can modify array elements by assigning new values to specific indices
939
940  // let fruits =['apple','orange','banana']
941  // fruits[1]='mango'
942  // console.log(fruits);
943
944
945
946
947  // ? 2. Array Traversal / Iterating over Array
948
949  // * for of loop also known as iterable
950
951  // the for of loop is used to iterate over the value of an iterable
952  // object such as array, string or other iterable object
953
954  // let fruits=['apple','orange','banana','mango','grapes']
955
956  // for(let items of fruits){
957  //     console.log(items);
958
959  // }
960
961

```

```

962
963  /* Using for loop
964
965  // let fruits=['apple','orange','banana','mango','grapes']
966  // for(let i=0;i<fruits.length;i++){
967  //     console.log(fruits[i]);
968  // }
969
970
971
972  /*for in loop
973
974  //the for in loop is used to iterate over the value properties
975  //(including indices) of an object
976
977
978  // let fruits=['apple','orange','banana','mango','grapes']
979  // for (let items in fruits){
980  //     console.log(items);
981  // }
982
983
984
985
986  /*forEach Method 🐼 🐼
987
988  //the arr.forEach() method call provided function once for
989  // each element of the array. the provided function may perform
990  //any kind of operation on the element of the given array
991
992
993  //?Syntax - forEach 👉 👉
994
995  // array.forEach((currElement,index,array)⇒{
996  //     your logic
997  // },thisValue)
998
999
1000
1001  // let fruits=['apple','orange','banana','mango','grapes']
1002  // fruits.forEach((currelem,index)⇒{
1003  //     console.log(`${currelem} ${index}`);
1004
1005  // })
1006
1007
1008
1009  // let fruits=['apple','orange','banana','mango','grapes']
1010  // const myforEach=fruits.forEach((currelem,index)⇒{
1011  //     return `${currelem} ${index}`
1012
1013  // })
1014  // console.log(myforEach);    //undefined    ⇒ we can't return in forEach method
1015

```

```

1016
1017
1018
1019 //* MAP Method 🧠🧠
1020
1021 //map create a new array from calling a function for every array
1022 //element. map() does not change the original array
1023
1024
1025 /// Syntax-: Map 🙋🙋🙋
1026
1027 // array.map((currElement,index,array)⇒{
1028 //     logic
1029 // })
1030
1031
1032 ///Example -: 🧠🧠
1033
1034 // let fruits=['apple','orange','banana','mango','grapes']
1035
1036 // fruits.map((curElem,index)⇒{
1037 //     console.log(`${curElem} ${index}`);
1038
1039 // })
1040
1041
1042
1043 // let fruits=['apple','orange','banana','mango','grapes']
1044
1045 // const myMaparr=fruits.map((curElem,index)⇒{
1046 //     return `${curElem} ${index}`
1047 // })
1048
1049 // console.log(myMaparr); //it create a new array
1050
1051
1052
1053
1054 ///Practice Time-: 🤖🤖
1055
1056 ///Write a program to multiply each element with 2
1057
1058 // const array=[1,2,3,4,5]
1059 // const multiplyElem=array.map((elem)⇒{
1060 //     return elem*2
1061 // })
1062
1063 // console.log(multiplyElem);
1064
1065
1066 ///Notes 🧠🧠
1067
1068 ///use forEach to perform action on each element
1069 ///use map to create a new array with transformed elements

```

```
1070
1071
1072
1073
1074 //? 3. How to insert , Add ,Replace and Delete Element in an Array(CURD)
1075
1076
1077 /*1. Push()-: Method that adds one or more element to the end of an Array
1078
1079 //Syntax:-👉👉👉
1080 // push(element_name)
1081
1082 //Example:-👉👉👉
1083
1084 // let fruits=['apple','orange','banana','mango','grapes']
1085 // fruits.push('pineapple')
1086 // console.log(fruits.push('pineapple'));
1087 // console.log(fruits);
1088
1089
1090
1091 /*2. Pop()-:Method that remove the last element from an array
1092
1093 //Syntax:-👉👉👉
1094 // pop(element_name) or pop()⇒if we don't put anything it remain remove alement
1095
1096 //Example:-👉👉👉
1097
1098 // let fruits=['apple','orange','banana','mango','grapes']
1099 // fruits.pop()
1100 // console.log(fruits.pop()); //mango⇒return which element is pop
1101 // console.log(fruits);
1102
1103
1104 /*3.unshift()-: Method that add one or more element to the beginning of an array
1105
1106 //Syntax:-👉👉👉
1107 // unshift(element_name)
1108
1109 //Example:-👉👉👉
1110
1111 // let fruits=['apple','orange','banana','mango','grapes']
1112 // fruits.unshift('pineapple')
1113 // console.log(fruits.unshift('guava'));
1114 // console.log(fruits);
1115
1116
1117 /*4.shift()-: method that remove the first element from an array
1118
1119 //Syntax:-👉👉👉
1120 // shift(element_name) or shift()
1121
1122 //Example:-👉👉👉
1123
```

```

1124 // let fruits=['apple','orange','banana','mango','grapes']
1125 // fruits.shift()
1126 // console.log(fruits.shift());
1127 // console.log(fruits);
1128
1129
1130
1131
1132 //What if want add or remove anywhere in an element
1133
1134 //?Splice()😬😬
1135
1136 //the splice() method of array instances change the content of an array
1137 // by removing or replacing existing element and/or adding new element in place
1138
1139 //Syntex:-👉👉
1140
1141 // splice(StaticRange,deleteCount,item1,item2,item3 ... itemN)
1142
1143 //Example:-👉👉
1144
1145 //let person=['Adarsh','Sita','Gita','Vivek','Ram']
1146 //console.log(person.splice()); // it return the empty array
1147 //console.log(person.splice(1)); //it start from index 1 and return ==>[
'Sita', 'Gita', 'Vivek', 'Ram' ]
1148 //console.log(person.splice(0)); // it start from index 0 and return [
'Adarsh', 'Sita', 'Gita', 'Vivek', 'Ram' ]
1149
1150 //console.log(person.splice(1,1)); //it return deleted element=> [ 'Sita' ]
1151
1152
1153 //?How to delete😬😬
1154 //person.splice(1, 1)
1155 // 👉 👉
1156 // start DeleteCount
1157 //console.log(person); // start from 1 and delete 1 element => [ 'Sita' ] and
return [ 'Adarsh', 'Gita', 'Vivek', 'Ram' ]
1158
1159
1160
1161 //?How to Add😬😬
1162
1163 //person.splice(2,1,'Hari','Narayan','Golu') //start from 2 deleted [ 'Gita' ]
and add 'Hari','Narayan','Golu'
1164 //console.log(person); //result is ['Adarsh','Sita','Hari',
Narayan','Golu','Vivek','Ram']
1165
1166
1167 //!What if you want to add the element at the end
1168
1169 // let person=['Adarsh','Sita','Gita','Vivek','Ram']
1170 // person.splice(person.length,0,'Vicky')
1171 // console.log(person);
1172

```

```

1173
1174
1175
1176 //? 5. Searching In Array
1177
1178 //for Searching we have indexOf,lastIndexOf and includes
1179
1180 //const numbers=[1,2,3,4,5,6,7,8,9]
1181
1182 /*1.indexOf Method :→ the index of method return the first index at
1183 /* which a given element can be found in the array or -1 if it is not found
1184
1185
1186 //syntax-: 📌📌
1187
1188 //1.indexOf(searchElement)
1189 //2.indexOf(searchElement,fromIndex)
1190
1191 //Example-: 📌📌
1192
1193 // console.log(numbers.indexOf(4)); //index of 4 is 3
1194 // console.log(numbers.indexOf(4, 5)); //return -1 because 4 is not
present after 5th index
1195 //
1196 //           📌       📌
           SearchElement  from_Index
1197
1198
1199 /**lastIndexOf():→ the lastIndexOf() method array return the last index at
which a
1200 /* given element can be found in the array , or -1 if it is not present.
1201 /* the array is searched backward, starting at fromIndex
1202
1203 // const numbers=[1,2,3,4,5,6,6,7,8,9,1]
1204 //           ←———— it sarch from backward
1205 // console.log(numbers.lastIndexOf(1)); // it reatun 10 because it return last
index
1206 // console.log(numbers.lastIndexOf(1,5)); //it sarch from 5 to 0 index and 1
is find at 0 index as it search backward
1207 //
1208
1209
1210
1211 /**Includes():→ The includes() method check whether an array includes a certain
elements
1212 /*returning true or false
1213
1214 //syantx:- 📌📌
1215
1216 //includes(searchElement)
1217 //includes(searchElem,fromIndex)
1218
1219
1220 //Example-: 📌📌
1221

```

```
1222 // const numbers =[1,2,3,4,5,6,7,8,9]
1223 // console.log(numbers.includes(5)); //return true because 5 is present at
    index 4
1224
1225 // console.log(numbers.includes(4,5)); //return false because 4 is not present
    after 5th index
1226
1227
1228
1229
1230
1231
1232 //todo Challenge time 🧠🧠😬😬
1233
1234 //!1. Add dec at the end of an array
1235
1236 // const months=['jan','feb','march','april','may','june','july',
    'aug','sep','oct','nov']
1237 // months.splice(months.length,0,'dec');
1238 // console.log(months); //addes dec 😎
1239
1240
1241 //!2.what is the return value of splice method
1242
1243 //an empty array ([])
1244
1245 //!3.update march to March
1246
1247 // const months=['jan','feb','march','april','may','june','july',
    'aug','sep','oct','nov']
1248
1249 // months.splice(2,1,'March')
1250 // console.log(months); //updated 😎😎
1251
1252
1253 //!4. delete june from an array
1254 // const months=['jan','feb','march','april','may','june','july',
    'aug','sep','oct','nov']
1255 // months.splice(5,1)
1256 // console.log(months); //june is deleted 😎😎
1257
1258
1259
1260
1261
1262 //? 4. Filter in an Array
1263
1264 //*1.Find() Method-:
1265
1266 // the find method is udes to find the first element in an array
1267 // that satisfies a provided testing function. it return the first matching
1268 // element or undefined if no element is found
1269
1270
```



```

1271
1272
1273 //Syntax-: 📌📌
1274
1275 // Array.find((currElem,index,array)⇒{
1276 //     logic
1277 // })
1278
1279
1280 //Example-: 📌📌
1281
1282 // const numbers=[1,2,3,4,5,4,5,6,7,8,4,1,2,3]
1283
1284 // const result=numbers.find((curElem)⇒{
1285 //     return curElem>5                //it start searching and if element is
found                                // then stop searching and give result
1286                                     // })
1287
1288
1289 // console.log(result);
1290
1291
1292
1293
1294 //*findIndex() Method:-
1295
1296 //it return the index of the first element that satisfies the provided testing
function
1297 //if no element satisfy the testing function -1 is returned
1298
1299
1300 //Syntax-: 📌📌
1301
1302 // Array.findIndex((currElem,index,array)⇒{
1303 //     logic
1304 // })
1305
1306
1307 //Example-: 📌📌
1308
1309 // const numbers=[1,2,3,4,5,4,5,6,7,8,4,1,2,3]
1310
1311 // const result=numbers.findIndex((curElem)⇒{
1312 //     return curElem>5
1313 // })
1314
1315 // console.log(result);    //7 it return the index of the result
1316
1317
1318
1319
1320 //*Filter Method():→ The filter method craete a new array with all elements that
pass
1321 //*test implemented by the provided function

```

```
1322
1323
1324 //Syntax:-👉👉
1325
1326 //1.filter(callbackFn)
1327 //2.filter(callbackFn,thisArg)
1328
1329
1330 //Example:-👉👉
1331
1332 // const numbers=[1,2,3,4,5,6,7,8,9]
1333
1334 // const result=numbers.filter((elem)⇒{
1335 //     return elem>5
1336 // })
1337
1338 // console.log(result);
1339
1340
1341
1342 //?UseCase:→In a E-commerce website when we want to remove or delete any product
1343 //? from addToCard page
1344
1345 //Example:-👉👉
1346
1347 //!lets say user want to delete value 6
1348
1349 // const numbers=[1,2,3,4,5,6,7,8,9,6]
1350
1351 // const deleteElement=numbers.filter((curElem)⇒{
1352 //     return curElem≠6
1353 // })
1354
1355 // console.log(deleteElement);
1356
1357
1358
1359 //! practice Time
1360
1361 //!1.Filtering Products by Price
1362
1363 // const products=[
1364 //     {name:'Leptop',price:1200},
1365 //     {name:'Phone',price:800},
1366 //     {name:'Tablet',price:500},
1367 //     {name:'SmartWatch',price:300},
1368 //     {name:'AirPod',price:150}
1369 // ]
1370
1371 //!Q1.filter product with a price less than or equal to 500
1372
1373 // const FilteredItem=products.filter((elem)⇒{
1374 //     return elem.price≤500
1375 // })
```

```

1376
1377 // console.log(FilteredItem);
1378
1379
1380
1381 //!Q2. Filter Unique Value
1382
1383 // const numbers=[1,2,3,4,5,6,5,7,8,9]
1384 // const unique=[]
1385 // const UniqueValue=numbers.filter((elem)⇒{
1386
1387 //     if(!unique.includes(elem)){
1388 //         unique.push(elem)
1389 //     }
1390 // })
1391
1392 // console.log(unique);
1393
1394
1395
1396
1397
1398
1399
1400 //? 6. How to sort and compare an Array
1401
1402 //*sorting an Array:→The sort method sort the element of an array
1403 //* in place and return the sorted array.by default it sort element as string
1404
1405
1406 // const fruits=['Banana','Apple','Orange','Mango']
1407 // fruits.sort()
1408 // console.log(fruits);
1409
1410 // const numbers=[1,2,13,8,4,5,1,2,4,6]
1411 // numbers.sort()
1412 // console.log(numbers);
1413
1414
1415
1416
1417
1418 //*Comparing the array
1419
1420 //syntax :- 🙋🙋
1421 // const sortedNumber=Number.sort((a,b)⇒{
1422 //     return a-b
1423 // })
1424
1425 // const numbers=[1,2,4,5,11,45,12,23]
1426 // numbers.sort((a,b)⇒{
1427 //     return a-b
1428 // })
1429 // console.log(numbers);

```

```
1430
1431
1432
1433 //? how to get descending order
1434
1435 // const numbers=[1,12,3,4,9,7,8,6]
1436 // numbers.sort((a,b)⇒{
1437 //     return b-a
1438 // })
1439
1440 // console.log(numbers);
1441
1442
1443
1444
1445
1446
1447 //? 7. Very Vary imp array Method
1448
1449 //*Map(): 🤔🤔
1450
1451 // ★ Create a new array from a calling a function for every array element
1452 // ★ does not execute the function for empty element
1453 // ★ does not change the original array
1454
1455 //!1.Using the map method write a function that takes an array of numbers and
return
1456 //! a array where each number is squared but only if its an even number
1457
1458 // const numbers=[1,2,3,4,5,6,7,8,9]
1459
1460 // const evenNumberSquare=numbers.map((elem)⇒{
1461 //     if(elem % 2 ===0){
1462 //         return elem*elem
1463 //     }
1464 // }).filter((elem)⇒{
1465 //     return elem !== undefined
1466 // })
1467
1468 // console.log(evenNumberSquare);
1469
1470
1471
1472
1473 //!2. using the map method write a function that takes an array of string and
return a new
1474 //! array where each string is captialized
1475
1476 // const words =['apple','banana','cherry','date']
1477
1478 // const captializedWords=words.map((elem)⇒{
1479 //     return elem.toUpperCase()
1480 // })
1481 // console.log(captializedWords);
```

```
1482
1483
1484
1485 //!3. using the map method write a function that takes an array
1486 //! of name and return a new array each name s prefixed with 'Mr'
1487
1488 // const Names=['Adarsh','Ram','Hari','Shyam']
1489 // const AddMr=Names.map((elem)⇒{
1490 //     return `Mr ${elem}`
1491 // })
1492 // console.log(AddMr);
1493
1494
1495
1496
1497
1498
1499
1500 /*Reduce Method(): 📌📌
1501
1502 //The reduce method in javascript is used to accumulate or reduce
1503 // an array to a single value. it iterate over the element of array
1504 // and applies a callback function to each element updating an accumulator value
1505 //with the result. the reduce method takes a callback as it's first argument and
1506 // an optional initialVlaue for the accumulator as the second argument
1507
1508 //Synatx 📌📌
1509
1510 // Array.reduce((accumulator,currentValue,index,array)⇒{
1511 //     logic
1512 // },initialValue)
1513
1514
1515
1516 /*BreakDown
1517
1518 //1.Callback :- a function that is called once for each element in tha array
1519 //2.accumulator :- the accumulated result of the prvious iterations
1520 //3.currentValue :- the currentValue element being processed in the array
1521 //4. index :- the index of the curElem
1522 //5. array :- the original array
1523 //6. initialValue :- An initialValue for the accumulator if not provided the
first element
1524
// of the array is used as the initial accumulator
1525
1526
1527
1528
1529 //!Q.1 Write a javaScript function that calculate the total price of item
1530 //!in a shopping cart the function should take an array of item price an input
1531 //! and return the total price
1532
1533
1534 // const productsPrice=[100,200,5,7,89,45,12,13]
```

```

1535
1536 // const totalPrice=productsPrice.reduce(function(accu,currElem,index,array){
1537 //     return accu+currElem
1538 // })
1539
1540 // console.log(totalPrice);
1541
1542
1543
1544
1545
1546 //*Algorithm of reduce method
1547
1548 //const productsPrice=[100, 200,5,7,89,45,12,13]
1549 //
1550 //           📌      📌
1551 //           acc    currElem
1552
1553 //!1. First Iteration 📌 📌
1554
1555 //accu+currElem ⇒ 100+200=300
1556
1557 //const productsPrice=[300, 5,7,89,45,12,13]
1558 //
1559 //           📌      📌
1560 //           acc    currElem
1561
1562
1563 //!1. Second Iteration 📌 📌
1564
1565 //accu+currElem ⇒ 300+5=305
1566
1567 //const productsPrice=[305, 7,89,45,12,13]
1568 //
1569 //           📌      📌
1570 //           acc    currElem
1571
1572
1573 //!1. Third Iteration 📌 📌
1574
1575 //accu+currElem ⇒ 305+7
1576
1577 //const productsPrice=[312,89,45,12,13]
1578 //
1579 //           📌      📌
1580 //           acc    currElem
1581
1582
1583
1584
1585
1586 //!1. Forth Iteration 📌 📌
1587
1588 //accu+currElem ⇒ 312+89

```

```

1589
1590 //const productsPrice=[401,45,12,13]
1591 //
1592 //          acc    currElem
1593
1594
1595
1596
1597 //!1. fifth Iteration
1598
1599 //accu+currElem ⇒ 401+45
1600
1601 //const productsPrice=[446,12,13]
1602 //
1603 //          acc    currElem
1604
1605
1606
1607 //!1. sixth Iteration
1608
1609 //accu+currElem ⇒ 446+12
1610
1611 //const productsPrice=[458,13]
1612 //
1613 //          acc    currElem
1614
1615
1616
1617
1618 //!1. seventh Iteration
1619
1620 //accu+currElem ⇒ 458+13=471
1621
1622 //const productsPrice=471
1623
1624
1625
1626
1627
1628
1629
1630 //* #. JavaScript Strings
1631
1632
1633 //*What we will cover
1634
1635 //todo 1. String and it's properties
1636 //todo 2. Escape Character
1637 //todo 3. String Search Method
1638 //todo 4. Extracting String Parts
1639 //todo 5. Extracting String Characters
1640 //todo 6. Replacing String Content
1641 //todo 7. Other Useful Methods
1642

```

```

1643
1644
1645
1646
1647
1648 //? 1. String and it's properties
1649
1650 /*String in javaScript
1651
1652 //string in javascript are a fundamental data type that
1653 // represent a sequence of characters
1654
1655
1656 //!Notes:-
1657
1658 //!string created with single or double quotes work the same
1659 //! there is no different between the single and double quotes
1660
1661
1662
1663 /*String Properties
1664
1665 //1.length:→property that return the length of the string(number of characters)
1666
1667 // const str='hello world'
1668 // console.log(str.length);    //including spaces
1669
1670
1671
1672
1673
1674 //? 2. Escape Character
1675
1676 //escape characters:→In javaScript backslash (\) is used as an
1677 //escape characters.it allow you to include special characters in a
1678 //string
1679
1680
1681 //      code      Result      Description
1682 //      \ '        '          single quotes
1683 //      \ "        "          double quotes
1684 //      \\         \          backslash
1685
1686
1687
1688 //Example: 🙋🙋
1689
1690 // let text="My name is "Adarsh Rai" "
1691 // console.log(text);    //we can't print double quotes  it give syntax error
1692
1693
1694
1695 //solution 1:→ put double quotes inside single quotes or vise varsha 🙋🙋
1696

```



```
1697 // const text ='My name is "Adarsh Rai "'
1698 // console.log(text);
1699
1700
1701 //solution 2:→use backslash 📌📌
1702
1703 // const text ="My name is \"Adarsh Rai\""
1704 // console.log(text);
1705
1706
1707 //*if we want a new line we use \n
1708
1709 // const text ="my name is \"Adarsh Rai\" \n i am a full stack developer"
1710 // console.log(text);
1711
1712
1713
1714
1715
1716 //? 3. String Search Method
1717
1718
1719 //*1.IndexOf():→ The indexOf() method return index of the
1720 //* first occurrence of a string in a string or it return -1 if
1721 //* the string is not found
1722
1723
1724 //Syntex:- 📌📌
1725 //1.indexOf(SearchString)
1726 //2.indexOf(SearchString,Position)
1727
1728
1729 //Exmaple:- 📌📌
1730
1731 // let text='Adarsh Rai'
1732 // console.log(text.indexOf('rai')); //it is case-sensitive
1733
1734
1735
1736 // let text='Adarsh Rai'
1737 // console.log(text.indexOf('Rai')); // return 7 ⇒ look only the 1st letter
1738
1739
1740 // let text='Adarsh Rai'
1741 // console.log(text.indexOf('a',7)); //return 8 ⇒ after 7 index 'a' index is 8
    because
1742 //it search from 7th index
1743
1744
1745
1746
1747
1748 //*2.lastIndexOf():-the lastIndex() method return the index of the last
1749 //* occurrence of a specified text in a String
```

```
1750
1751
1752 //Syntax:-👉👉
1753 //1.lastIndexOf(searchElem)
1754 //2.lastIndexOf(searchElem,position)
1755
1756
1757 //Example:-👉👉
1758
1759 // let text ='Hello JavaScript,welcome to our world best JavaScript Course'
1760 // console.log(text.lastIndexOf('JavaScript')); // return 43 ⇒ it search
backward and 2nd
1761                                     // 'JavaScript' find in last and then
it return last 'JavaScript' index
1762
1763
1764 // let text ='Hello JavaScript,welcome to our world best JavaScript Course'
1765 // console.log(text.lastIndexOf('our',5)); //return -1 it start searching
backward and it search from 5-0
1766                                     // and 'our' is not find from 5-0 so it
return -1
1767
1768
1769
1770 //*3.Search():→The Search() method search a string for a string(or regular
expression)
1771 //* and return the position of the match
1772
1773 //*return the index number where the first match is found return -1
1774 //* if no match found
1775
1776 //*it is case-sensitive
1777
1778
1779 //Example:-👉👉
1780
1781 // let text ='Hello JavaScript,welcome to our world best JavaScript Course';
1782 //console.log(text.search('JavaScript')); // return 6
1783
1784 //*In search we use Regular Expression to avoid case-sensitive
1785
1786 // console.log(text.search(/javascript/i)); // return 6 → i is a regular
expression to avoid case-sensitive
1787
1788
1789
1790 //!Important Tips 👉👉
1791
1792 //!1.The search() method can't take second start position argument
1793
1794 //search(searchElem,fromIndex) ❌❌
1795
1796 //!the indexOf() method cannot take powerful search value (regular Expression)
1797
```

```

1798 //!they accept same argument(parameter) and return the same value
1799
1800
1801
1802
1803 //*4.match():→return an array of the match value or null
1804 //* if no match is found also a case-sensitive
1805
1806
1807 // let text = 'Hello JavaScript,welcome to our world best JavaScript Course';
1808 // console.log(text.match('javascript')); //return null if string is not found
1809
1810
1811 // console.log(text.match('JavaScript'));
1812
1813 //it return 👉👉 full details in array formate
1814
1815 //! [
1816 //!     'JavaScript',
1817 //!     index: 6,
1818 //!     input: 'Hello JavaScript,welcome to our world best JavaScript Course',
1819 //!     groups: undefined
1820 //! ]
1821
1822
1823
1824
1825 //!Tips:here the JS convert the normal text into regular expression
text.match(/javascript/)
1826 //!without the g flag
1827
1828 // console.log(text.match(/JavaScript/g)); //it return ⇒ [ 'JavaScript',
'JavaScript' ]
1829
1830 // console.log(text.match(/javascript/ig)); // it avoid case-sensitive if we use
regular expression
1831
1832
1833
1834
1835 //*5.matchAll():→ return an iterator of all the match's providing detailed
1836 //*information about each match return an empty iterator if no match is found
1837
1838
1839 // let text = 'Hello JavaScript,welcome to our world best JavaScript Course';
1840 // console.log(text.matchAll('javascript')); // if no match found it return ⇒
Object [RegExp String Iterator] {}
1841
1842 // console.log(text.matchAll('JavaScript')); //also return same ⇒ Object
[RegExp String Iterator] {}
1843
1844 //!Tips:here the JS convert the normal text into regular expression
text.match(/javascript/)
1845 //!also add the the g flag at the end

```

```
1846
1847 // let matchAll=text.matchAll('JavaScript')
1848 // console.log(...matchAll); //after this it give the result
1849
1850
1851
1852
1853 //*6.includes():→return true if the string contain the specified value
1854 //* and false otherwise case-sensitive
1855
1856 // let text ='Hello JavaScript,welcome to our world best JavaScript Course';
1857 // console.log(text.includes('Java')); //return true beacuse it match in
1858 // 'JavaScript'
1859
1860 // console.log(text.includes('Javac')); //return false because it not match
1861
1862 //*7.StartsWith():→the startWith() method return true if string begins with
1863 //* a specified value otherwise it return false
1864
1865
1866
1867 // let text ='Hello JavaScript,welcome to our world best JavaScript Course';
1868 // console.log(text.startsWith('JavaScript')); //return false because string
1869 // start with "Hello"
1870
1871 // console.log(text.startsWith('hello')); // rteurn false it is a case-sensitive
1872
1873 // console.log(text.startsWith('Hello')); //return true because string start
1874 // with 'Hello'
1875
1876
1877 //*endsWith():→ the endWith() method return true if a string end with a
1878 //*specified value otherwise it return false
1879
1880 // let text ='Hello JavaScript,welcome to our world best JavaScript Course';
1881 // console.log(text.endsWith('Hello')); //return false beacuse string end with
1882 // Course not Hello
1883
1884 // console.log(text.endsWith('course')); //return false because it is case-
1885 // sensitive
1886
1887 // console.log(text.endsWith('Course')); //return true because string end with
1888 // Course
1889
1890
1891 //? 4. Extracting String Parts
1892
1893 //*1.Substr():→ it is deprecated
```

```
1894
1895 //*2.Slice():→ extract a part of a string and return the extracted part in
1896 //* a new string.but not including last index
1897
1898 //syntax:-👉👉
1899
1900 //slice(start,end)
1901
1902
1903 //example:👉👉
1904
1905 // let text='Adarsh'
1906 // console.log(text.slice(1,4)); //return dar start from 1st index and end 3rd
index not include 4th index
1907
1908 // console.log(text.slice(1)); //return darsh start with 1st and include all
till end
1909
1910
1911
1912
1913 //*Substring():→ extract a portion of the string based on starting and ending
indices.
1914 //*it also not include last index
1915
1916
1917 //syntax:-👉👉
1918 //1.substring(index)
1919 //2.substring(startIndex,endIndex)
1920
1921
1922 //!Notes:-
1923 //substring () is similar to slice() the difference is that start and end
1924 // values less than 0 are treated as 0 in substring ()
1925
1926 // let text ='Adrash Rai'
1927 // console.log(text.substring(-1)); //return Adrash Rai it treat 0 and print
all data from 0 to end index
1928
1929
1930 // console.log(text.substring(1,5)); //return dras start from 1 and end at 4th
index
1931
1932
1933 //!similarities:-both slice and substring exclude the endIndex
1934
1935
1936
1937
1938 //!Interview Question :-👉👉👀👀
1939
1940 //!1.What is the output for the following code?
1941
1942 // let text='Hello JavaScript,welcome to our world best JavaScript Course';
```

```
1943 // console.log(text.slice(1)); // start from 1st index and go to end
1944
1945 // console.log(text.substring(1)); //same output
1946
1947
1948
1949
1950
1951
1952 //? 5. Extracting String Characters
1953
1954 //there are 3 method for extracting string characters
1955
1956 //1.The chatAt(position)
1957 //2.the charCodeAt(position)
1958 //3.the at(position)
1959
1960
1961
1962
1963 //*1.The chatAt(position)
1964
1965 //the charAt() method return the character at a specified index(position)
1966 //in a string if not it give empty string
1967
1968 // let text='Adarsh Rai'
1969 // console.log(text.charAt(5)); //return h because it present at 5th index
1970
1971
1972 // console.log(text.charAt(20)); //no char found at 20th index so it give empty
    string
1973
1974
1975
1976
1977
1978
1979
1980 //*1.The chatCodeAt(position)
1981
1982 //the charCodeAt() method return the code of the character at a specified index
1983 //in a string. the method return a UTF-16 code (an integer between 0 and 65535)
1984
1985
1986 // let text ='Adarsh Rai'
1987 // console.log(text.charCodeAt(5)); //return 104 h is present at 5th index and
    it's charCode is 104
1988
1989
1990
1991
1992 //*3.the at(position)
1993
1994 //the at() method return the character at a specified index in a string.
```

```
1995 //the at() method return the same as charAt()
1996
1997 // let text='Adarsh Rai'
1998 // console.log(text.at(5)); //return h because h is present at 5th index
1999
2000 //!it also take negative value
2001 // console.log(text.at(-2)); //at -2 index a is present
2002
2003
2004 //!Note
2005 //1.the at() method is a new addition to javascript
2006 //2.it allow the use of nagative index while charAt() do not
2007 //3.it introduce in ES2022
2008
2009
2010
2011 //? 6. Replacing String Content
2012
2013 //*1.replace():→the method is used to replace a specified value with another
value
2014
2015 // const text='Adarsh Rai'
2016 // console.log(text.replace('Adarsh','Vivek')); //Adarsh replaced with Vivek
2017
2018
2019 //!case-insensitive Replacement : to perform a case-insensitive replacement you
can use the
2020 //! i falg in the regular expression
2021
2022 // let text='Adarsh Rai'
2023 // console.log(text.replace(/adarsh/i,'Vivek')); //it avoid case-sensitive and
replace adarsh with Vivek
2024
2025
2026
2027 //*2.replaceAll():- the method is used to replace all match value with another
value
2028
2029 let text='Hello Adarsh Rai,Good Morning Adarsh'
2030 // console.log(text.replaceAll(text.replaceAll('Adarsh','Vivek')));
2031
2032
2033
2034 //!To avoid case-sensitive use regular expression
2035
2036 // console.log(text.replaceAll(/adarsh/gi,'Vivek')); //g falg for change all
match value
2037
2038
2039
2040 //? 7. Other Useful Methods
2041
2042 //*1.toUpperCase and toLowerCase:→ convert the string to uppcase or lowercase
2043
```

```
2044 // const str='javaScript'
2045 // console.log(str.toLowerCase()); //return ⇒ javascript convert string into
lowerCase
2046 // console.log(str.toUpperCase()); //convert return ⇒JAVASCRIPT string into
uppercase
2047
2048
2049
2050 /*2.trim:→ remove whitespace from both end of the string
2051
2052 // const str='    hello    '
2053 // console.log(str.trim()); //remove white space from both end
2054
2055
2056
2057
2058 /*3.split():→ split the string an array or substring based on a specified
delimiter
2059
2060 // const str='apple,orange,banana'
2061 // console.log(str.split(",")); //return [ 'apple', 'orange', 'banana' ] split
based on ","
2062
2063 // console.log(str.split("")); //split based on empty string
2064
2065
2066 /*to reverse the array
2067
2068 // console.log(str.split(",").reverse()); //reverse the array [ 'banana',
'orange', 'apple' ]
2069
2070
2071 /*to get back into the original string
2072
2073 // console.log(str.split(",").join()); //use join to get vack into the original
string apple,orange,banana
2074
2075
2076
2077
2078
2079
2080
2081 //!InterView Question 🙋🙋🙋👀👀
2082
2083 //!1.write a javascript function that print the letter 'a' through 'z' in the
console
2084 //!you should use a loop to iterate through the letter and print each one on a
new line
2085
2086
2087 // const charCodeAt=(()=>{
2088 //     for(i=97;i≤122;i++){
2089 //         console.log(String.fromCharCode(i));
2090 //         fromCharCode() used to convert back to the string in this
```



```

2091 //      we want to pass integer value
2092 //      }
2093 // }
2094 // charCodeAt()
2095
2096
2097
2098
2099 //!2.write a function to count the number of vowels in a string
2100
2101 // const countVowels=(str)⇒{
2102 //     const vowels='aeiou'
2103 //     let count =0
2104 //     for(let char of str){
2105 //         if(vowels.includes(char)){
2106 //             count++
2107 //         }
2108 //     }
2109
2110 //     return count
2111
2112 // }
2113 // console.log(countVowels('hello a e i o u Morning'));
2114
2115
2116
2117
2118
2119 //!3.write a function to check if all the vowels presnet in a string or not
2120
2121 // const checkVowelsPresent=(str)⇒{
2122 //     const vowels='aeiou'
2123 //     for(let char of vowels){
2124 //         if(!str.includes(char)){
2125 //             return false
2126 //         }
2127 //     }
2128 //     return true
2129 // }
2130
2131 // console.log(checkVowelsPresent('hello e i o'));
2132
2133
2134
2135
2136
2137 //! write a javascript function ispangram that takes a string as input and return
true
2138 //! if the string is a panfram(contain all letter of the alphabet) and false
otherwise
2139 //!the function should be case insensitive and ignore spacse
2140
2141
2142 //*pangram:→ a string contain A TO Z alphabet

```

```

2143
2144 // const ispangram=(str)⇒{
2145 // let alphabet=''
2146 // for(let i=97;i≤122;i++){
2147 //     alphabet=alphabet+String.fromCharCode(i)
2148 // }
2149 //     for(let char of alphabet){
2150 //         if(!str.includes(char)){
2151 //             return false
2152 //         }
2153 //     }
2154 //     return true
2155 // }
2156
2157 // console.log(ispangram('the quick brown fox jumps over the lazy dog'));
2158
2159
2160
2161
2162
2163 //* #. 🧐🧐 JavaScript Math Object 🧐🧐
2164
2165
2166 //*Math:→the math object contain static properties and methods
2167 //* for mathematical constant and function
2168
2169 //*Math work with the number type it doesn't work with BigInt
2170
2171
2172 //?1.Constants
2173
2174 //*i Math.PI:→represent the mathematical constant PI
2175
2176 // console.log(Math.PI); //return 3.141592653589793
2177
2178 //?Basic Operation
2179
2180 //*ii Math.abs:→ the math.abs() return the absolute value of a number
2181 //* how far the number is from 0
2182
2183 // console.log(Math.abs(5)); //return 5 always return positive value
2184
2185 // console.log(Math.abs(-5)); //convert negative number to positive value
2186
2187
2188 //*iii Math.round(x):→round a number to the nearest Integer
2189
2190 // console.log(Math.round(4.9));
2191 // console.log(Math.round(4.5));
2192
2193
2194 //*Vi Math.floor(x):→return the value of a x rounded down to its
2195 //*nearest integer
2196

```

```
2197 // console.log(Math.floor(3.7)); //convert to the down nearest value
2198
2199
2200
2201 /*Math.ceil(x):→return the value of x rounded up to its nearest value
2202
2203 // console.log(Math.ceil(3.7)); //convert up its nearest value
2204
2205
2206 /*Math.trunc(x):→ return integer party of x
2207
2208 // console.log(Math.trunc(3.7)); //return 3 because it is a integer part
2209
2210
2211
2212
2213
2214 //!Difference between floor and trunc
2215 // console.log(Math.floor(-3.7)); //it return rounded down value
2216 // console.log(Math.trunc(-3.7)); //it return integer part
2217
2218
2219
2220
2221
2222 /*Math.min(x,y):→ return min value
2223
2224 // console.log(Math.min(5,10)); //return 5
2225
2226
2227
2228 /*Math.max(x,y):→ return max value
2229
2230 // console.log(Math.min(5,10)); //return 10
2231
2232
2233
2234 //?Exponential and logarithmic function
2235
2236 /*i.Math.pow(x,y):-return the value of x to the power of y
2237
2238 // console.log(Math.pow(4,2)); //4 square is 16
2239
2240
2241
2242 /*ii.Math.sqrt(x):→ return the square root of x
2243
2244 // console.log(Math.sqrt(4)); //square root of 4 is 2
2245
2246
2247 /*Math.log(x):→ return the natural logarithm of x
2248
2249 // console.log(Math.log(1)); //return 0
2250 // console.log(Math.log(2)); //return 0.6931471805599453
```

```

2251
2252
2253 //*iii. Math.log2(x):→ return the base 2 logarithm of x
2254
2255 // console.log(Math.log2(5));
2256
2257
2258
2259
2260
2261 //*Math.random():→Math.random() return a random number
2262 //* between 0(inclusive) and 1(exclusive)
2263
2264 // console.log(Math.random()); //return 0.6680246362533855 its a random number
2265
2266
2267 // console.log(Math.round(Math.random()*10)); //now return the number between 1
to 10
2268
2269
2270 // console.log((Math.random()*100).toFixed(3)); // only return 3 value after
decimal
2271
2272
2273
2274
2275
2276
2277
2278
2279 //* #. 🐼🐼 Window In JavaScript 🐼🐼
2280
2281 //?.Window Object :=> the window object represent the global window in a browser
2282 //? Both the (BOM) and (DOM) are part of the window object
2283
2284
2285 //? BOM(Browser object model) :=>the BOM represent the browser as an object and
2286 //? provide method and properties form interacting with browser itself
2287 //? (not directly related to the content of a web page)
2288
2289
2290 //Example of BOM features include window.navigator for browser info
2291 // window.location for URL manipulation and window.alert for displaying alert
2292
2293
2294
2295 //? DOM(Document object Model):=> the dom represent the structured document
2296 //? as a tree of object where each object corresponds to a part of the
2297 //? document(such as element , attributes and text)
2298
2299 //Dow allow javascript to intreact with and manipulate the HTML element
2300
2301
2302 //! Summery ⇒ so while the DOM is focused on the content of the page the

```

```

2303 //! BOM is focused on the browser environment. the winode object serve as a
2304 //! global object
2305
2306
2307
2308
2309
2310 /* Window Object
2311 //<button onclick="window.open('https://hacklikeberlin.netlify.app/', '_blank')">
2312 //navigate
2313 //</button>
2314
2315 //👉👉 here if we click navigate button we go the the given website
2316
2317
2318
2319
2320 /*Window History Object
2321
2322 //1.<button onclick="history.back()">Go Back</button> ==> go back to the backward
privious website
2323
2324 //2. <button onclick="history.forward()">Go Forward</button> ==> go forward to
the new open website
2325
2326 //3. <button onclick="history.go(-1)">Move Backwoard or forward</button> ==>
move any forward or backward
2327
2328
2329
2330
2331
2332 /*Location Object
2333
2334 //1.<button onclick="location.reload()">Reload The Page</button> ==> here we use
to reload the page(refresh page)
2335
2336 //2. <button
onclick="location.replace('https://hacklikeberlin.netlify.app/')">Nagivate Page
using replace</button> ==>here we go to the given URL but
2337
// problem is that we can't go back or forward (← → this is disable)
2338
2339 //3.<button
onclick="location.assign('https://hacklikeberlin.netlify.app/')">Navigate page
using Assign</button> ==> here we go the given URL but benifit is that we can
go privious website (← → this is enable)
2340
2341
2342
2343
2344
2345
2346
2347

```

```
2348  /* DOM Navigation
2349  //go to 👉👉👉👉 \JavaScript_Full_course\window
      object\bom\DOM\DomNavigation.html
2350
2351  /*Dom  Searching
2352
2353  //? getElementById(id): Find an element by its ID.
2354
2355  //? getElementsByClassName(className): Find elements with a specific class name.
2356
2357  //? getElementsByTagName(tagName): Find elements with a specific tag name.
2358
2359  //? querySelector(selector): Find the first element that matches the specified
      CSS selector.
2360
2361  //? querySelectorAll(selector): Find all elements that match the specified CSS
      selector.
2362
2363
2364  //go to 👉👉👉 \JavaScript_Full_course\window object\bom\DOM\DomSearching.html
2365
2366
2367
2368  /* =====
2369  /* DOM - CRUD (Create, Read, Update, Delete):
2370  /* =====
2371
2372  //? createElement(tagName): Create a new HTML element.
2373
2374  //? appendChild(node): Append a node as the last child of a parent node.
2375
2376  //? removeChild(node): Remove a child node from its parent.
2377
2378  //? addEventListener(event, function): Create an event listener to handle events.
2379
2380  //? removeEventListener(event, function): Remove an event listener.
2381
2382  //? setAttribute(name, value): Set the value of an attribute on an element.
2383
2384  //? getAttribute(name): Get the value of a specific attribute on an element.
2385
2386  //? innerHTML: Read or update the HTML content of an element.
2387
2388  //? textContent: Read or update the text content of an element.
2389
2390
2391
2392  //go to 👉👉👉 \JavaScript_Full_course\window object\bom\DOM\DOMCURD.html
2393
2394
2395
2396
2397
2398
```

```
2399
2400
2401
2402
2403          /* #. 🐞🐞 Events In JavaScript 🐞🐞
2404
2405 //todo 1. 3 ways of writing Events in JS
2406 //todo 2. What is Event object
2407 //todo 3.MouseEvent in JavaScript
2408 //todo 4.KeyBoardEvent in JavaScript
2409 //todo 5.InputEvent in JavaScript
2410
2411
2412 // go to 🖱️🖱️ \JavaScript_Full_course\Events
2413
2414
2415          /* #. 🐞🐞 Local Storage In JavaScript 🐞🐞
2416
2417 //? it allow you to save key/value parirs in the browser
2418 //? stores data with no expiration data
2419 //? the data is not deleted when the browser is closed and are available
2420 //? for future sessions
2421
2422
2423
2424 //? what we will cover
2425
2426 //todo how to add/store data in localStorage
2427 //todo how to get/retrieve data in localStorage
2428 //todo how to remove data in localStorage
2429
2430
2431 //? how to add/store data in localStorage
2432 // localStorage.setItem('name','adarsh') //name is a key and adarsh is a value
2433
2434
2435 //? how to get/retrieve data in localStorage
2436 // localStorage.getItem('name') return adarsh if key doesn't exist it give null
value
2437
2438 //? how to remove data in localStorage
2439 // localStorage.removeItem('name')
2440
2441
2442 //! to check data go to browser and inspect then go to application >
localStorage >Chrome://new-tab
2443
2444
2445
2446 //todo Note:=>local storage can only store string, so when you want to store
2447 //todo a complex data structure like an array or an object you need to convert it
to a string
2448 //todo using JSON.stringify()
2449
```

```
2450
2451
2452
2453
2454
2455 //*1.JSON.Stringify()
2456 //? Convert A JavaScript object into a JSON string
2457 //? useful when you want to send data to a server or store it in a text file
2458 //? as JSON is a common data interchange format
2459
2460
2461
2462 //Example: 🙌🙌
2463
2464 // const data={name:'Adarsh',age:21,city:'thane'}
2465
2466 // const JsonString=JSON.stringify(data)
2467 // console.log(JsonString); //now convert data into string
2468
2469
2470 //*1.JSON.Parse()
2471 //? convert a JSON string into a JavaScript object
2472
2473 // const ParseData=JSON.parse(JsonString)
2474 // console.log(ParseData); //convert back to the object
2475
2476 //!check the project in JavaScript_Full_course\LocalStorage and Json Project
2477
2478
2479 //* #. 🤔🤔 Date and Time Object 🤔🤔
2480
2481 //?The Date() constructor create Date object when called as function,it
2482 //? return a string representing the current time
2483
2484
2485 //synatx 🙌🙌
2486 //1.new Date()
2487 //2.new Date(date string)
2488
2489
2490
2491 //? 9 ways to create a new Date objects
2492
2493 //1.new Date()
2494 //2.new Date(date string)
2495 //3.new Date(year,month)
2496 //4.new Date(year,month,day)
2497 //5.new Date(year,month,day,hours)
2498 //6.new Date(year,month,day,hours,minutes)
2499 //7.new Date(year,month,day,hours,minutes,seconds)
2500 //8.new Date(year,month,day,hours,minutes,seconds,millisecond)
2501 //9.new Date(millisecond)
2502
2503
```



```
2504
2505
2506 // *1. new Date() ⇒ create a date object representing the current date and time
2507
2508 // const currDate = new Date()
2509 // console.log(currDate); // return 2025-04-29T06:36:38.239Z
2510
2511 // 👉👉👉👉 this is a ISO 8081 format which is standard for representing date
and time
2512 // in this format the date and time are represented together separated by the
letter
2513 // 'T' and 'Z' at the end indicate that the time is in UTC (coordinated universal
time)
2514
2515 // !if we run in browser it return
2516 // Tue Apr 29 2025 12:10:28 GMT+0530 (India Standard Time)
2517
2518
2519 // *2. new Date(date string)
2520 // const dateString = new Date('2025-04-25T11:25:43.136Z')
2521 // console.log(dateString); return 2025-04-25T11:25:43.136Z
2522
2523
2524
2525 // *3. new Date(year, month)
2526 // const date1 = new Date(2025, 4)
2527 // console.log(date1);
2528
2529 // *4. new Date(year, month, day)
2530 // const date2 = new Date(2024, 1, 19)
2531 // console.log(date2);
2532
2533
2534 // *5. new Date(year, month, day, hours)
2535 // const date3 = new Date(2024, 1, 19, 10)
2536 // console.log(date3);
2537
2538
2539 // *6. new Date(year, month, day, hours, minutes)
2540 // const date4 = new Date(2024, 1, 19, 10, 44)
2541 // console.log(date4);
2542
2543 // *7. new Date(year, month, day, hours, minutes, seconds)
2544 // const date5 = new Date(2024, 1, 19, 10, 44, 30)
2545 // console.log(date5);
2546
2547 // *8. new Date(year, month, day, hours, minutes, seconds, millisecond)
2548 // const date6 = new Date(2024, 1, 19, 10, 44, 30, 500)
2549 // console.log(date6);
2550
2551
2552 // *9. new Date(millisecond)
2553 // const dateFromMillisecond = new Date(1745583148462)
2554 // console.log(dateFromMillisecond);
```

```
2555
2556
2557 //! Note 🖥️📱📺
2558 //! javascript count month 0 to 11 [jan=0,feb=1..] and [sun=0 mon=1...]
2559 //! javascript store data as millisecond: javascript store data as number of
2560 //!millisecond since jan 01 1970
2561
2562
2563 //? date string formate : if the datestring is in a recognizable formate
2564 //? the date object will be created accordingly
2565
2566 // const date1=new Date('2024-01-05')
2567 // const date2=new Date('january 5 ,2024')
2568
2569 // console.log(date1);
2570 // console.log(date2); //both give correct data
2571
2572
2573
2574
2575
2576 //*JavaScript Get Date Methods 📌📌
2577
2578 //?1.GetFullYear 📌📌
2579 // const currDate=new Date()
2580 // const year=currDate.getFullYear()
2581 // console.log(year); //return 2025
2582
2583
2584 //?2.GetMonth 📌📌
2585 // const month=currDate.getMonth()
2586 // console.log(month);
2587 // console.log(currDate); //return 3 {april index is 3}
2588
2589
2590
2591 //?2.GetDate 📌📌
2592 // const date=currDate.getDate()
2593 // console.log(date); //return 26{current date}
2594
2595
2596
2597 //?2.GetDay 📌📌
2598 // const day=currDate.getDay()
2599 // console.log(day); /return 6 {saturday index number is 6 and sun index number
    is 0}
2600
2601
2602
2603 //*JavaScript Get Time Methods 📌📌
2604
2605
2606 //?1.GetHours 📌📌
2607 // const hour=currDate.getHours()
```

```
2608 // console.log(hour); //return 16 {4'o clock}
2609
2610
2611
2612 //?2.GetMinutes 📌 📌
2613 // const minute=currDate.getMinutes()
2614 // console.log(minute); //return 10 {10 minutes }
2615
2616
2617
2618 //?3.GetSecond 📌 📌
2619 // const second=currDate.getSeconds()
2620 // console.log(second); //return 30 {30 second}
2621
2622
2623
2624 //?4.GetMillisecond 📌 📌
2625 // const millisecond=currDate.getMilliseconds()
2626 // console.log(millisecond); return 576 {576 millisecond as 1s=1000ms}
2627
2628
2629 //?5.GetTime 📌 📌
2630 // const time =currDate.getTime()
2631 // console.log(time); //return 1745910454425 ms from jan 1 1970
2632
2633
2634
2635 /*JavaScript Set Date Methods 📌 📌
2636
2637 // const date=new Date()
2638
2639
2640 //?1.setFullYear 📌 📌
2641 // date.setFullYear(2026)
2642 // console.log(date); return 2026-04-23T11:45:02.642z {2026 is set}
2643
2644
2645 //?2.setMonth 📌 📌
2646 // date.setMonth(5)
2647 // console.log(date); //return 2026-06-23T11:45:02.642z
2648
2649
2650 //?3.setDate 📌 📌
2651 // date.setDate(15)
2652 // console.log(date); //return 2026-06-15T11:45:02.642z
2653
2654
2655
2656
2657
2658 /*JavaScript Set Time Methods 📌 📌
2659
2660 //?1.setHours 📌 📌
2661 // date.setHours(10)
```

```
2662 // console.log(date); //return 10:26:24 {run in browser}
2663
2664
2665 //?2.setMinutes 📌📌
2666 // date.setMinutes(30)
2667 // console.log(date); //return 10:30:24
2668
2669
2670 //?3.setSecond 📌📌
2671 // date.setSeconds(45)
2672 // console.log(date); //return 10:30:45
2673
2674
2675 //?4.setMillisecond 📌📌
2676 // date.setMilliseconds(450)
2677 // console.log(date); //return 10:30:45 {we can't see millisecond}
2678
2679
2680
2681 //?5.setTime
2682 // date.setTime(1832090690883)
2683 // console.log(date); //return Fri Jan 21 2028 23:34:50 GMT+0530 (India Standard
Time)
2684
2685
2686
2687
2688
2689 /* A few useful methods of the date object in javascript
2690
2691 //?1.toLocaleString():→ return a string representing the date and time portion of
a date
2692 //? object using the current locals conventions
2693
2694 // console.log(date.toLocaleString()); //return '1/21/2028, 11:34:50 PM'
2695
2696
2697
2698 //?2.toLocaleDateString():→return a string representing the date and time portion
of a date
2699 //? object using the current locals conventions
2700 // console.log(date.toLocaleDateString()); //return 1/21/2028
2701
2702
2703
2704 //?3.toLocaleTimeString():→ →return a string representing the time portion of a
date
2705 //? object using the current locals conventions
2706 // console.log(date.toLocaleTimeString()); //return 11:34:50 PM
2707
2708
2709
2710
2711 //?parse():→ parse a string representation of a date and time return the number
```

```
2712         //? of millisecond since jan 1,1970 00:00:00 UTC
2713 // console.log(Date.parse('2025-04-28T05:16:30.238Z')); //return 1703054200000
    here Date is a object
2714
2715
2716
2717
2718
2719 //*Bonus 🙌🙌
2720
2721 //?Date.now()
2722 // let newDate=new Date()
2723 // console.log(Date.now()); //return 1745912201688 this is current date and time
2724 // console.log(newDate.getTime()); //also this return same
2725
2726
2727 //!it return the current timestamp (number of millisecond) representing the
    current moment
2728 //! use new Date().getTime() if you have an existing Date object from elsewhere
    and
2729 //! want it's timestamp
2730
2731
2732 //*interview Question 🙌🙌📚📚🧠🔧
2733
2734 //!2.write a function to add a specified number of days to a given date
2735
2736 // const addDate=(date,extraDate)⇒{
2737 //     return new Date(date.setDate(date.getDate()+extraDate)).toLocaleString()
2738
2739
2740 // }
2741
2742 // const date=new Date('2025-04-10')
2743 // console.log(addDate(date,7));
2744
2745
2746 //!3.Write a function to calculate the difference in days between two given dates
2747
2748 // const getDaysDifference=(date1,date2)⇒{
2749 //     let oneDayMs=24*60*60*1000
2750 //     let diff=date2-date1
2751 //     console.log(diff);
2752
2753 //     return diff/oneDayMs
2754 // }
2755
2756 // const date1=new Date('2024-02-19')
2757 // const date2=new Date('2024-03-01')
2758 // console.log(getDaysDifference(date1,date2));
2759
2760
2761
2762
```

```
2763
2764             /* #. 🔄🔄 List of Timing Based Events 🔄🔄
2765
2766 //todo1. setTimeout()
2767 //todo1. setInterval()
2768 //todo1. clearTimeout()
2769 //todo1. setInterval()
2770
2771
2772
2773 /**.setTimeout() 📌📌
2774 //? used to execute a function or code block after a specified delay in ms
2775
2776 //syntax 📌📌
2777
2778 // setTimeout(()⇒{
2779 //     console.log('it sehedule the callback function after a delay of 2000 ms');
2780
2781 // },2000) //after 2 sec function is run and then stop
2782
2783
2784 /**.setInterval() 📌📌
2785 //? used to repeatedly execute a function or code block after a specified
interval in ms
2786
2787 //syntax 📌📌
2788
2789 // setInterval(()⇒{
2790 //     console.log('it sehedule the callback function after a delay of 2000 ms');
2791
2792 // },2000) //after 2 sec function is run and run repeatedly until we stop
2793
2794
2795 //!setInterval⇒ execute a function after and after a particular time
2796 //!setTimeout⇒ execute a function after a particular time and then stop
2797
2798
2799
2800
2801 /**.clearTimeout() 📌📌
2802 //? if you want to cancle a scheduled timeout before it occur,you can
2803 //? use the clearTimeout function
2804
2805 //? it cancels a timeout priviously established by calling function
2806
2807
2808 //syntax 📌📌
2809 // clearTimeout(timeOutId)
2810
2811
2812 //Example 📌📌
2813
2814 // const clearTimeOut=setTimeout(()⇒{
2815 //     console.log('clear the time out');
```

```

2816
2817 // },2000)
2818
2819 // clearTimeout(clearTime) //it stop the execution of the function
2820
2821
2822
2823
2824 //*clearInterval() 🙌🙌
2825 //? if you want to cancle a scheduled interval before it occur,you can
2826 //? use the clearInterval function
2827
2828 //syntax 🙌🙌
2829 // clearInterval(IntervalId)
2830
2831
2832 //Example 🙌🙌
2833
2834 // const clearIntervals=setInterval(()⇒{
2835 //     console.log('clear interval before it occur');
2836
2837 // },2000)
2838
2839 // clearInterval(clearIntervals) //it stop the execution of the function
2840
2841
2842 //!write a javaScript program that defines a function called repeatedFunction.
2843 //!this function should log the message "this function repeat every 1000
ms(1sec)"
2844 //!then,set up interval using setInterval() to call repeatedFunction every 1000
2845 //!ms. additionally after 5 second have elapsed,use setTimeout() to clear the
2846 //! interval previously set up.make sure log the message "interval cleared after
"
2847 //!after 5 sec" when the interval is cleared
2848
2849 // const repeatedFunction={()⇒{
2850 //     console.log('this function repeat every 1000 ms(1sec)');
2851
2852 // }
2853
2854 // const intervalId=setInterval(repeatedFunction,1000)
2855
2856 // setTimeout(()⇒{
2857 //     clearInterval(intervalId)
2858 // },5000) //here we stop the function after 5 second
2859
2860
2861 //! the projects in JavaScript_Full_course\TimeBasedEvents
2862
2863
2864
2865 //* #. 🙌🙌 Object In JavaScript 🙌🙌
2866
2867

```

```
2868  /*What is the object 🤔🤔
2869  /*? Objects are a fundamental part of JavaScript, providing a way to group
2870  /*? related data and functions together. In JavaScript, an object is a collection
2871  /*? of key-value pairs, where each key is a string (or a symbol) and each value
2872  /*? be any data type, including other objects.
2873
2874  /*? Objects can have properties and
2875  /*? methods, making them versatile for various use cases.
2876
2877
2878  //syntax 🙋🙋
2879
2880  // const obj={}
2881
2882
2883  /*Creating Object
2884
2885  /*? There are several ways to create objects in JavaScript.
2886  /*? The most common one is using object literals: 🤔🙋🙋
2887
2888
2889  // let person = {
2890  //     name: "Adarsh",
2891  //     age: 30,
2892  //     "is'Student": false,
2893  //     greet: function () {
2894  //         console.log("Welcome to World Best JavaScript Course");
2895  //     },
2896  // };    //here person is a object name is a key and Adarsh is a value
2897
2898
2899
2900
2901
2902
2903  /*Accessing Properties
2904
2905  /*? You can access object properties using dot notation
2906  /*? or square bracket notation:
2907
2908  // let person={
2909  //     name:'Adarsh',
2910  //     age:22,
2911  //     "is'Student":false,
2912  //     greet:function (){
2913  //         console.log('hello');
2914  //     }
2915  // }
2916  // }
2917
2918  // console.log(person.age); //return 22
2919  // console.log(person.name); //return Adarsh
2920  //console.log(person.is'student); // give error so we use square bracket
```



```
2921 // console.log(person["is'Student']); //always put in string or back
    tick[template literals]
2922
2923
2924
2925
2926
2927
2928 /*Adding and Modifying Property
2929
2930 //? You can add new property or modify existing once
2931
2932 // person.job='web dev' //here we add a new property in person object
2933 // person.age=21; //here we modify the age
2934 // person['age']=21 //both are same
2935 // console.log(person); 🙋🙋🙋🙋
2936
2937 //!Result 🙋🙋
2938
2939 // let person={
2940 //     name:'Adarsh',
2941 //     age:21,
2942 //     "is'Student":false,
2943 //     greet:function (){
2944 //         console.log('hello');
2945
2946 //     },
2947 //     job:'web Dev',
2948 // }
2949
2950
2951
2952
2953
2954 /*Methods In Object
2955
2956 //? Methods in objects are functions associated with the object.
2957 //? They can be invoked using the same notation as properties:
2958
2959 // console.log(person.greet); //it return [Function:greet]
2960
2961 //To avoide this use 🙋🙋
2962 // console.log(person.greet()); //return hello
2963
2964
2965
2966
2967
2968 /*We Can Add Dynamic Key in an Object
2969
2970
2971 // let idType="studentId";
2972 // let student={
2973 //     idType:'A123456',
```

```
2974 //      s_name:'Adarsh',
2975 //      s_age:22,
2976 //      "is'Student":false,
2977 //      greet:function (){
2978 //          console.log(`hey my ${idType} is ${student[idType]} and my name is
2979 //          ${student.s_name}`);
2980 //      }
2981 // }
2982
2983 //? if we want to make idType dynamic we use []
2984
2985 // let idType="studentId";
2986 // let student={
2987 //     [idType]:'A123456',
2988 //     s_name:'Adarsh',
2989 //     s_age:22,
2990 //     "is'Student":false,
2991 //     greet:function (){
2992 //         console.log(`hey my ${idType} is ${student[idType]} and my name is
2993 //         ${student.s_name}`);
2994 //     }
2995 // }
2996
2997 // student.greet()    // return Hey my studentId is A123456 and my name is Adarsh
2998
2999
3000
3001
3002
3003 //*Data Modeling
3004
3005 //? Data modeling is the process of creating a visual representation of
3006 //? either a whole information system or parts of it to communicate
3007 //? connections between data points and structures. The goal is to illustrate
3008 //? the types of data used and stored within the system, the relationships among
3009 //? these data types, the ways the data can be grouped and organized and its
3010 //? formats and attributes.
3011
3012
3013 //? Objects are excellent for modeling real-world entities.
3014 //? For instance, you might represent a car, a user, or a product as
3015 //? an object with properties like color, brand, username, etc.
3016
3017 // let car ={
3018 //     brand:'Toyota',
3019 //     model:'camry',
3020 //     year:'2022',
3021 //     start:function(){
3022 //         console.log('start');
3023 //     }
3024 // }
3025 // } //all info of the car
```

```

3026
3027
3028
3029 //!Interview Question
3030
3031 //! Explain the difference between passing objects by reference and by value in
3032 JavaScript. Provide an example to demonstrate each scenario.
3033
3034 //? sol: In JavaScript, primitive data types like numbers and strings are
3035 //? passed by value, while objects are passed by reference.
3036
3037 //? Passing by value:=> When passing by value, a copy of the primitive value
3038 //? is created and passed to the function or assigned to a variable. Any changes
3039 //? made to the copy do not affect the original value.
3040
3041 // let a=10;
3042 // const modifiedValue=(x)=>(x=20)
3043 // console.log(modifiedValue(a));
3044 // console.log(a); //here original data is not change in pass by value
3045
3046
3047 //? Passing by reference: When passing by reference, a reference to the
3048 //? memory location of the object is passed to the function or assigned to a
3049 //? variable. Any changes made to the object through this reference will affect
3050 //? the original object.
3051
3052
3053
3054 // let obj={id:5,name:'Adarsh'}
3055
3056 // let obj1=obj
3057 // console.log(obj1); //it copy the obj into the obj1 and return
3058 // {id:5,name:'Adarsh'}
3059
3060 // obj.name='Vivek'
3061 // console.log(obj1); //{ id: 5, name: 'Vivek' }
3062 // console.log(obj); //{ id: 5, name: 'Vivek' } here value is change in both
3063 // obj
3064
3065 //? To avoid this behavior and create a true copy of the object,
3066 //? you can use methods like Object.assign() or the spread operator (...):
3067
3068
3069 //*1.Object.assign() :→is used to copy properties from one or more source
3070 //* objects to a target object.
3071
3072 // let obj={id:5,name:'Adarsh'}
3073
3074 // let newObj=Object.assign({},obj) //here obj copy in {}
3075
3076 // console.log(newObj); //return {id:5,name:'Adarsh'}

```

```

3077
3078 // newObj.name='Vivek'
3079 // console.log(newObj); //it change { id: 5, name: 'Vivek' }
3080
3081 // console.log(obj); //here original data not change {id:5,name:'Adarsh'}
3082
3083
3084
3085
3086 // *Comparison by Reference: 📌📌
3087
3088 //? Two objects are equal only if they refer to the same object.
3089 //? Independent objects (even if they look alike) are not equal:
3090
3091 // let obj1={name:'adarsh'}
3092 // let obj2={name:'adarsh'}
3093 // let obj3=obj1
3094
3095 // const isEqual=obj1===obj2 ? true:false
3096
3097 // console.log(isEqual); //return false
3098
3099 // const isEqual=obj3===obj1 ? true:false
3100 // console.log(isEqual); //true because obj3 refer to the obj1 and both memory
    is same hence it is true
3101
3102
3103
3104
3105
3106 // * =====
3107 // * "this" Object
3108 // * =====
3109
3110
3111 //? In JavaScript, the this keyword refers to an object.
3112
3113 // Which object depends on how this is being invoked (used or called).
3114
3115 // The this keyword refers to different objects depending on how it is used:
3116
3117 // In an object method, this refers to the object.
3118 // Alone, this refers to the global object.
3119 // In a function, this refers to the global object.
3120 // In a function, in strict mode, this is undefined.
3121 // In an event, this refers to the element that received the event.
3122 // Methods like call(), apply(), and bind() can refer this to any object.
3123
3124 // Note
3125 // this is not a variable. It is a keyword. You cannot change the value of this.
3126 // ("use strict");
3127
3128
3129

```

```

3130 // const obj={
3131 //     name:'Adarsh',
3132 //     greet:function(){
3133 //         console.log(this)
3134 //     }
3135 // }
3136
3137 // obj.greet() return {name:'Aarsh',greet:[Function:greet]} because this refer
to obj
3138
3139
3140 //? In this example, the greet method is defined using the
3141 //? "Method Shorthand" syntax.
3142 //? It's a more concise way to define methods in object literals.
3143
3144
3145 //*for method shorthand syntax
3146
3147 // const obj={
3148 //     name:'Adarsh Rai',
3149 //     greet(){                                //this is method shorthand syntax we no write
to fn keyword
3150 //         console.log(this);
3151
3152 //     }
3153 // }
3154
3155 // obj.greet() //return { name: 'Adarsh Rai', greet: [Function: greet] }
3156
3157
3158
3159 //*For fat arrow function
3160
3161 // const obj={
3162 //     name:'Adarsh Rai',
3163 //     greet:()=>{
3164 //         console.log(this);
3165
3166 //     },
3167 // }
3168
3169 // obj.greet() //return {}
3170
3171 //? 🙌🙌🙌 if i run in vs code arrow function do not have their own this
3172 //? they look outside to find what is this and in vs code this is just {}
3173 //? so it return {}⇒empty object
3174
3175 //?but if we run in console it return window object
3176
3177
3178
3179
3180
3181 //* =====

```

```

3182  /* Object Useful Method
3183  /*  =====
3184
3185
3186  // const product={
3187  //     id:1,
3188  //     name:'Leptop',
3189  //     category:'Computer',
3190  //     brand:'ExampleBrand',
3191  //     price:999.99,
3192  //     stock:50,
3193  //     description:'powerful laptop with a quad-core i5 processor 8gb ram,256gb
3194  //     image:'will be available',
3195
3196  // }
3197
3198
3199  /*Object.keys():→Returns an array containing the names of all enumerable
3200  /* own properties of an object.
3201
3202  // let keys=Object.keys(product)
3203  // console.log(keys);
3204
3205  //return 👉👉
3206  // [
3207  //     'id',
3208  //     'name',
3209  //     'category',
3210  //     'brand',
3211  //     'price',
3212  //     'stock',
3213  //     'description',
3214  //     'image'
3215  // ]
3216
3217
3218
3219
3220
3221  /*Object.value():return an array containing the values of all enumerable own
3222  /* properties of an object
3223
3224  // let values=Object.values(product)
3225  // console.log(values);
3226
3227  //rteurn 👉👉
3228  // [
3229  //     1,
3230  //     'Leptop',
3231  //     'Computer',
3232  //     'ExampleBrand',
3233  //     999.99,
3234  //     50,

```

```

3235 //      'powerful laptop with a quad-core i5 processor 8gb ram,256gb ssd',
3236 //      'will be available'
3237 //  ]
3238
3239 //*Object.entries():return an array containing array of key-value pairs for
3240 //*each enumerable own properties of an object
3241
3242 // let entries=Object.entries(product)
3243 // console.log(entries); //convert object into array every key-value pair
//convert into array
3244
3245 //return 👉👉
3246 // [
3247 //     [ 'id', 1 ],
3248 //     [ 'name', 'Leptop' ],
3249 //     [ 'category', 'Computer' ],
3250 //     [ 'brand', 'ExampleBrand' ],
3251 //     [ 'price', 999.99 ],
3252 //     [ 'stock', 50 ],
3253 //     [
3254 //         'description',
3255 //         'powerful laptop with a quad-core i5 processor 8gb ram,256gb ssd'
3256 //     ],
3257 //     [ 'image', 'will be available' ]
3258 // ]
3259
3260
3261 //*Object.hasOwnProperty(): return a boolean indicating whether the object has
//the
3262 //*specified property as an own property.
3263
3264 // console.log(product.hasOwnProperty('name')); //return true
3265 // console.log(product.hasOwnProperty('isStudent')); //return false
3266
3267
3268
3269
3270 //*Object.assign(): copies the value of all enumerable own property
3271 //* from one or more source object to a target object.
3272
3273 // const target={a:1,b:2,b:5}
3274 // const source={c:3,b:4}
3275 // const newObj={}
3276
3277 // Object.assign(newObj,target,source) //here target value and source value is
//copy to newObjcet
3278
3279
3280 // console.log(newObj); //{ a: 1, b: 4, c: 3 } here source b:4 is overwrite to
//target b:2 and than we get only sourse b:4
3281
3282
3283
3284

```

```

3285  /*object.freez():freezs an object,preventing new properties from being added to
      it
3286  /*and existing properties from being modified or deleted
3287
3288
3289  // Object.freeze(product)
3290  // product.id='5656'
3291  // console.log(product);  //we can't edit after applying object.freez
3292
3293
3294
3295
3296  //!Interview Question
3297
3298
3299  //!Given an object representing a student write a function to add a new
3300  //!subject with it's corresponding grade to the student record also check if
3301  //!the grade property is paresent or not
3302
3303
3304  // let student={
3305  //     name:'Bob',
3306  //     age:20,
3307  //     gardes:{
3308  //         math:90,
3309  //         science:85,
3310  //         history:88
3311  //     }
3312
3313  // }
3314
3315  // const addSubjectGrade=(obj,sub_name,grade)⇒{
3316  //     if(!obj.gardes){
3317  //         obj.gardes={}
3318  //     }
3319  //     return (obj.gardes[sub_name]=grade)
3320  // }
3321
3322  // console.log(addSubjectGrade(student,"computer",92));
3323  // console.log(student);
3324
3325
3326
3327
3328
3329
3330  //!write a function that transform an array of an object into an
3331  //!object where the key are the object ids
3332  //! convert into this 👉👉
3333  //! { '1': { id: 1, name: 'Adarsh' }, '2': { id: 2, name: 'Vivek' }, '3': { id:
      3, name: 'Amit' } }
3334
3335
3336  // let inputArray=[

```



```

3337 //      {id:1,name:'Adarsh'},
3338 //      {id:2,name:'Vivek'},
3339 //      {id:3,name:'Amit'},
3340 // ]
3341
3342
3343 // const arrToObj=(array)⇒{
3344 //     let obj={}
3345 //     for(let key of array){
3346
3347 //         obj[key.id]=key
3348
3349 //     }
3350
3351 //     return obj
3352 // }
3353
3354 // console.log(arrToObj(inputArray));
3355
3356
3357
3358
3359 //* #. 🤖🤖 How JavaScript work 🤖🤖
3360
3361
3362 //! 1: Parsing Phase
3363 //* 1. Lexical analysis
3364 //? In this phase JavaScript break the program into the tokens
3365
3366
3367 //* 2. Syntax analysis
3368 //? Takes the stream of tokens from lexical analysis and checks them
3369 //? for correct syntax. If the syntax is correct, syntax analysis generates
3370 //? a tree-like structure called a parse tree or abstract syntax tree (AST).
3371 //? The AST represents the hierarchical structure of the program.
3372
3373 //* 3. Compilation (JIT - Just-In-Time Compilation):
3374 //?here JavaScript convert the code into machine code/Bytes code
3375
3376 //* 3. Execution:
3377 //? Once the code is compiled, the JavaScript engine executes
3378 //? it. During execution, the engine creates execution contexts,
3379 //? manages the scope chain, handles variable assignments, and calls functions.
3380
3381 //? The execution context consists of two phases:
3382
3383 //*i Creation Phase → Variable and function are hoisted
3384 //*ii execution phase → where the code is actual run
3385
3386
3387 //?JavaScript engine use a call stack and heap memory
3388
3389 //! Call Stack
3390 //? In order to manage the execution contexts,

```

```
3391  //? the JavaScript engine uses a call stack.
3392  //? The call stack is a data structure that keeps track of
3393  //? the currently executing functions in a program. It operates on the
3394  //? Last In, First Out (LIFO) principle, meaning that the last function
3395  //? added to the stack is the first one to be executed and completed.
3396
3397  //! Heap Memory:
3398  //? The heap memory is where dynamically allocated memory resides.
3399  //? This is where objects, closures, and other dynamically allocated data are
3400  //? stored. While the call stack manages the flow of execution and function
3401  //? contexts, the heap memory holds data that is referenced by these execution
3402  //? contexts.
3403
3404
3405  //*Synchronous Vs Asynchronous
3406
3407  //? Synchronous code executes line by line, blocking further
3408  //? execution until each line is completed, while asynchronous
3409  //? code allows other code to continue executing while it waits for
3410  //? an asynchronous operation to complete.
3411
3412  //*Synchronous code
3413
3414  // const fun2={()=>{
3415  //     console.log('fun2 start and end');
3416  // }}
3417
3418
3419  // const fun1={()=>{
3420  //     console.log('fun1 start ');
3421  //     fun2()
3422  //     console.log('fun1 end ');
3423  // }}
3424
3425
3426  // fun1()    //here it run line by line
3427
3428
3429
3430
3431  //*Asynchronous code
3432
3433  // const fun2={()=>{
3434  //     setTimeout(()=>{
3435  //         console.log('fun2 start and end');
3436  //     },2000)
3437  // }}
3438
3439
3440
3441  // const fun1={()=>{
3442  //     console.log('fun1 start ');
3443  //     fun2()
3444  //     console.log('fun1 end ');
```

```
3445
3446 // }
3447
3448 // fun1() //here fun2 wait 2 sec and fu1 start run after executing fun2 also
execute
3449
3450
3451
3452
3453
3454 // *Hoisting In JavaScript
3455 // ? Hoisting is a JavaScript mechanism where variables and function
3456 // ? declarations are moved to the top of their scope before code execution.
3457 // ? This means that no matter where functions and variables are declared,
3458 // ? they are moved to the top of their scope regardless of whether their
3459 // ? scope is global or local.
3460
3461 // !But it work only in case of var not let and const
3462
3463
3464 // console.log(myVar);
3465 // greet();
3466
3467 // var myVar = 10;
3468 // function greet() {
3469 //     console.log("Welcome, If you are reading this, Don't forget you are
awesome");
3470 // };
3471
3472
3473
3474 // * _____
3475 // * Closure:
3476 // * _____
3477
3478 // ? A closure is created when an inner function has access to the
3479 // ? variables of its outer function, even after the outer function has
3480 // ? finished executing.
3481
3482
3483
3484 // function outerFunction(){
3485 //     var outerVariable='i am from outer'
3486
3487 //     function innerFunction(){
3488 //         console.log(outerVariable);
3489
3490 //     }
3491
3492 //     return innerFunction;
3493 // }
3494
3495 // var closureFunction=outerFunction()
```

```

3496 // closureFunction() // inner function access because the variable store in
    heap memory
3497
3498
3499
3500
3501
3502
3503 /* #. 🔄🔄 ECMAScript 2015-2023 🔄🔄
3504
3505
3506 /*ECMAScript-2015
3507
3508 /* Destructuring Arrays:
3509 /* Destructuring is a JavaScript expression that makes it possible to unpack
3510 /* values from arrays, or properties from objects, into distinct variables.
3511 /* That is, we can extract data from arrays and objects and assign them to
3512 /* variables.
3513
3514 /* 1: Extracting specific elements:
3515 // const numbers = [10, 20, 30];
3516 // const first = numbers[0]; // Traditional way
3517 // const [first, second, third] = numbers;
3518 // console.log(second); return 20
3519
3520 /* 2: Ignoring elements:
3521 // const [, , third] = numbers;
3522 // console.log(third);
3523
3524
3525 /*Write a program to swap two variable without using third variable
3526
3527 // let a=10;
3528 // let b=20;
3529
3530 // [a,b]=[b,a]
3531 // console.log(a,b);
3532
3533
3534
3535 /*Destructuring Object
3536
3537 // const user = {name: 'Adarsh', age: 30}
3538
3539 // const {myName, Myage} = user
3540 // console.log(myName, Myage); //return undefined because we should write only
    key name is object Destructuring
3541
3542 // const {name, age} = user
3543 // console.log(name, age); //now it give correct result
3544
3545
3546
3547

```

```
3548
3549 // *Rename Properties
3550
3551 // const user={
3552 //     name:'adarsh',
3553 //     age:30
3554 // }
3555
3556 // const {name:myName,age}=user;    //here we rename name with Myname
3557 // console.log(myName,age);
3558
3559
3560
3561 // *Spread Operator
3562
3563 // The syntax 🙌🙌
3564
3565 // ( ... )
3566
3567
3568 // *1. Copying an Array
3569
3570 // let fruits=['Apple','Orange','Mango'];
3571
3572 // let newFruits=[...fruits]
3573 // console.log(newFruits); //return ['Apple','Orange','Mango'];
3574
3575
3576 // *1. Concatonating The Array
3577
3578 // const num1=[1,2,3]
3579 // const num2=[4,5,6]
3580
3581 // const allNum=[...num1,...num2]
3582 // console.log(allNum); //return [ 1, 2, 3, 4, 5, 6 ]
3583
3584
3585
3586 // *Adding Element to existing array
3587
3588 // let num=[1,2,3,4,5]
3589 // num.push(...[6,7,8,9,10])
3590 // console.log(num); //return [1,2,3,4,5,6,7,8,9,10]
3591
3592
3593
3594 // ! One more useCases
3595 // ? In JavaScript, when you spread a string using the spread syntax (...),
3596 // ? it converts the string into an array of its individual characters.
3597
3598
3599 // *Traditional Way
3600
3601 // const country='INDIA'
```

```

3602 // console.log(country.split("")); //return [ 'I', 'N', 'D', 'I', 'A' ]
3603
3604 // *New Way
3605
3606 // console.log([ ... country]); //return [ 'I', 'N', 'D', 'I', 'A' ]
3607
3608
3609 // * =====
3610 // * Rest parameters - Modern JavaScript
3611 // * =====
3612 // ? The rest parameter syntax allows a function to accept an indefinite
3613 // ? number of arguments as an array, providing a more flexible way to work
3614 // ? with functions that can accept varying numbers of arguments.
3615
3616
3617 // const sum=( ... numbers)⇒{
3618 //     console.log(numbers); //it is array [ 1, 2, 3, 4 ]
3619
3620 //     return numbers.reduce((acc,curr)⇒{ //it is array so we use reducer
function for operation
3621 //         return acc+curr
3622 //     })
3623 // }
3624
3625 // console.log(sum(1,2,3,4));
3626
3627
3628 // TODO NOTE: A function definition can only have one rest parameter, and the rest
parameter must be the last parameter in the function definition.
3629 // function wrong1( ... one, ... wrong) {} //there should be only one rest
parameter
3630 // function wrong2( ... wrong, arg2, arg3) {} //always put in last not first
because it take all argument
3631
3632
3633
3634
3635
3636 // *ECMAScript-2016
3637
3638 // *Exponentiation Operator
3639
3640 // ? ES7 introduces a new mathematical operator called exponentiation operator.
3641 // ? This operator is similar to using Math.pow() method. Exponentiation operator
3642 // ? is represented by a double asterisk **. The operator can be used only with
3643 // ? numeric values.
3644
3645
3646 // Syntax 👉👉
3647
3648 // base_value ** exponent_value
3649
3650 // Example 👉👉
3651

```

```

3652 // let base=2
3653 // let expo=3
3654
3655 // console.log(base ** expo);    //return 8 as 2 to the power 3 is 8
3656
3657
3658
3659 

---



---


3660 

---

    ECMAScript Features (2017) / ES8
3661 

---



---


3662
3663 

---

    List of new useful features added in ES8 📌
3664 

---

    1. String padding
3665 

---

    2. Object.values()
3666 

---

    3. Object.entries()
3667 

---

    4. Trailing commas in function parameter lists and calls
3668 

---

    5. Async functions
3669
3670 

---



---


3671 

---

    String padding
3672 

---



---


3673
3674 

---

    String padding in JavaScript is a way to add extra characters (like spaces)
3675 

---

    to a string to make it a specific length.
3676
3677 

---

    Using padStart() to pad from the beginning:
3678
3679 

---

    const Myname='Adarsh'
3680 

---

    const padName=Myname.padStart(10)
3681 

---

    console.log(padName);    //it add space from beggning
3682 

---

    console.log(padName.length);
3683
3684 

---

    console.log(Myname.padEnd(10,'$'));    //here we can also add any thing in
3685 

---

    palce of space    it add 4 star and length become 10
3686
3687
3688
3689 

---



---


3690 

---

    Trailling Comma
3691 

---



---


3692
3693 

---

    function greet(name,age,boolean,){    //here we can add commo it not give error
3694 

---

        console.log(`hello ${name} you are ${age} year old`);
3695
3696 

---

    }
3697
3698 

---

    greet('Adarsh',30,true,)    //here also
3699
3700
3701
3702
3703
3704

```

```

3705  /** =====
3706  /**      ECMAScript Features (2018) / ES9
3707  /** =====
3708
3709  //? List of new useful features added in ES8 📌
3710  //todo 1. Rest/Spread Properties
3711  //todo 2. Promise.prototype.finally()
3712
3713
3714
3715  /** Rest/Spread Properties
3716
3717  // const number=[1,2,3,4]
3718  // const [first ,second, ... other]=number    //here we use rest operator in array
3719  // console.log(other);
3720
3721
3722
3723  /**Rest Operator in Object
3724
3725  // `const student={
3726  //     name:'Adarsh',
3727  //     age:10,
3728  //     isStudent:true,
3729  // }
3730
3731  // const {age, ... other}=student
3732  // console.log(other);` //it return { name: 'Adarsh', isStudent: true } new obj
3733
3734
3735
3736  // const obj1={a:10,b:20,c:50}
3737  // const obj2={c:30,d:40}
3738  // const newObj={ ... obj1, ... obj2}
3739
3740  // console.log(newObj); //it return { a: 10, b: 20, c: 30, d: 40 } here it
overWrite c value in obj not array
3741
3742
3743
3744
3745
3746  /** =====
3747  /**      ECMAScript Features (2019) / ES10
3748  /** =====
3749
3750  //? List of new useful features added in ES8 📌
3751  //todo 1. Array.prototype.{flat,flatMap}
3752  //todo 2. Object.fromEntries()
3753  //todo 3. String.prototype.{trimStart,trimEnd}
3754  //todo 4. Symbol.prototype.description
3755  //todo 5. Optional catch binding
3756
3757

```



```

3758
3759 /**Array.flat() and Array.flatMap()
3760
3761 /**? flat() is a new array instance method that can create a one-dimensional
3762 /**? array from a multidimensional array. (nested arrays into a single, flat
3763 array.)
3764
3765 // const array=[1,2,[3,4],5]
3766 // console.log(array.flat()); //here 2-dimension array convert to single
3767 dimeansion
3768
3769
3770 // const array2=[1,[2,[3,4]],5]
3771 // console.log(array2.flat()); //it return [ 1, 2, [ 3, 4 ], 5 ] it properly
3772 not convert in single dimension so we increae the level
3773
3774
3775 // const array3=[1,[2,[3,4]],5]
3776 // console.log(array3.flat(2)); //here we increase the level and return [ 1,
3777 2, 3, 4, 5 ]
3778
3779 // const array4=[1,[2,[3,[4]]],5]
3780 // console.log(array4.flat(3)); //here we increase the more level
3781
3782
3783
3784 /**? flatMap() is a new Array instance method that combines flat() with map().
3785 /**? It's useful when calling a function that returns an array in the map()
3786 /**? callback, but you want your resulted array to be flat:
3787
3788
3789 /**If we use Map
3790
3791 // const arr=['my name','is adarsh','rai']
3792
3793 // const newArr=arr.map((elem)⇒{
3794 //     return elem.split(' ')
3795 // })
3796
3797 // console.log(newArr); //it return multi dimension array [ [ 'my', 'name' ],
3798 [ 'is', 'adarsh' ], [ 'rai' ] ]
3799
3800
3801 /**To avoid this we use flatMap()
3802
3803 // const arr=['my name','is adarsh','rai']
3804
3805 // const newArr=arr.flatMap((currElem)⇒{
3806 //     console.log(currElem);

```

```

3807
3808 //      return currElem.split(" ")
3809 // })
3810
3811 // console.log(newArr);    //now it return single dimension array[ 'my', 'name',
    'is', 'adarsh', 'rai' ]
3812
3813
3814
3815
3816 /* =====
3817 /*  Object.fromEntries()
3818 /* =====
3819 //? Objects have an entries() method, since ES2017.
3820 //? It returns an array containing all the object own properties,
3821 //? as an array of [key, value] pairs:
3822
3823 // const person={name:'adarsh',age:30}
3824 // const entries=Object.entries(person)
3825 // console.log(entries);    //convert object to array in key pair form
3826
3827 // const newPerson=Object.fromEntries(entries)
3828 // console.log(newPerson);    //convert back to the object from array
3829
3830
3831
3832 // console.log(person===newPerson);    //here perosn and new person are not same
    obj because
3833                                     //there memory location is difference we
    compare object based on reference not values
3834
3835
3836
3837
3838
3839
3840 /* =====
3841 /*  String.prototype.{trimStart,trimEnd}
3842 /* =====
3843
3844 //? trimStart(): Return a new string with removed white space from the start
3845 //? of the original string
3846 // console.log("Testing".trimStart());
3847 // console.log("      Testing".trimStart());
3848 // console.log("    Testing    ".trimStart());
3849 // console.log("Testing    ".trimStart());
3850
3851 //? trimEnd(): Return a new string with removed white space from the end of
3852 //? the original string
3853 // console.log("Testing".trimEnd());
3854 // console.log("      Testing".trimEnd());
3855 // console.log("    Testing    ".trimEnd());
3856 // console.log("Testing    ".trimEnd());
3857

```

```
3858
3859
3860 

---



---


3861 

---


3862 

---



---


3863
3864 

---


3865 

---


3866 

---


3867 

---


3868
3869 

---


3870 

---


3871 

---


3872 

---


3873
3874 

---


3875 

---


3876 

---


3877
3878
3879 

---



---


3880 

---


3881 

---



---


3882 

---


3883 

---


3884 

---


3885
3886 

---


3887 

---


3888 

---


3889 

---


3890 

---


3891 

---


3892
3893 

---


3894 

---


3895 

---


3896 

---


3897 

---


3898 

---


3899
3900
3901
3902
3903
3904 

---



---


3905 

---


3906 

---



---


3907
3908 

---


3909 

---


3910 

---


3911 

---


```

```

3912 // promise.allSettled
3913
3914
3915
3916 

---



---


3917 

---


3918 

---



---


3919 

---


3920 

---


3921
3922
3923 

---


3924 

---


3925
3926 

---


3927 

---


3928 

---


3929
3930
3931 

---


3932 

---


3933
3934
3935 

---


3936 

---


3937 

---


3938
3939
3940
3941 

---



---


3942 

---


3943 

---



---


3944 

---


3945 

---


3946 

---


3947 

---


3948
3949 

---


3950 

---


3951 

---


3952
3953
3954 

---


3955 

---


3956 

---


3957
3958
3959
3960 

---



---


3961 

---


3962 

---



---



```

```

3963 //? It provides a concise way to access properties of an object without worrying
3964 //? about the existence of intermediate properties. It's particularly useful when
3965 //? working with nested objects or accessing properties of objects that may be
3966 //? null or undefined.
3967
3968 // const person = {
3969 //   name: "John",
3970 //   address: {
3971 //     city: 'newYork',
3972 //     zipCode: 12345,
3973 //   },
3974 // };
3975
3976
3977 // const city=person.address?.city
3978 // console.log(city); //return newyork
3979
3980
3981
3982 // const person = {
3983 //   name: "John",
3984 //   address: {
3985 //     city: "New York",
3986 //     zipCode: 12345,
3987 //     coordinates: {
3988 //       latitude: 40.7128,
3989 //       longitude: -74.006,
3990 //     },
3991 //   },
3992 // };
3993
3994
3995 // const latitude = person.address?.coordinates?.latitude ?? "not present";
3996 // console.log(latitude); //retrun 40.7128
3997
3998
3999 /** =====
4000 /**   ECMAScript Features (2021) / ES12
4001 /** =====
4002
4003 //? List of new useful features added in ES8 📌
4004 // String.prototype.replaceAll()
4005 // Logical Assignment Operators ( ||=, &&=, ??=)
4006 // Numeric Separators
4007 // Promise.any()
4008
4009 /** =====
4010 /**   String.prototype.replaceAll()
4011 /** =====
4012 //? replaceAll in JavaScript is a function that replaces all occurrences of a
4013 //? specified value with another value in a given string.
4014
4015 //? Replacing all occurrences of a word:
4016 // const originalString = "Hello, world! Hello again.";

```

```

4017 // const newString = originalString.replaceAll("Hello", "Hi");
4018 // console.log(newString);
4019
4020 //? Replacing multiple spaces with a single space:
4021 // const text = "This   has   extra       spaces.";
4022 // const normalizedText = text.replaceAll(/\s+/g, " ");
4023 // console.log(normalizedText);
4024
4025 //* =====
4026 //* Logical Assignment Operators ( ||=, &&=, ??=)
4027 //* =====
4028
4029 //? Logical OR-Assignment ( ||=): This operator assigns the value of its
4030 //? right-hand operand to its left-hand operand if the left-hand operand
4031 //? evaluates to a falsy value (false, null, undefined, 0, '', NaN). Otherwise,
4032 //? it leaves the left-hand operand unchanged.
4033
4034 // let x = false;
4035 // x = x || true; // equivalent to: x = x || true;
4036 // console.log(x); // Output: true
4037
4038 // let y = 10;
4039 // y ||= 20; // equivalent to: y = y || 20;
4040 // console.log(y); // Output: 10 (unchanged)
4041
4042 //? Logical AND-Assignment (&&=): This operator assigns the value of its
4043 //? right-hand operand to its left-hand operand if the left-hand operand
4044 //? evaluates to a truthy value. Otherwise, it leaves the left-hand operand
4045 //? unchanged.
4046 // let x = true;
4047 // x &&= false; // equivalent to: x = x && false;
4048 // console.log(x); // Output: false
4049
4050 // let y = 0;
4051 // y &&= 20; // equivalent to: y = y && 20;
4052 // console.log(y); // Output: 20
4053
4054 //* =====
4055 //* Numeric Separators
4056 //* =====
4057 //? This feature allows underscores (_) to be used as separators within numeric
4058 //? literals to improve readability.
4059 // const bigNumber = 1_000_000;
4060 // console.log(bigNumber);
4061 // Output: 1000000;
4062
4063
4064
4065
4066 //* =====
4067 //* ECMAScript Features (2022) / ES13
4068 //* =====
4069
4070 //? List of new useful features added in ES8 👉

```

```

4071 // .at() function for indexing
4072 // Object.hasOwn(obj, propKey)
4073
4074
4075
4076
4077 /** =====
4078 /**  .at()
4079 /** =====
4080 /** Before ES2022, square bracket notation was used to fetch a particular
4081 /** element in an array. This method is straightforward unless you need to
4082 /** perform a backward iteration, i.e., negative indexing. In the case of
4083 /** negative
4084 /** indexing, the common practice was to use arr[arr.length - N],
4085 /** where array length is referred to and indexed from the end.
4086 /** The .at() method introduced in ES2022 has simplified this process. In a case
4087 /** of positive indexing, .at() will work the same as the square brackets. But for
4088 /** negative indexing, the .at() method allows starting the iteration from the end.
4089
4090 // const array = [1, 2, 4, 5, 6, 7];
4091
4092 // console.log(array.at(-1)); //return 7
4093
4094
4095
4096 /** =====
4097 /**  Object.hasOwn(obj, propKey)
4098 /** =====
4099 /** Object.hasOwn() is a static method that you can use to check if a
4100 /** property exists in an object or not. It returns true if the specified object
4101 /** contains the indicated property as its own, and if the property is inherited
4102 /** or doesn't exist, it returns false. This method takes the object as the first
4103 /** argument and the property you want to check as the second.
4104
4105 /** Object.hasOwn is the intended alternative for the
4106 /** Object.prototype.hasOwnProperty
4107 /** method. Although Object.prototype.hasOwnProperty has been in JavaScript
4108 /** specification for quite a time, it has some drawbacks.
4109
4110 // const book = {
4111 //   name: "World Best JS Course",
4112 //   author: "Thapa Technical",
4113 // };
4114
4115 // console.log(book.hasOwnProperty('name')); //return true
4116
4117
4118 /** Issue with hasOwnProperty it doesn't work for object created
4119 /** using Object.create(null)
4120

```

```

4121
4122 // const student =Object.create(null)
4123 // student.name='Adarsh'
4124 // console.log(student.hasOwnProperty('name')); //it give erroe
4125
4126 // console.log(Object.hasOwn(student,'name')); //it retun true result
4127
4128
4129 

---



---


4130 

---

 ECMAScript Features (2022) / ES13 

---


4131 

---



---


4132
4133 

---

/? List of new useful features added in ES8 📌
4134 // Array.findLast()
4135 // Array.findLastIndex()
4136 // Array.prototype.toReversed()
4137 // Array.prototype.toSorted(compareFn)
4138 // Array.prototype.toSpliced(start, deleteCount, ... items)
4139 // Array.prototype.with(index, value)
4140
4141 

---



---


4142 

---

 Array.findLast() & Array.findLastIndex() 

---


4143 

---



---


4144 

---

/? This function will allow us to find element from the last to first
4145 

---

/? of the array based on a condition.
4146 // const array = [1, 2, 3, 4, 5, 6, 4];
4147 // console.log(array.findLast((elem) => elem > 4));
4148 // console.log(array.findLastIndex((elem) => elem));
4149
4150 

---



---


4151 

---

 New Array.prototype functions 

---


4152 

---



---


4153
4154 const myNames = ["Adarsh", "Rai", "Vivek", "HackLikeBerlin"];
4155
4156 

---

 Array.prototype.toReversed();
4157 // const reversedNum = myNames.toReversed();
4158 // console.log("original", myNames);
4159 // console.log("reversed", reversedNum);
4160 //todo if it's not working run in browser
4161
4162 

---

 Array.prototype.toSorted(compareFn);
4163 // const sortedArr = myNames.toSorted();
4164 // console.log("original", myNames);
4165 // console.log("sorted", sortedArr);
4166
4167 

---

Array.prototype.toSpliced(start, deleteCount, ... items);
4168 // const splicedArr = myNames.toSpliced(1, 1, "thapaTechnical");
4169 // console.log("original", myNames);
4170 // console.log("spliced", splicedArr);
4171
4172 

---

 Array.prototype.with(index, value);
4173 

---

/?The with() method in JavaScript is used to change the value
4174 

---

/? of an element at a specific index in an array. It takes two arguments:

```



```

4175 //? the index of the element to be changed and the new value.
4176 //? It returns a new array with the changed element,
4177 //? leaving the original array unchanged.
4178
4179 // const replaceWith = myNames.with(2, "thapaTechnical");
4180 // console.log("original", myNames);
4181 // console.log("replaced", replaceWith);
4182
4183
4184
4185             /* #. 🐞🐞Advanced JavaScript 🐞🐞
4186
4187 //todo 1.Event Propagation (Bubbling & Capturing)
4188 //todo 2.Higher order function
4189 //todo 3.Callback function
4190 //todo 4. Closure & function curring
4191 //todo 5.Callback hell
4192 //todo 6.fetch Api
4193 //todo 7.Promises
4194 //todo 8. Asyn-await
4195 //todo 9. Error Handling
4196
4197
4198
4199 /* 1.Event Propagation (Bubbling & Capturing)
4200
4201 //? it is the process of how event propogates or travel through the DOM hierarchy
4202 //? in JS there are tow phase of event propagation capturing phase and bubbling
    phase
4203
4204
4205
4206 /*Capturing Phase
4207 //? the event start from the root of the D0m and goes down to the target elem
4208
4209 /*Target Phase
4210 //?the event reaches the target element
4211
4212 /*Bubbling Phase
4213 //? the event start from the target elem and bubbles up to the root of the DOM
4214 //? this is default
4215
4216 //!JavaScript_Full_course\Event_Propagation\Event_Propagation.html ⇒ check
    example here
4217
4218
4219
4220
4221 /* =====
4222 /* Event Delegation:
4223 /* =====
4224 //? Event delegation is a concept in JavaScript where instead of attaching event
4225 //?listeners to individual elements, you attach a single event listener to a
4226 //?common ancestor of those elements. This is particularly useful when you have a

```

```
4227 //?large number of similar elements and want to reduce the number of event
4228 listeners,
4229 //?improve performance, and simplify code.
4230 /* Example: List with Delegation →
4231 //? Consider a scenario where you have an unordered list (<ul>) with multiple
4232 //? list items (<li>), and you want to handle click events on each list item.
4233 //? Instead of adding a separate event listener to each list item,
4234 //? you can use event delegation. →</li>
4235
4236 //!JavaScript_Full_course\Event_Propagation\Event_delegation.html ⇒ here is a
4237 example of event delegation
4238
4239
4240 /* _____
4241 /* First-Class Function - it's just a concept
4242 /* _____
4243
4244 //? A "first-class function" means that functions can be treated as
4245 //? values, assigned to variables, and passed around as arguments.
4246
4247 // function sayHello(){
4248 //     return 'hello'
4249 // }
4250
4251 // var greetFunction=sayHello();
4252 // console.log(greetFunction); //this is first order function
4253
4254
4255
4256
4257
4258
4259 /* _____
4260 /* Higher-Order Functions:
4261 /* _____
4262 //? Definition: A higher-order function is a function that takes one or
4263 //? more functions as arguments or returns a function as a result.
4264
4265 /* _____
4266 /* Callback Functions:
4267 /* _____
4268 //? Definition: A callback function is a function passed as an argument
4269 //? to another function and is executed after the completion of a task.
4270
4271 /* Here is the example ✅
4272 // Callback function
4273 // function processUserInput(name, greetUser) {
4274 //     console.log("Received input: " + name);
4275 //     greetUser(name);
4276 // }
4277
4278 /* Function to be used as a callback
```

```

4279 // function greetUser(name) {
4280 //   console.log(`Hello! ${name}`);
4281 // }
4282
4283 // processUserInput("Vinod", greetUser);
4284
4285 /* processUserInput ⇒
4286  ?? is a higher-order function because it takes another
4287  ?? function (callback) as an argument.
4288  * greetUser ⇒
4289  ?? is a callback function because it's passed as an argument to
4290  ?? processUserInput and gets executed after the completion of the main task
4291
4292
4293
4294  * =====
4295  * Interview Question:
4296  * =====
4297  !! Write a program to perform mathematical operations using callback
4298  !! functions and two variables in JavaScript.
4299
4300
4301 // const mathOperation = (a, b, operation) ⇒ {
4302 //   return operation(a, b);
4303 // };
4304
4305 //   const add = (a, b) ⇒ {
4306 //     return a + b;
4307 // };
4308
4309 //   const sub = (a, b) ⇒ {
4310 //     return b - a;
4311 // };
4312
4313 //   console.log(mathOperation(5, 15, add));
4314 //   console.log(mathOperation(5, 15, sub));
4315
4316
4317
4318
4319
4320  * =====
4321  * Callback hell
4322  * =====
4323
4324  ?? Callback hell, also known as the Pyramid of Doom, refers to a
4325  ?? situation in asynchronous JavaScript programming where multiple nested
4326  ?? callbacks are used to handle asynchronous operations.
4327  ?? This often results in code that is difficult to read, understand,
4328  ?? and maintain due to its deeply nested structure.
4329
4330 // const studentData=(()=>{
4331 //   setTimeout(()⇒{
4332 //     console.log('hello i am adarsh ');

```

```

4333 //      setTimeout(()⇒{
4334 //          console.log('hello my last name is rai  ');
4335 //      setTimeout(()⇒{
4336 //          console.log('born in mumbai  ');
4337 //      setTimeout(()⇒{
4338 //          console.log('i am from up  ');
4339 //      setTimeout(()⇒{
4340 //          console.log('i am mern stack developer  ');
4341 //      setTimeout(()⇒{
4342 //          console.log('i like cricket ');
4343 //      },1000)
4344 //      },1000)
4345 //      },1000)
4346 //      },1000)
4347 //      },1000)
4348 // },1000)
4349 // }
4350
4351 // studentData()
4352
4353
4354
4355
4356 /** =====
4357 /**  Promise in JavaScript
4358 /** =====
4359
4360 /**? 👉 In simpler terms, a promise is like a placeholder for the result
4361 /**? of an asynchronous operation. Or A container for the future result or value.
4362
4363 /*** It can be in one of three states:
4364
4365 /**? Pending: Initial state, neither fulfilled nor rejected.
4366 /*** Fulfilled(Resolved): The operation completed successfully.
4367 /**! Rejected: The operation failed or encountered an error.
4368
4369
4370 /**? Promises have built-in methods like then and catch to handle the results
4371 /**? of asynchronous operations when they complete or encounter errors, making it
4372 /**? easier to write asynchronous code that is more readable and maintainable
4373 /**? compared to traditional callback-based approaches.
4374
4375
4376 /*** =====
4377 /*** Using the Promise Constructor (Class):
4378 /*** =====
4379
4380 /**? You can create a promise using the Promise constructor.
4381 /**? This involves creating a new instance of the Promise class,
4382 /**? which takes a function as an argument. This function, often referred
4383 /**? to as the "executor function," takes two parameters: resolve and reject.
4384 /**? You call resolve when the asynchronous operation is successful and reject
4385 /**? when it encounters an error.
4386

```

```

4387 // const Promise=new Promise(function(resolve,reject){
4388 //   code
4389 // })
4390
4391
4392 

---



---


4393 

---



---


4394 

---



---


4395
4396 

---



---


4397 

---



---


4398 

---



---


4399 

---



---


4400 

---



---


4401
4402 // syntax
4403 // function myPromiseFunction() {
4404 //   return new Promise((resolve, reject) => {
4405 //     code
4406 //   });
4407 // }
4408
4409
4410
4411
4412 //!Exmaple 1. 🙌🙌🙌
4413
4414 // const pr=new Promise((resolve,reject)=>{
4415 //   setTimeout(()=>{
4416 //     resolve('hii miss you')
4417 //   },2000)
4418 // })
4419
4420
4421 // .then((res)=>{
4422 //   console.log(res);
4423 // }).catch((error)=>{
4424 //   console.log(error);
4425 // }).finally(()=>{
4426 //   console.log('dont worry we all miss you and keep smiling');
4427 // })
4428
4429
4430
4431
4432
4433
4434 //!Exmaple 2. 🙌🙌🙌
4435
4436 // const studentName='Adarsh'
4437
4438 // const enrollStudent=(studentName)=>{
4439 //   return new Promise((res,rej)=>{
4440 //     setTimeout(()=>{

```

```

4441 //          const isSuccessfull=Math.random()>0.4;
4442 //          if(isSuccessfull){
4443 //              res(`Enrollment successful for ${studentName}`)
4444 //          }
4445 //          else{
4446 //              rej(`Enrollment failed for ${studentName}. Please try again.`)
4447 //          }
4448 //          },2000)
4449 //      })
4450 //  }
4451
4452
4453 // enrollStudent(studentName)
4454
4455 // .then((res)⇒{
4456 //     console.log(res);
4457
4458 // }).catch((e)⇒{
4459 //     console.log(e);
4460
4461 // }).finally(()⇒{
4462 //     console.log("Enrollment process completed.");
4463
4464 // })
4465
4466
4467
4468
4469
4470 /* =====
4471 /* Promise Methods
4472 /* =====
4473
4474 /* Promise.all()
4475 /*? is used when you want to wait for all promises to complete
4476 /*? successfully. Reject state will throw an error.
4477
4478 /*? Promise.allSettled()
4479 /*? is used when you want to wait for all promises to complete,
4480 /*? regardless of success or failure, and get information about their outcomes.
4481
4482 /* Promise.race()
4483 /*? is used when you are interested in the result of the first
4484 /*? promise that completes, regardless of success or failure.
4485
4486
4487 //!All Example 🙌🙌🙌
4488
4489 // const promise1=new Promise((resolve,reject)⇒{
4490 //     setTimeout(()⇒resolve('First'),2000)
4491 // })
4492
4493 // const promise2=new Promise((resolve,reject)⇒{
4494 //     setTimeout(()⇒resolve('Second'),5000)

```

```

4495 // })
4496
4497 // const promise3=new Promise((resolve,reject)⇒{
4498 //     setTimeout(()⇒resolve('Third'),3000)
4499 // })
4500
4501 // const promise4=new Promise((resolve,reject)⇒{
4502 //     setTimeout(()⇒reject('Fourth'),100)
4503 // })
4504
4505
4506
4507 // Promise.all([promise1,promise2,promise3,promise4])
4508 // .then((res)⇒{
4509 //     console.log(res);
4510
4511 // }).catch((e)⇒{
4512 //     console.log(e);
4513
4514 // })
4515
4516
4517
4518 // Promise.allSettled([promise1,promise2,promise3,promise4])
4519 // .then((res)⇒{
4520 //     console.log(res);
4521
4522 // }).catch((e)⇒{
4523 //     console.error(e);
4524
4525
4526 // })
4527
4528
4529
4530 // Promise.race([promise1,promise2,promise3,promise4])
4531 // .then((res)⇒{
4532 //     console.log(res);
4533
4534 // }).catch((e)⇒{
4535 //     console.log(e);
4536
4537 // })
4538
4539
4540 //!JavaScript_Full_course\Promise\Promise.html ⇒check the project in this file
4541
4542
4543
4544
4545 //* =====
4546 //* Async and Await
4547 //* =====
4548

```

```
4549  //? async and await are JavaScript keywords introduced in ECMAScript 2017 (ES8)
4550  //? that make asynchronous code look and behave more like synchronous code,
4551  //? making it easier to write, read, and reason about asynchronous operations. --
4552  >
4553  //? async Function Declaration: The async keyword is used to
4554  //? declare an asynchronous function. An asynchronous function returns a
4555  //? Promise implicitly, even if the return value is not explicitly wrapped
4556  //? in a Promise. Inside an asynchronous function, you can use the await
4557  //? keyword to pause the execution of code until a Promise is resolved or
4558  //? rejected. →
4559
4560  //? await Operator: The await keyword is used to pause the execution
4561  //? of an async function until a Promise is settled (resolved or rejected).
4562  //? It can only be used inside an async function. When await is used with a
4563  //? Promise, it waits for the Promise to resolve and returns the resolved value.
4564  //? If the Promise is rejected, it throws an error that can be caught using a
4565  //? try...catch block. →
4566
4567
4568
4569  //!JavaScript_Full_course\Promise\Asyn-await.html ⇒ check here project
4570
4571
4572
4573
4574  /** =====
4575  /**  Error-handling in JS(try-catch)
4576  /** =====
4577
4578  //? In JavaScript, the try...catch statement is used for error handling.
4579  //? It allows you to catch and handle exceptions (errors) that occur within a
4580  //? block of code. Here's how it works:
4581
4582  //? try Block: The try block contains the code that you want to execute.
4583  //? It is the block of code where you anticipate that an error might occur. →
4584
4585  //? catch Block: The catch block follows the try block and is used to
4586  //? catch any exceptions (errors) that occur within the try block.
4587  //? If an exception occurs, JavaScript jumps to the catch block to handle
4588  //? the error. The catch block takes an error object as a parameter,
4589  //? which can be used to access information about the error, such as
4590  //? its message or stack trace. →
4591
4592  //? finally Block (Optional): The finally block, if provided,
4593  //? is executed regardless of whether an error occurs or not.
4594  //? It is typically used for cleanup tasks that should always be
4595  //? performed, such as closing resources or releasing locks. →
4596
4597
4598
4599  //!JavaScript_Full_course\Promise\error-handling.html ⇒ check here for example
4600
4601
```



//\* 😊😊😊😊 🎉🎉🎉🎉🎉 🎊🎊🎊🎊🎊 ...END.. 🎉🎉🎉🎉🎉 🎊🎊🎊🎊🎊😊😊😊😊