

```
1  **Javascript**
2
3
4  **Background**
5
6  <!-- Previously javascript was compiled only by the
7  browser as only browser was
8  having the compiler which can understand
9  java script and therefore js was predominantly known
10  as client side scripting language.
11  Thus previously if we needed to run a js file we
12  needed to add scripts in the
13  index.html and then run the local host
14  and then the browser was able to understand the
15  script.
16
17  However later on the compiler was been taken
18  individually and is known as node.js.
19  This helps in understanding of the js code
20  without the need of the browser and
21  thus the scope of js increased to
22  server side as well.
23
24  Dino.js is also one such compiler.
25  Thus in order to run standalone js files
26  in our machine we need node.js
27
28
29  Js is dynamically typed language as we
30  don't give types of any variable and
31  is by default taken by
32  js and moreover its one variable assigned
33  to number can be again reassigned
34  back to a string even. -->
35
36
37  <----->
38
39  **Hoisting**
40
41  <!-- Hoisting is a behavior where variable and
42  function declarations are moved
43  to the top of their scope during the compilation
44  phase,
45  before the code is executed. This means you can use
46  variables or call
47  functions before they are declared in the code.
48
49  However different dataTypes have different
50  properties like:
51
52  1. var type - its declaration is hoisted and not
53  the value at top of the scope and its value is
54  initialized with undefined
55
56  1. let type - its declaration is hoisted at top of the
57  scope and are not initialized and accessing them before
58  declaration result in reference error as they are
59  placed in temporal dead zone and are not accessible to
60  compiler before its initialization in execution phase
61
62  2. const type - its declaration is hoisted at top
63  of the scope and are not initialized and accessing them
64  before declaration result in reference error as they are
65  placed in temporal dead zone and are not accessible to
66  compiler before its initialization in execution phase -->
67
```

```

68
69 <!-- console.log(hoistVar); undefined and no
70 reference error -->
71 var hoistVar = 10;
72
73 <!-- console.log(hoistLet);
74 // reference error -->
75 let hoistLet = 10;
76
77 <!-- console.log(hoistConst);
78 // reference error -->
79 const hoistConst = 10;
80
81
82
83 <!-- /**Functions are also hoisted before
84 their declaration so we can call
85 a function even before its declared */ -->
86 greet();
87
88 function greet() {
89     console.log("Hello, world!");
90 }
91
92
93 <!-- /**Data types when used as functional
94 expressions are also treated as same based
95 on hoisting rules for var, let and const */ -->
96
97 <!-- console.log(foo); // undefined -->
98 var foo = function () {
99     console.log("Hi!");
100 };
101
102 <!-- console.log(bar);
103 // ReferenceError -->
104 let bar = function() {
105     console.log("Hi!");
106 };
107
108
109
110 <!-- Lets understand why hoisting happens.
111
112 So there are two phases one is compilation and other
113 is execution.
114
115 During compilation all variables are hoisted so for
116 example say our code is:
117
118     console.log(a);
119     var a = 10;
120
121     console.log(b);
122     let b = 20;
123
124     hello();
125     function hello() {
126         console.log("Hello, world!");
127     }
128
129
130
131 Now compilation phase will happen and everything
132 will be hoisted
133
134 // During compilation phase:

```

```

135
136     var a;           // Hoisted and initialized with
137                       undefined
138
139     let b;           // Hoisted but uninitialized
140                       (in temporal dead zone)
141
142     function hello() {
143     console.log("Hello, world!");
144     }
145
146
147
148 // During execution phase:
149     console.log(a); // undefined
150     a = 10;
151
152     console.log(b); // ReferenceError
153                       (b is in Temporal dead zone)
154     b = 20;
155
156     hello();         // "Hello, world!"
157
158

```

So to summarize, during compilation phase all the functions and variables declarations are hoisted. var and other functions are taken to top of scope let and const are taken to temporal dead zone.

Now in case of var its hoisted by taken to top of scope and initialized with value undefined, in case of let and const its hoisted by taken to temporal dead zone and not initialized.

Now when console.log(a) comes , compiler sees it is present and it was hoisted with value undefined so undefined gets printed.

Now a gets set to 10.

Now compiler see console.log(b), compiler can't find it as its hoisted and present in temporal dead zone and thus reference error comes up -->

<----->

****Var, let & Const****

<!--

* 1. Scope difference:

*

* var has a function/global scope where as let and const are having block scope only.

*

* So a is defined inside a block but still its having a block and global

* scope thus we can print a,

* however b and c are let and

```

202     * const respectively and thus its having block scope only and are
203     * not
204     * accessible outside the block.
205     -->
206
207     if (true) {
208         var a = "10";
209         let b = 1;
210         const c = 2;
211     }
212
213     console.log(a);
214     console.log(b);
215     console.log(c);
216
217
218
219     <!-- /**
220     * 2. Hoisting difference:
221     *
222     * var, let and const all are hoisted however only var is hoisted and
223     * initialized with
224     * undefined where as let and const are hoisted in
225     * temporal dead zone but not initialized and due to their presence
226     * in temporal dead zone
227     * after hoisting if we access them before its
228     * declaration it gives a reference error.
229     *
230     */ -->
231
232     console.log(p);
233     var p = 10;
234
235     console.log(q);
236     let q = 20;
237
238     console.log(r);
239     const r = 1;
240
241
242
243     <!-- * 3. Re declaration
244     *
245     * var can be re-declared in the same scope whereas
246     * let and const can not be re
247     * declared in the same scope -->
248
249
250     var name = "Gaurav";
251     var name = "Pankaj";    // allowed
252
253
254     let surname = "bhatt";
255     // let surname = "bhatt";    // not allowed
256
257     const z = 1;
258     // const z = 2;    // not allowed
259
260
261
262     <!--
263     * 4. Re Assignment
264     *
265     * var and let can be re-assigned in the same scope
266     * whereas const
267     * can not be re assigned in the same scope -->
268

```

```

269
270 var school = "DDPS";
271 school = "Dps";    // allowed
272
273
274 let city = "Ghaziabad";
275 city = "Banglore";    // not allowed
276
277 const state = 1;
278 // state = 2; // not allowed
279
280
281
282 <
-----
->

283
284 **Data types**
285
286
287 <!-- * Java script has primitive as well as non primitives
288 data types:
289 *
290 * Primitives: They are called by values
291 *
292 * string
293 * number
294 * BigInt
295 * boolean
296 * undefined
297 * symbol
298 * null
299 *
300 *
301 * Non primitives: They are called by reference
302 *
303 * object
304 * Array
305 * functions
306 * -->
307
308
309 let a = "Gaurav";
310 let b = 3;
311 let c = BigInt(10);
312 let d = true;
313 <!-- // e is declared but never initialized and thus its undefined as
314 its value is not defined -->
315 let e;
316 let f = null; // this means f is empty and has no value
317
318 let id1 = Symbol('123');
319 let id2 = Symbol('123');
320
321 <!-- // false as symbol is used to make things unique no
322 matter even if -->
323 console.log(id1 === id2)
324 same values are being passed.
325
326
327 console.table([a, b, c, d]);
328 console.log(typeof undefined) // undefined
329 console.log(typeof null) // object
330
331 /**Its an object and typeof obj is obj itself */
332 let obj = {
333     name: "Gaurav",

```

```

334     age: 23
335 }
336 console.log(typeof obj); // obj
337
338
339 let addFunc = function addNumberS(a, b) {
340     return a + b;
341 }
342 console.log(typeof addFunc); // function
343
344 let arr = [1,2,3];
345
346 console.log(typeof arr) // obj
347
348
349 <!-- * The main difference between null and undefined is
350 undefined means
351 * the value is not defined yet but its data type
352 * is itself undefined
353 *
354 * whereas null means void or nothing or empty and doesn't
355 * means its not defined.
356 * Its type is an object.
357 -->

```

<----->

****Type conversions****

```

363
364
365 <!-- /**
366  * Conversion to number type
367  */ -->
368
369 let a = "0";
370 console.log(Number(a)); // 0
371
372 let b = true;
373 console.log(Number(b)); // 1
374
375 let c = "33abcd";
376 console.log(Number(c)); //NaN
377
378 let d = NaN;
379 console.log(Number(d)); // Nan
380
381 let e = null;
382 console.log(Number(e)); // 0
383
384 let f = undefined;
385 console.log(Number(f)); // Nan
386
387
388 <!-- /**
389  * O/p:
390  *
391  */ -->

```

| (index) | Values |
|---------|--------|
| 0 | 0 |
| 1 | 1 |
| 2 | NaN |
| 3 | NaN |
| 4 | 0 |
| 5 | NaN |

```

401 |_____|
402 */ -->
403
404 console.table([Number(0), Number(true), Number("33abc"), Number(NaN), Number(null),
405               Number(undefined)]);
406
407 <!-- /**Operations
408  *
409  * If we are concat a string with number, the result will always be a string
410  * concat only, however
411  * if two numbers are added before string concat , then numbers are added first
412  * and then concatenated
413  * with string.
414  * Ex: console.log(1+2+"3") will be 33
415  * however
416  * console.log("1"+2+3) will be 123
417 */ -->
418
419 console.log("1" + "2") // 12
420 console.log("1" + 2) // 12
421 console.log(1 + "2") //12
422 console.log("1" + 2 + 3) // 123
423 console.log(1 + 2 + "3") // 33
424 <!-- // + operator, when used with a non-number operand,
425 tries to convert it into a number so 1 -->
426 console.log(+true);
427
428
429 <!-- /**Comparisons
430  *
431  * In comparisons things are being converted to numbers and then
432  * checked upon, in case of ==
433  * only values are compared in case of === values as well as data
434  * types are being compared.
435  */ -->
436
437 <!-- //true because only values are compared -->
438 console.log(2 == "2");
439
440 <!-- //false as type and value both are compared -->
441 console.log(2 === "2");
442
443 <!-- // true as "2" is converted to a number and compared
444 with number 1 -->
445 console.log("2" > 1)
446
447 <!-- // null when converted to number is 0 and 2 when
448 converted to number is 2 -->
449 console.log(null > "2");
450
451 so 0> 2 is false.
452
453 console.log(null == "2") // false
454
455
456 <
-----
-->
457
458 **Stack and heap memory**
459
460 <!-- /**
461  * Just like other programming languages there are two kinds
462  * of memory in js as well
463  * i,e heap and stack memory.
464  *

```

```
465 * All primitives are call by value or a copy of their value
466 * is provided whereas
467 * all non primitives in js are call by
468 * reference.
469 *
470 * Let's take an example to understand
471 *
472 * let name = "Gaurav";
473 * let name1 = name;
474 *
475 * name1 = "bhatt";
476 *
477 * console.log(name);
478 * console.log(name1);
479 *
480 * o/p is: "Gaurav"
481 *        "bhatt"
482 *
483 * Let's understand how flow happened.
484 *
485 * So when we said let name so a name variable
486 * got declared inside
487 * stack memory and it is referencing
488 * to a memory address in heap
489 * which contains a value "Gaurav"; say $100000
490 *
491 * Now when we said let name1 so a name1 variable
492 * got declared inside
493 * stack memory and is referencing to a
494 * new memory address in
495 * heap which contains a duplicated value from name
496 * "Gaurav". say $100001
497 *
498 * Now when we changed name1 to bhatt so the name1
499 * in stack was referencing
500 * to $100001 and thus at that address
501 * the value got changed
502 * from "gaurav" to "bhatt" and the name variable value which
503 * is present
504 * in $100000 remained intact.
505 *
506 * This same happens with all primitives in js.
507 *
508 * However in js , the non primitives like array,
509 * object and functions
510 * are passed by reference. Let's understand this.
511 *
512 * let obj = {
513 *   name: "Gaurav",
514 *   email: "bhatt@yahoo"
515 * }
516 *
517 * let obj1 = obj;
518 * obj1.name = "bhatt";
519 * console.log(obj.name);
520 * console.log(obj1.name);
521 *
522 * o/p : bhatt
523 *      bhatt
524 *
525 * So when we said let obj so obj named variable
526 * got created in the
527 * stack which is referencing to a memory location
528 * in heap say $111111 which is storing two of the
529 * properties i,e name and email.
530 *
531 * Now when we said let obj1 = obj so we created
```



```
532     * another variable
533     * obj1 inside stack and it is referencing to the same
534     * memory location which obj is
535     * referencing to i,e $111111
536     * and thus any change by obj1 in any of the properties in directly
537     * changing the properties present in the same
538     * reference as that of obj.
539 */ -->
```

```
540
541
542
543 <!-- /**
544     * Pass by values for primitives
545     */ -->
```

```
546
547 let name = "Gaurav";
548 let name1 = name;
```

```
549
550 name1 = "bhatt";
```

```
551
552 console.log(name);
553 console.log(name1);
```

```
554
555
556 <!-- /**
557     * Pass by reference for non primitives
558     * */ -->
```

```
559
560 let obj = {
561     name: "Gaurav",
562     email: "bhatt@yahoo"
563 }
```

```
564
565 let obj1 = obj;
566 obj1.name = "bhatt";
567 console.log(obj.name);
568 console.log(obj1.name);
```

```
569
570
571
572 <
-----
->
```

```
573
574 **Strings**
```

```
575
576 <!-- /**
577     * The main difference between
578     * let str = "Gaurav" and let str1 = new String("Gaurav")
579     * is that the first syntax creates
580     * a primitive string whereas the
581     * second string creates a string object.
582     *
```

```
583     * Lets understand it:
584     *
```

```
585     *         let str = "Gaurav";
586     *         let str1 = str;
587     *         str1 = "Bhatt";
```

```
588
589         console.log(str);
590         console.log(str1);
```

```
591
592     str variable is created in stack and has reference to a
593     memory location which is
594     storing the value
595     "Gaurav" and is a primitive string.
596     We said str1 = str;
```

```

597
598 Now str1 is given a copy of str, so str1 is created
599 in stack and now is
600 referencing to a new memory
601 address which is storing a copy
602 of str i,e "Gaurav";
603
604 Once we change str1 then str1's value which is
605 present in a
606 different memory address gets
607 updated and thus no effect on str.
608
609 Lets understand string object:
610
611     let str2 = new String("Gaurav1");
612     let str3 = str2;
613
614     str3 = "Testing";
615     console.log(str2);
616     console.log(str3);
617
618 when we say str2 = new String so str2 gets created
619 in stack and now is
620 pointing towards a memory
621 address which holds a string object.
622 when we say str3 = str2 now str3 variable gets created in
623 the stack and
624 is also referencing to
625 same memory location as that of str2.
626 Objects are passed by reference.
627
628 However now we did str3 = "Testing", i,e assigned it to a primitive
629 string and not changed
630 the property of the object which it's referencing
631 to and thus this is a case of reference reassignment thus now
632 str3 is pointing to a new memory address which holds
633 a primitive string "Testing" thus str2 remains unchanged.
634
635 Lets summarize:
636
637 str2 Points to a String Object:
638
639 str2 holds a reference to a String object (String { "Gaurav1" }).
640 This object is stored in memory.
641 str3 = str2; Shares the Reference:
642
643 str3 now references the same object as str2. Both
644 point to the same memory location.
645 str3 = "Testing"; Reassigns str3:
646
647 When you write str3 = "Testing";, you are not modifying
648 the String object. Instead:
649 str3 is reassigned to point to a new primitive string ("Testing").
650 This breaks the reference link between str2 and str3.
651 str2 Remains Unchanged:
652
653 str2 still holds the reference to the original
654 String object in memory
655 because you never modified the object itself.
656 You only changed what str3 points to.
657
658 */ -->
659
660
661 let str = "Gaurav";
662 let str1 = str;
663 str1 = "Bhatt";

```

```

664
665 console.log(str); // "Gaurav"
666 console.log(str1); // "Bhatt"
667
668
669 let str2 = new String("Gaurav1");
670 let str3 = str2;
671
672 str3 = "Testing";
673 console.log(str2);
674 console.log(str3);
675
676
677
678 <!-- /**
679  * Even methods which manipulates the string,
680  * doesn't go and effect
681  * the object as they give back a
682  * primitive string.
683  *
684  * So str4 was an reference variable pointing to an
685  * string object in heap
686  * str5 is also a reference variable pointing to the same
687  * object as that of str4
688  *
689  * str5 = str5.replaceAll('a','b')
690  * replaced all a's with b's and is a primitive string
691  * and thus reference
692  * reassignment happened
693  * and str4 remain intact.
694  */ -->
695
696 let str4 = new String("Gaurav12");
697 let str5 = str4;
698
699 str5 = str5.replaceAll('a','b');
700
701 console.log(str4); // Object{'Gaurav'}
702 console.log(str5); // Gburbv12
703
704
705
706 <----->
707
708 **Arrays**
709
710 // const _ = require('lodash');
711
712 <!-- /**
713  * Arrays are objects in js.
714  */ -->
715
716
717 <!--
718 /**
719  * Here we are creating a reference to the same array in
720  * memory.
721  * This means arr2 and arr point to the exact same object,
722  * not separate copies.
723  * Thus change in any property will lead to change in original
724  * object itself.
725  */ -->
726
727 let arr = [1, 2, 3, 4, 5];
728 let arr2 = arr;
729 arr2[0] = 9;
730

```

```

731 console.log(arr); // [ 9, 2, 3, 4, 5 ]
732 console.log(arr2); // [ 9, 2, 3, 4, 5 ]
733
734
735 <!-- /**
736  * Shallow copy:
737  * In shallow copying a copy of the original object is
738  * created however its
739  * nested properties are shared via reference and
740  * any change in the nested property will change the
741  * original object.
742  *
743  * So in below an array variable is created in stack
744  * and is referencing
745  * to an object in heap.
746  * when we do array1 = [...array] we are doing a shallow copy
747  * so a new variable array1 is
748  * created which is referencing to a new
749  * memory address and the object is shallow copied
750  * from original object however the nested
751  * properties are still referenced to the
752  * same original object and thus any change in nested
753  * property will also mark a change in original object.
754  *
755  * array1[5].name = "bhatt" will also change the name value in array
756  * variable also as we are changing nested property
757  *
758  * array1[0] = 9 will only change the array1 and not array itself.
759  *
760  * There are many ways of shallow copy like: [...origValue] or Array.from(origValue)
761  */ -->
762
763 let array = [1, 2, 3, 4, 5, { "name": "gaurav" }];
764 let array1 = [...array]; // shallow copy
765
766 array1[5].name = "bhatt"
767 console.log(array);
768 console.log(array1);
769
770 const originalArray = [1, 2, { a: 3 }];
771 const shallowCopy = Array.from(originalArray); // shallow copy
772
773
774
775 <!-- /**Deep copy
776  *
777  * A deep copy creates an entirely copy that is completely independent
778  * of the original. It copies all levels of the structure, including
779  * nested properties ensuring there is no shared references between the
780  * original and the copy.
781  *
782  * Thus any change in the copied value will not affect the original
783  *
784  * There are many ways to deep copy like:
785  * JSON.parse(JSON.stringify(origArray)),
786  * _.cloneDeep(),
787  * structuredClone()
788  *
789  */ -->
790
791 let orig = [1, 2, [3, 4], { name: "gaurav" }];
792 let deepCopy = JSON.parse(JSON.stringify(array)); // deep copy
793
794 deepCopy[2][0] = 9; // Modify nested array
795 deepCopy[3].name = "bhatt"; // Modify nested object
796
797 console.log(orig); // [1, 2, [3, 4], { name: "gaurav" }]

```

```

798 console.log(deepCopy); // [1, 2, [9, 4], { name: "bhatt" }]
799
800
801
802 // let deepCopy1 = _.cloneDeep(orig); // deep copy
803 // deepCopy1[2][0] = 9;
804
805 console.log(orig); // [1, 2, [3, 4], { name: "gaurav" }]
806 console.log(deepCopy); // [1, 2, [9, 4], { name: "gaurav" }]
807
808 let deepCopy3 = structuredClone(orig); // deep copy
809
810
811
812 <!-- /**
813  * Slice vs splice
814  *
815  * In slice and well as splice the last index does not get included ,
816  * in case of slice the original array
817  * does not get altered whereas in case of splice the original
818  * array gets altered.
819  */ -->
820
821 let original = [1, 2, 3, 4, 5];
822 let sliced = original.slice(0, 3); // last index not included
823
824 console.log(`After slicing original does not change and remains ${original} and sliced
array is ${sliced}`);
825
826 let spliced = original.splice(0, 3); // last index not included
827 console.log(`After splicing original got changed to ${original} and spliced array is
${spliced}`);
828
829
830
831 <!-- /**Merge arrays
832  *
833  * Spread operator is used to merge arrays.
834  * Spread means when we use spread on anything it
835  * scatters its individual values
836  */ -->
837
838 let a1 = [1, 2, 3, 4, 5];
839 let a2 = [6, 7, 8, 9, 10];
840
841 console.log([...a1, ...a2]);
842
843 <!--
844 /**Flat
845  *
846  * This is used to flat an nth depth array
847  */ -->
848
849 let a3 = [1, 2, [3], [4, [5, [6]]], [7, 8, 9], 10];
850 console.log(a3.flat(Infinity));
851
852
853
854 <
-----
-->
855
856 **Js execution**
857
858 <!-- /**Entire end to end javascript execution explanation:**/ -->
859
860 let num1 = 2;

```

```

861 let num2 = 3;
862
863 function addNumbers(num1, num2) {
864     return num1 + num2;
865 }
866
867 let res1 = addNumbers(num1, num2);
868 let res2 = addNumbers(3, 7);
869
870 <!-- /**
871
872  * Javascript execution happens in javascript execution context
873  * which is composed of
874  *
875  * 1. Global execution context
876  * 2. Functional execution context
877  *
878  * Global execution context is different for js executed on browser
879  * (window object) and is different from execution in node (global object)
880  *
881  * In each of these context there composed of two things:
882  * 1. Memory Creation phase
883  * 2. Execution phase
884  *
885  * So first in memory creation phase for above code.
886  *
887  *     num1 is hoisted to top of its scope assigned in memory and
888  *     send to temporal dead zone as its a let.
889  *     They are not accessible until the code reaches
890  *     their declaration.
891  *
892  *     same happens with num2.
893  *
894  *     The function addNumbers is hoisted with its
895  *     definition
896  *
897  *     res1 and res2 are hoisted and send into TDZ
898  *     like num1 and num2.
899  *
900  *
901  * Now comes the execution phase.
902  *
903  *     num1 reaches its declaration and gets assigned
904  *     with value 2
905  *     num2 reaches its declaration and gets assigned
906  *     with value 3
907  *
908  *     function addNumber was already hoisted
909  *     with its definition.
910  *
911  *     res1 was called as return from function
912  *     addNumber thus now function addNumber
913  *     needs to be executed
914  *     and thus function addNumbers is pushed to
915  *     call stack.
916  *
917  *     Now for addNumbers again two process happen
918  *     just like two process of memory
919  *     creation and execution happened for
920  *     GEC or global execution context
921  *
922  *     **in memory creation phase num1 and num2
923  *     which are function parameters gets
924  *     hoisted and the arguments passed to
925  *     function gets assigned with value of num1 and num2.
926  *
927  *     After than execution happens and sum

```

```

928      *           of num1 and num2 is returned.
929      *
930      *           Once its complete for addNumbers its
931      *           execution context is destroyed and function is
932      *           taken out from call stack and value it
933      *           returns gets assigned to res1.
934      *
935      *           Now res2 was called as return from
936      *           function addNumber thus now function
937      *           addNumber needs to be executed
938      *           and thus function addNumbers is pushed to call stack.
939      *
940      *           Now for addNumbers again two process happen
941      *           just like two process of memory
942      *           creation and execution happened for
943      *           GEC or global execution context
944      *
945      *           Once its complete for addNumbers its execution context is
946      *           destroyed and function is
947      *           taken out from call stack and value it returns
948      *           gets assigned to res2.
949      *
950      */ -->
951
952
953
954      <!-- /**
955      * Let's understand call stack concept in js with
956      * help of an example.
957      */ -->
958
959      function one() {
960          console.log(`Inside function one`);
961      }
962
963      function two() {
964          console.log(`Inside function two`);
965      }
966
967      function three() {
968          console.log(`Inside function three`);
969      }
970
971      one();
972      two();
973      three();
974
975      <!-- /**
976      * So javascript executes line by line,
977      * so first it came to one() and sees that
978      * its being called so one() came
979      * inside the function call stack and gets executed,
980      * since execution of one() is completed so its being
981      * removed from function call
982      * stack and then two() is inserted in call stack
983      * after two() is completed then three() is
984      * pushed to call stack and then
985      * once executed is pushed out from call stack.
986      * This call stack is known as functional call
987      * stacks and takes care of the
988      * function execution schedule.
989      *
990      */ -->
991
992
993      <

```

```

----->
994
995 **If & else**
996
997 <!-- /**
998  * The condition inside if is evaluated as either
999  * a true value or a false value.
1000  * However we can also evaluate truthy or falsy values.
1001  *
1002  * There are some specific things which are
1003  * considered as falsy like
1004  *
1005  * "", 0, -0, 0n, null, Nan, undefined, false
1006  * rest are considered as truthy value.
1007  */ -->
1008
1009 let userName = "g@bhatt"
1010 if (userName) {
1011     console.log(`Username evaluated to truthy`);
1012 }
1013
1014
1015 <!-- /**Switch case */ -->
1016
1017 switch (userName) {
1018     case "g@bhatt":
1019         console.log(`Found Username : ${userName}`);
1020         break;
1021     case "panda@abc":
1022         console.log(`Found Username : ${userName}`);
1023         break;
1024     default:
1025         console.log("woooo");
1026 }
1027
1028
1029 <!-- /**
1030  * The nullish coalescing operator (??)
1031  *
1032  * It is a logical operator that provides a way to
1033  * assign a default value to a
1034  * variable when the original value is null or undefined.
1035  */ -->
1036
1037 let val = null;
1038 val = null ?? 10;
1039 // console.log(val);
1040
1041
1042 <!-- /**
1043  * Ternary operator
1044  */ -->
1045
1046 let val1 = 3 >= 3 ? 'yes' : 'false';
1047
1048
1049 <
-----
->
1050
1051 **Objects**
1052
1053
1054 <!-- /**
1055  * An object in js is a dataType .
1056  *
1057  * There are two ways to create an object i,e via

```



```

1058 * 1. Object literal
1059 * let obj = {};
1060 * This is not a singleton objects i,e multiple instances
1061 * of obj can be
1062 * created and shared across.
1063 *
1064 * 2. Constructor Function
1065 * let obj = new Object();
1066 * This is a singleton object i,e this is the only single
1067 * instance of obj and
1068 * only this single instance
1069 * will be shared across all the other places if needed.
1070 */ -->
1071
1072
1073 <!-- /**
1074 * 1.Object literal:
1075 */ -->
1076
1077 let obj = {
1078     name: "Gaurav",
1079     age: 27,
1080     "full_name": "Gaurav Bhatt",
1081     email: "abc@pqrs.com",
1082
1083 }
1084
1085 // console.log(obj.name);
1086 // console.log(obj["full_name"]);
1087 // console.log(obj["email"]);
1088
1089
1090 <!-- /**1.1
1091 *
1092 * Insert a dynamic valued key in an object,
1093 *
1094 * this dynamic key property is accessed via obj[nickname]
1095 * and not obj["nickName"]
1096 * whereas rest normal properties are accessed either as
1097 * obj["name"] or obj.name
1098 */ -->
1099
1100 let nickName = "PropertyName";
1101 obj = {
1102     name: "Gaurav",
1103     age: 27,
1104     "full_name": "Gaurav Bhatt",
1105     email: "abc@pqrs.com",
1106     [nickName]: "booo"
1107 }
1108
1109 // console.log(obj[nickName]);
1110
1111
1112 <!-- /**1.2
1113 *
1114 * Insert a dynamic key as a property which is having
1115 * a dynamic value in an object
1116 * */ -->
1117
1118 let propName = "prop_name";
1119 let propValue = "prop_value";
1120
1121 obj = {
1122     name: "Gaurav",
1123     age: 27,
1124     "full_name": "Gaurav Bhatt",

```

```

1125     email: "abc@pqrs.com",
1126     [propName]: `${propValue}`
1127 }
1128
1129 // console.log(obj [propName]);
1130
1131
1132 <!-- /**1.3
1133  *
1134  * Change value of a property in object
1135  * */ -->
1136
1137 obj[propName] = 'prop_value_1'
1138 // console.log(obj);
1139
1140
1141 <!-- /**1.4
1142  *
1143  * Don't allow any change in value for any property
1144  * in object
1145  * */ -->
1146
1147 Object.freeze(obj);
1148 obj[propName] = 'prop_value_2';
1149 // console.log(obj);
1150
1151
1152 <!-- /**
1153  * 1.5
1154  *
1155  * Don't allow change only for email property
1156  * in object
1157  *
1158  */ -->
1159
1160 propName = "prop_name";
1161 propValue = "prop_value";
1162
1163 let obj1 = {
1164     name: "Gaurav",
1165     age: 27,
1166     "full_name": "Gaurav Bhatt",
1167     email: "abc@pqrs.com",
1168     [propName]: `${propValue}`
1169 }
1170
1171 Object.defineProperty(obj1, "email", {
1172     writable: false, // prevents the property from being modified
1173
1174     configurable: false // prevents the property from being
1175                        deleted or reconfigured
1176 });
1177
1178 obj["email"] = "change@change.com";
1179 // console.log(obj1);
1180
1181
1182 <!-- /**
1183  * 1.6
1184  *
1185  * Insert a function in value of a key in object
1186  */ -->
1187
1188 obj1 = {
1189     name: "Gaurav",
1190     age: 27,
1191     "full_name": "Gaurav Bhatt",

```



```

1259 <!-- // In case of nested objects it don't destructure
1260 the object entirely and give in
1261 accordance to the first or top level -->
1262
1263
1264 obj3 = {
1265     email: "xyz@gmail.com",
1266     details: {
1267         fullName: {
1268             firstName: "gaurav",
1269             lastName: "bhatt"
1270         },
1271         address: {
1272             pinCode: 201002,
1273             city: "ghaziabad"
1274         }
1275     }
1276 }
1277
1278 const keys = Object.keys(obj3);
1279 const values = Object.values(obj3);
1280
1281 // console.log(keys); // [ 'email', 'details' ]
1282
1283 // console.log(values);
1284 /**
1285     'xyz@gmail.com',
1286     {
1287         fullName: { firstName: 'gaurav', lastName: 'bhatt' },
1288         address: { pinCode: 201002, city: 'ghaziabad' }
1289     }
1290 ] */
1291
1292
1293 <!-- /**
1294  * 1.10
1295  *
1296  * Check if a property exists in an object
1297  */ -->
1298
1299 obj3 = {
1300     email: "xyz@gmail.com",
1301     details: {
1302         fullName: {
1303             firstName: "gaurav",
1304             lastName: "bhatt"
1305         },
1306         address: {
1307             pinCode: 201002,
1308             city: "ghaziabad"
1309         }
1310     }
1311 }
1312
1313 // console.log(obj3.hasOwnProperty("email"));
1314
1315
1316 <!-- /**
1317  *
1318  * 1.11
1319  *
1320  * Destructure an object
1321  */ -->
1322
1323 obj3 = {
1324     email: "xyz@gmail.com",
1325     details: {

```

```

1326         fullName: {
1327             firstName: "gaurav",
1328             lastName: "bhatt"
1329         },
1330         address: {
1331             pinCode: 201002,
1332             city: "ghaziabad"
1333         }
1334     }
1335 }
1336
1337 let { email, details } = obj3;
1338 let { fullName, address } = details;
1339 // console.log(details);
1340 // console.log(email);
1341 // console.log(fullName);
1342 // console.log(address);
1343
1344
1345
1346 <!-- /**
1347  * Property descriptor and making a property non
1348  * writable in js
1349  */ -->
1350
1351 let object = {
1352     name: "Gaurav",
1353     phoneNumber: 9354337987,
1354     email: "bhatt@xcy.com"
1355 }
1356
1357 console.log(Object.getOwnPropertyDescriptor(object, 'name'));
1358 <!-- /**
1359  * We got o/p as
1360  * {
1361     value: 'Gaurav',
1362     writable: true,
1363     enumerable: true,
1364     configurable: true
1365  }
1366
1367  so every property inside an object also has few
1368  properties like
1369
1370  writable: true,
1371  enumerable: true,
1372  configurable: true
1373
1374  so we can alter these properties.
1375  writable means that this property of the object can be edited
1376  enumerable means that this object property is iterable
1377  and by making it false
1378  it wont get detected if we iterate over the object
1379  if we want to edit any property we can do.
1380  */ -->
1381
1382 for (let [key, value] of Object.entries(object)) {
1383     console.log(`${key}: ${value}`);
1384 }
1385
1386 Object.defineProperty(object, 'phoneNumber', {
1387     writable: false,
1388     enumerable: false
1389 })
1390
1391 object.phoneNumber = 99999999;
1392 console.log(object);

```

```
1393 for (let [key, value] of Object.entries(object)) {
1394     console.log(`${key}: ${value}`);
1395 }
1396
1397
1398 <----->
1399
1400 **Object practices problems**
1401
1402 <!-- /**
1403  * Given an object, give me all the keys and values of
1404  * this object
1405  *
1406  */ -->
1407
1408 const obj = {
1409     name: "Alice",
1410     age: 30,
1411     salary: 50000
1412 }
1413
1414 for (let [key, value] of Object.entries(obj)) {
1415     console.log(`${key}: ${value}`);
1416 }
1417
1418
1419 <!-- /**
1420  *
1421  * Given an array of objects simple, find the total
1422  * salary in this
1423  */ -->
1424
1425 const employees = [
1426     { name: "Alice", age: 30, salary: 50000 },
1427     { name: "Bob", age: 25, salary: 60000 },
1428     { name: "Charlie", age: 35, salary: 70000 }
1429 ];
1430
1431 function getTotalSalary(employees) {
1432     let sm = 0;
1433     for (let item of employees) {
1434         if (item.hasOwnProperty("salary")) {
1435             sm += item["salary"];
1436         }
1437     }
1438     return sm;
1439 }
1440
1441 // console.log(getTotalSalary(employees));
1442
1443
1444
1445 <!-- /**
1446  *
1447  * Given an array of objects complex, find the
1448  * total salary in this
1449  */ -->
1450
1451 const employeeDetails = [
1452     {
1453         name: "Alice",
1454         age: 30,
1455         carrer: [
1456             {
1457                 salary: 1,
1458                 position: "SDE",
1459                 dept: "IT"
```

```

1460     },
1461     {
1462         salary: 1,
1463         position: "SDE",
1464         dept: "IT"
1465     },
1466     {
1467         salary: 1,
1468         position: "SDE",
1469         dept: "IT"
1470     }
1471 ]
1472
1473 },
1474 {
1475     name: "Bob",
1476     age: 25,
1477     carrer: [
1478         {
1479             salary: 1,
1480             position: "SDE",
1481             dept: "IT"
1482         },
1483         {
1484             salary: 1,
1485             position: "SDE",
1486             dept: "IT"
1487         },
1488         {
1489             salary: 1,
1490             position: "SDE",
1491             dept: "IT"
1492         }
1493     ]
1494 },
1495 {
1496     name: "Charlie",
1497     age: 35,
1498     carrer: [
1499         {
1500             salary: 1,
1501             position: "SDE",
1502             dept: "IT"
1503         },
1504         {
1505             salary: 1,
1506             position: "SDE",
1507             dept: "IT"
1508         },
1509         {
1510             salary: 1,
1511             position: "SDE",
1512             dept: "IT"
1513         }
1514     ]
1515 }
1516 ];
1517
1518 function getSalary(empDetails) {
1519     let sm = 0;
1520     for (let empDetail of empDetails) {
1521         if (empDetail.hasOwnProperty("carrer")) {
1522             for (let carrerDetails of empDetail["carrer"]) {
1523                 if (carrerDetails.hasOwnProperty("salary")) {
1524                     sm += carrerDetails["salary"];
1525                 }
1526             }
1527         }
1528     }
1529     return sm;
1530 }

```

```

1527     }
1528 }
1529 return sm;
1530 }
1531
1532 // console.log(getSalary(employeeDetails));
1533
1534
1535
1536 <!-- /**
1537  * Given an infinite object, the task is which
1538  * ends with a key false, the task is
1539  * to print a counter which
1540  * resembles the number of nested objects inside it
1541  */ -->
1542
1543 const nestedObject = {
1544   boolean: true,
1545   next: {
1546     boolean: true,
1547     next: {
1548       boolean: true,
1549       next: {
1550         boolean: true,
1551         next: {
1552           boolean: true,
1553           next: {
1554             boolean: true,
1555             next: {
1556               boolean: true,
1557               next: {
1558                 boolean: true,
1559                 next: {
1560                   boolean: true,
1561                   next: {
1562                     boolean: true,
1563                     next: {
1564                       boolean: false,
1565                       next: null
1566                     }
1567                   }
1568                 }
1569               }
1570             }
1571           }
1572         }
1573       }
1574     }
1575   }
1576 };
1577
1578 function countNestedObj(object) {
1579   let bool = false;
1580   let cnt = 0;
1581   if (object.hasOwnProperty("boolean")) {
1582     bool = object["boolean"];
1583   }
1584   let { boolean, next } = object;
1585   while (bool) {
1586     if (!boolean) {
1587       break;
1588     } else {
1589       boolean = next["boolean"];
1590       next = next["next"];
1591       cnt++;
1592     }
1593   }

```



```

1594     return cnt;
1595 }
1596
1597 // console.log(countNestedObj(nestedObject));
1598
1599
1600
1601 <!-- /**
1602  * Given an array and a chunk size , the task is to
1603  * segregate the array elements
1604  * based on chunks and return an array of
1605  * array of these chunks
1606  *
1607  * Ex: arr = [1,2,3,4,5], chunkSize = 3
1608  * o/p: [[1,2,3],[4,5]]
1609  */ -->
1610
1611 var chunk = function (arr, size) {
1612     let res = [];
1613     let i = 0;
1614     while (i < arr.length) {
1615         let chunk = arr.slice(i, size + i);
1616         res.push(chunk);
1617         i = size + i;
1618     }
1619     console.log(res);
1620 };
1621
1622 // console.log(chunk([1, 2, 3, 4, 5], 3));
1623
1624
1625 <!-- /**
1626  * Flatten n densely deep array
1627  *
1628  * Ex: arr = [1, 2, 3, [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]], n = 0
1629  * [1, 2, 3, [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]
1630  *
1631  * Ex: arr = [1, 2, 3, [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]
1632  * n = 1
1633
1634  * o/p: [1, 2, 3, 4, 5, 6, 7, 8, [9, 10, 11], 12, 13, 14, 15]
1635
1636  * Ex: arr = [[1, 2, 3], [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]
1637  * n = 2
1638
1639  * [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
1640  */ -->
1641
1642 function flatten(arr, n, currentDepth = 0) {
1643     if (n === 0 || currentDepth >= n) {
1644         return arr;
1645     }
1646     let res = [];
1647     for (let val of arr) {
1648         if (Array.isArray(val)) {
1649             res.push(...flatten(val, n, currentDepth + 1));
1650         } else {
1651             res.push(val);
1652         }
1653     }
1654     return res;
1655 }
1656
1657 function flatten1(arr, n) {
1658     return arr.flat(n);
1659 }
1660

```

```

1661
1662 <!-- /**
1663  * Given two arrays arr1 and arr2, return a new array joinedArray.
1664  * All the objects in each of the two inputs arrays will
1665  * contain an id
1666  * field that has an integer value.
1667  * joinedArray is an array formed by merging arr1 and arr2 based on
1668  * their id key.
1669  * The length of joinedArray
1670  * should be the length of unique values of id. The returned array
1671  * should be sorted
1672  * in ascending order based
1673  * on the id key.
1674  * If a given id exists in one array but not the other,
1675  * the single object with that
1676  * id should be included
1677  * in the result array without modification.
1678  * If two objects share an id, their properties should be merged
1679  * into a single object:
1680  * If a key only exists in one object, that single key-value pair
1681  * should be included in the object.
1682  * If a key is included in both objects, the value in the object
1683  * from arr2 should override
1684  * the value from arr1.
1685  *
1686  * Example 1:
1687  * Input:
1688  * arr1 = [
1689  *   {"id": 1, "x": 1},
1690  *   {"id": 2, "x": 9}
1691  * ],
1692  * arr2 = [
1693  *   {"id": 3, "x": 5}
1694  * ]
1695
1696   Output:
1697
1698   [
1699   {"id": 1, "x": 1},
1700   {"id": 2, "x": 9},
1701   {"id": 3, "x": 5}
1702   ]
1703
1704   Explanation: There are no duplicate ids so arr1 is
1705   simply concatenated with arr2.
1706
1707   Example 2:
1708
1709
1710   Input:
1711
1712   arr1 = [
1713   {"id": 1, "x": 2, "y": 3},
1714   {"id": 2, "x": 3, "y": 6}
1715   ],
1716
1717   arr2 = [
1718   {"id": 2, "x": 10, "y": 20},
1719   {"id": 3, "x": 0, "y": 0}
1720   ],
1721
1722   Output:
1723
1724   [
1725
1726
1727

```

```

1728     {"id": 1, "x": 2, "y": 3},
1729     {"id": 2, "x": 10, "y": 20},
1730     {"id": 3, "x": 0, "y": 0}
1731
1732 ]
1733
1734 Explanation: The two objects with id=1 and id=3 are
1735 included in the result array without modification.
1736 The two objects with id=2 are merged together. The
1737 keys from arr2 override the values in arr1.

```

Example 3:

Input:

```

arr1 = [
{"id": 1, "b": {"b": 94}, "v": [4, 3], "y": 48}

```

```

]
```

```

arr2 = [
{"id": 1, "b": {"c": 84}, "v": [1, 3]}

```

```

]
```

```

Output: [
{"id": 1, "b": {"c": 84}, "v": [1, 3], "y": 48}

```

```

]
```

Explanation: The two objects with id=1 are merged together.
For the keys "b" and "v"
the values from arr2 are used.
Since the key "y" only exists in arr1,
that value is taken from arr1.

```

*/ -->
```

```

function mergeObjects(arr1, arr2) {
  let mp = new Map();
  for (let item1 of arr1) {
    mp.set(item1.id, item1);
  }
  for (let item2 of arr2) {
    if (mp.has(item2.id)) {
      let obj = mp.get(item2.id);
      let mergedObj = { ...obj, ...item2 };
      mp.set(item2.id, mergedObj);
    } else {
      mp.set(item2.id, item2);
    }
  }
  mp = new Map([...mp.entries()].sort((a, b) => a[0] - b[0]));
  let res = [];
  for (const [key, value] of mp) {
    res.push(value);
  }
  return res;
};

```

<----->

****Functions****

```

<!-- /**

```

```

1795     * Pass n number of arguments to a function and
1796     * return the sum of all of them.
1797     *
1798     * This can be done with the help of rest operators.
1799     */ -->
1800
1801 function addNumbers(...nums) {
1802     let sm = 0;
1803     for (let val of nums) {
1804         sm += val;
1805     }
1806     return sm;
1807 }
1808
1809 // console.log(addNumbers(1, 2, 3, 4, 5));
1810
1811 <!-- /**
1812     * Treating functions like variables
1813     */ -->
1814
1815 let addMultiNumbers = function (...nums) {
1816     let sm = 0;
1817     for (let val of nums) {
1818         sm += val;
1819     }
1820     return sm;
1821 }
1822
1823 // console.log(addMultiNumbers(1, 2, 3, 4, 5));
1824
1825
1826 <!-- /**
1827     * Difference between normal function declaration and
1828     * using variable to hold the function is
1829     *
1830     * 1. Syntax is different
1831     * 2. Normal function undergoes hoisting i,e if i say
1832     *
1833     * addNumbers(1,2,3);
1834     * function addNumbers(...nums) {
1835     *     let sm = 0;
1836     *     for (let val of nums) {
1837     *         sm += val;
1838     *     }
1839     *     return sm;
1840     * }
1841     *
1842     * Clearly we have called the function first and then declared it,
1843     * since function are hoisted
1844     * and thus there was no issue.
1845     * Hoisting means taking the declaration to top of the their scope.
1846     *
1847     * Variable associated function works based upon the scope of
1848     * hoisting of the variable
1849     * type used ex: var, let or const.
1850     * See different cases for better understanding.
1851     */ -->
1852
1853
1854 <!-- /**
1855     * Case1:
1856     */ -->
1857
1858 // console.log(normal(2, 3)); // no problem
1859 function normal(a, b) {
1860     return a + b;
1861 }

```

```

1862
1863
1864 <!-- /**
1865  * Case2:
1866  * This will have error fun is not defined because after
1867  * hoisting the code will look like
1868  * var fun;
1869  * console.log(fun(2,3))
1870  * fun = function (a, b) {
1871  *   return a + b;
1872  * };
1873  * Clearly error is thrown in line 2 console.log(undefined(2,3));
1874  */ -->
1875
1876 // console.log(fun(2, 3));
1877 var fun = function (a, b) {
1878   return a + b;
1879 }
1880
1881
1882 <!-- /**
1883  * Case3:
1884  * Using let and const
1885  * Let and const are also hoisted just like var however
1886  * they remain in Temporal dead zone and are not initialized
1887  * till its declaration is encountered in code.
1888  * So when code comes to console.log(fun1), fun1 was present
1889  * in temporal dead zone and was not
1890  * initialized and thus error will be
1891  * thrown that can't access fun1 before initialization.
1892  */ -->
1893
1894 // console.log(fun1(2, 3, 4));
1895 let fun1 = function (...num) {
1896   return num
1897 }
1898
1899
1900 <!-- /**
1901  * Closures:
1902  * In cases of having function inside a function,
1903  * the inner function will have access to the variables of
1904  * outer function as for
1905  * inner function these variables of
1906  * outer function has global scope but the outer function can
1907  * not have access to variables
1908  * of inner function as
1909  * the inner function variables has block scope.
1910  */ -->
1911
1912 function one() {
1913   let variableOne = "outerVariable";
1914   function two() {
1915     let variableTwo = "innerVariable";
1916     console.log(variableOne);
1917   }
1918   two();
1919   console.log(variableTwo); // problematic as this is inner variable and has
1920                             only block scope within function two.
1921 }
1922
1923 // one();
1924
1925
1926 <!-- /**
1927  * Function inside an object and this keyword
1928  *

```

```

1929  * function which is used as value to property
1930  * welcome of obj is an anonymous
1931  * function as its without any name.
1932  *
1933  * In obj the context is having property like name, email.
1934  * this keyword refers to the current instance of obj
1935  * or the current context of obj.
1936  */ -->
1937
1938  let obj = {
1939      name: "Gaurav",
1940      email: "xyz@abc.com",
1941      welcome: function () { // its an anonymous function i,e without any name
1942          return `Hello ${this.name}`;
1943      }
1944  }
1945
1946  // console.log(obj.welcome());
1947
1948
1949  <!-- /**
1950   * This keyword in browser vs this keyword in node env
1951   *
1952   * console.log(this) gives the current context.
1953   *
1954   * In browser the current context or the current instance is the window so
1955   * window object is shown where as in node env
1956   * the current context or current instance in an empty object.
1957   */ -->
1958
1959  // console.log(this);
1960
1961  function thisTesting() {
1962      const userName = "gaurav";
1963      console.log(this); // gives a lot of global object prop associated
1964                          with function but does not
1965                          include any kind of variables associated with function
1966
1967      console.log(this.userName); // undefined as we can only get props of
1968                                  objects through this and not of functions.
1969  }
1970
1971  // thisTesting();
1972
1973
1974
1975  <!-- /**
1976   * Arrow functions
1977   */ -->
1978
1979  let arrow = () => {
1980      let name = "gaurav";
1981      console.log(this.name); // undefined same concept as that of a
1982                              normal function
1983
1984      return `we are using arrow function`;
1985  }
1986
1987
1988  <!-- /**
1989   * Case: Implicit return arrow function
1990   *
1991   * if we only have one liner logic then we can use
1992   * () and avoid return statement
1993   * however if we use {} then a return statement is must.
1994   */ -->
1995

```

```

1996 let arrow1 = (num1, num2) => (num1 + num2);
1997
1998
1999 <!-- /**
2000  * IIFE (Immediately invoked function executions)
2001  *
2002  * IIFE are self executed functions and
2003  * since variables declared inside the
2004  * scope of the function (let & const),
2005  * will have scope within it so IIFE are used to
2006  * avoid global variables pollutions,
2007  * and used in scenarios where
2008  * the function need to be invoked immediately.
2009  */ -->
2010
2011 (function IIFE() { // named IIFE
2012     console.log(`IIFE syntax as normal function`);
2013 })();
2014
2015
2016 /**IIFE syntax as arrow function */
2017
2018 (() => {
2019     console.log(`IIFE syntax for arrow function`)
2020 })();
2021
2022
2023 /**IIFE syntax for implicit return arrow function */
2024
2025 (() => (console.log(`IIFE syntax for implicit returned arrow function`)))();
2026
2027
2028 /**IIFE syntax with function parameters */
2029
2030 ((name) => {
2031     console.log(`Hello my name is ${name}`);
2032 })("Gaurav");
2033
2034
2035
2036 <!-- /**
2037  * Callback functions:
2038  *
2039  * A callback function is a function that is passed
2040  * as an argument to another function
2041  * and is intended to be executed later or immediately,
2042  * either synchronously or asynchronously.
2043  *
2044  */ -->
2045
2046
2047 <!-- /**
2048  * Here the arrow function passed inside as an
2049  * argument to forEach is a callback function.
2050  */ -->
2051
2052 let arr = [1, 2, 3, 4, 5];
2053 arr.forEach((val) => {
2054     console.log(val);
2055 })
2056
2057 function print(val) {
2058     console.log(val);
2059 }
2060
2061 // arr.forEach(print); // passed reference of a
2062                      callback function print inside for each

```

```
2063
2064
2065 <!-- /**
2066  * Here the callback function of print which is passed
2067  * as an argument to forEach
2068  * function is executed immediately
2069  * in a synchronous way, that is code is getting executed
2070  * for print and don't
2071  * have to wait to print to get completed.
2072  *
2073  * We can also take an example of asynchronous callback.
2074  *
2075  * So returnResp takes a callback function which it
2076  * calls back once it
2077  * completes its execution.
2078  * so we call returnResp with callback function (the function which
2079  * returnResp needs to callBack)
2080  * once its completes
2081  * execution and is getResponse.
2082  *
2083  * So once in 2 secs returnResp complete execution, it call
2084  * back to getResponse and thus the o/p is
2085  *
2086  * Call callback function after 2 secs
2087  * Call back function called successfully
2088 */ -->
2089
2090 function returnResp(callback) {
2091     setTimeout(() => {
2092         console.log("Call callback function after 2 secs");
2093         callback();
2094     }, 200)
2095 }
2096
2097 function getResponse() {
2098     console.log("Call back function called successfully");
2099 }
2100
2101 returnResp(getResponse);
2102
2103
2104
2105 let obj1 = [
2106     {
2107         language: "JS",
2108         description: "Javascript"
2109     },
2110     {
2111         language: "Java",
2112         description: "Java"
2113     },
2114     {
2115         language: "python",
2116         description: "py"
2117     }
2118 ]
2119
2120
2121
2122
2123 for (let val of obj1) {
2124     console.log(`${val.language} has description ${val.description}`);
2125 }
2126
2127
2128
2129 <
```



```

----->
2130
2131 **Loops**
2132
2133 <!-- /**
2134  *
2135  * Break statement is used to break and terminate the loop,
2136  * whereas continue statement is used to
2137  * skip that particular iteration and jump to next iteration.
2138  */ -->
2139
2140 for (let i = 0; i < 20; i++) {
2141     if (i === 5) {
2142         continue
2143     }
2144     if (i === 10) {
2145         break;
2146     }
2147     // console.log(i);
2148 }
2149
2150
2151
2152 <!-- /**
2153  * while loop
2154  */ -->
2155
2156 let i = 0;
2157 while (i <= 10) {
2158     // console.log(i);
2159     i++;
2160 }
2161
2162
2163
2164 <!-- /**
2165  * Do while loop
2166  */ -->
2167
2168 let j = 11;
2169 do {
2170     // console.log(j);
2171     j++;
2172 } while (j <= 10);
2173
2174
2175
2176
2177 <!-- /**For of loops */ -->
2178
2179
2180 let arr = [1, 2, 3, 4, 5]
2181
2182 for (let val of arr) {
2183     // console.log(val);
2184 }
2185
2186 let obj = {
2187     name: "Gaurav",
2188     age: 27,
2189     number: 9354377832
2190 }
2191
2192 // for (let key of obj) { // objects are not iterable and can't be
2193 //     looped in using for of loop
2194 //     // console.log(obj[key]);

```

```

2195 // }
2196
2197 let mp = new Map();
2198 mp.set(1, "India");
2199 mp.set(2, "US");
2200
2201 for (const [key, value] of mp) {
2202     // console.log(key + '-> ' + value);
2203 }
2204
2205
2206
2207 <!-- /**
2208  *
2209  * For in loop
2210  */ -->
2211
2212 obj = {
2213     name: "Gaurav",
2214     age: 27,
2215     number: 9354377832
2216 }
2217
2218 for (let keys in obj) {
2219     // console.log(keys + '-> ' + obj[keys]);
2220 }
2221
2222
2223 arr = [1, 2, 3, 4, 5];
2224 for (let i in arr) {
2225     // console.log(arr[i]);
2226 }
2227
2228 // for(let key in mp){ // it wont work as map is an iterable object and
2229 //                      needs for of.
2230 //     console.log('abc');
2231 //     console.log(key);
2232 // }
2233
2234
2235 <!-- /**
2236  * The main difference between for in loop and for of loop is that,
2237  * for of loop gives values directly, whereas for in loop gives the keys of
2238  * which we are iterating.
2239  *
2240  * for of loop is specifically designed to iterate over the values
2241  * of iterable objects.
2242  * for in loop is designed to iterate over the enumerable properties
2243  * of an object,
2244  *
2245  * For ex: for of loop failed in object because keys in
2246  * object are random arbitrary
2247  * user given and thus to get
2248  * value of an object it needs keys but keys are also not
2249  * available whereas in for in loop,
2250  * we have access to keys
2251  * so we can iterate using keys in obj and then use these
2252  * keys to get obj.
2253  *
2254  * For of passes in array as it can get hold of values,
2255  * for in loop also passes in array as array is also an
2256  * object with non arbitrary index
2257  * and specif index i,e 0,1,2,3,4....
2258  * so a for in loop will give index in array which can be
2259  * used to get value.
2260  */ -->
2261

```

```
2262
2263 <----->
2264
2265 **Filter map & reduce**
2266
2267 <!-- /**
2268  * For each loop;
2269  *
2270  * For each loop is used to loop over the items and take a call back
2271  * function which is capable
2272  * of handling three parameters i,e val,index and arr.
2273  *
2274  * Val is arbitrary name for the inputs, index can be considered as index
2275  * values and arr is the array itself.
2276  * For each loop does not return anything and its return type is void.
2277  */ -->
2278
2279 let arr = [1, 2, 3, 4, 5];
2280 arr.forEach((val, index, arr) => {
2281   // console.log(val, index, arr);
2282 })
2283
2284
2285 <!-- /**Filters
2286  *
2287  * Filter function in array is used to filter out some elements
2288  * based upon a certain
2289  * condition and it also takes a callback function as argument
2290  * and returns a number [];
2291  */ -->
2292
2293 let res = arr.filter((val) => {
2294   return val > 2;
2295 })
2296
2297 // console.log(res);
2298
2299 let obj = [
2300   {
2301     name: "Gaurav",
2302     role: "SDE1"
2303   },
2304   {
2305     name: "Poonam",
2306     role: "SDE1"
2307   },
2308   {
2309     name: "Brijesh",
2310     role: "SDE2"
2311   },
2312   {
2313     name: "Pankaj",
2314     role: "SDE2"
2315   },
2316   {
2317     name: "Vinod",
2318     role: "SDE2"
2319   }
2320 ]
2321
2322 let res1 = obj.filter((item) => {
2323   return item.role === "SDE2";
2324 })
2325
2326 // console.log(res1);
2327
2328
```

```

2329 <!-- /**
2330  * Map function in array is used to transform elements in the array.
2331  * It takes an callback function as an argument parameter
2332  * and also returns a nums []
2333  */ -->
2334
2335 let res2 = arr.map((item) => {
2336     return item * 2;
2337 })
2338
2339 // console.log(res2);
2340
2341
2342
2343 <!-- /**
2344  * Chaining of methods
2345  */ -->
2346
2347 let res3 = arr.map((val) => {
2348     return val * 10;
2349 }).map((val) => {
2350     return val + 1;
2351 }).filter((val) => {
2352     return val % 11 === 0;
2353 })
2354
2355 // console.log(res3);
2356
2357
2358 <!-- /**Reduce methods
2359  *
2360  * Find sum of elements in array
2361  *
2362  * Reduce methods takes a callback as an function argument ,
2363  * this call back function needs
2364  * a prevValue which is a number and a currentValue which
2365  * is the current iterated value
2366  *
2367  * So in our case of finding sum of elements of array, we used
2368  * prevValue as 0 and the currentValue will be
2369  * iteration over every element of the array.
2370  *
2371  * we return prevValue + currentValue thus after every iteration
2372  * the prevValue keeps on updating itself
2373  * as prevValue + currentValue
2374  */ -->
2375
2376 let initial = 0;
2377 let res4 = arr.reduce((prevValue, currentValue) => {
2378     return prevValue + currentValue;
2379 }, initial)
2380
2381 // console.log(res4);
2382
2383
2384 <!-- /**
2385  * Add all prices in shopping cart using reduce
2386  *
2387  * so we have two parameters in callback function argument
2388  * of reduce which are prev and current,
2389  * prev is the previously obtained value which is number and
2390  * current is the current iteration.
2391  *
2392  * So we return prev + current["price"] where initial value
2393  * of prev we set to 0.
2394  */ -->
2395

```

```

2396 let cart = [
2397   {
2398     course: "js",
2399     price: 100
2400   },
2401   {
2402     course: "DSA",
2403     price: 200
2404   },
2405   {
2406     course: "ts",
2407     price: 300
2408   },
2409 ]
2410
2411 let sum = cart.reduce((prev, current) => {
2412   return prev + current["price"];
2413 }, 0);
2414
2415 console.log(sum);
2416
2417
2418

```

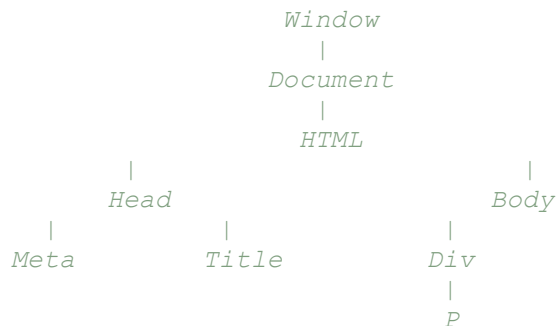
<----->

****DOM (Document object model)****

```

2423 <!--
2424 Lets understand DOM, its called as document object model.
2425 An html is composed of objects
2426 of a document and thus collectively
2427 they are document object model.
2428
2429 Lets see a DOM diagram for the below code.
2430

```



```

2443 So we can manipulate DOM according to our needs and this concept
2444 is simply DOM manipulation.

```

-->

```

2448 <!DOCTYPE html>
2449 <html lang="en">
2450
2451 <head>
2452   <meta charset="UTF-8">
2453   <title>Document</title>
2454 </head>
2455
2456 <body>
2457   <div>
2458     <h1>E-Commerce</h1>
2459     <p>This is a shopping paradise</p>
2460   </div>
2461
2462 </body>

```

```

2463
2464 </html>
2465
2466
2467 <----->
2468
2469 **DOM manipulation**
2470
2471 <!--HTML>
2472
2473 <!DOCTYPE html>
2474 <html lang="en">
2475
2476 <head>
2477   <meta charset="UTF-8">
2478   <title>Document</title>
2479   <!--Kind of a custom css/scss file content -->
2480   <style>
2481     .bg-class {
2482       background-color: #212121;
2483       color: #ffff;
2484     }
2485   </style>
2486
2487 <body class="bg-class">
2488   <div>
2489     <h1 id="title" class="header">E-Commerce</h1>
2490     <p id="description">This is a shopping paradise
2491       <span style="display: none;">for women</span>
2492     </p>
2493     <h2 id="sub-header1">Lorem.</h2>
2494     <h2 id="sub-header2">Lorem ipsum</h2>
2495     <h2 id="sub-header3" class="header3">Lorem, ipsum dolor.</h2>
2496     <ul>
2497       <li class="list-item">Checker1</li>
2498       <li class="list-item">Checker2</li>
2499       <li class="list-item">Checker3</li>
2500     </ul>
2501     <div class="parent">
2502       <div class="child1">Monday</div>
2503       <div class="child2">Tuesday</div>
2504       <div class="child3">Wednesday</div>
2505       <div class="child4">Thursday</div>
2506       <div class="child5">Friday</div>
2507       <div class="child6">Saturday</div>
2508       <div class="child7">Sunday</div>
2509     </div>
2510     <ul class="languageList">
2511       <li class="language1">js</li>
2512     </ul>
2513   </div>
2514   <!--Script or js file is added here-->
2515   <script src="02.DomManipulation.js"></script>
2516 </body>
2517
2518 </html>
2519
2520
2521 <!--SCRIPT>
2522
2523 <!-- /**Used to get the element by the id and get
2524 stored in title variable */ -->
2525 const title = document.getElementById('title');
2526
2527
2528 <!-- /**Used to get the element by class name and returns
2529 a html collection */ -->

```

```

2530 let item = document.getElementsByClassName('list-item');
2531
2532 //converting html collection into an array
2533 let itemArray = [...item];
2534 itemArray.forEach((li) => {
2535     li.style["font-size"] = "20px";
2536 })
2537
2538
2539 <!-- /**Used to get the attribute class value of an element
2540 which we are fetching via id */ -->
2541 console.log(document.getElementById('title').getAttribute('class'));
2542
2543
2544 <!-- /**Used to set attribute class over an element
2545 which we got by id. Here two
2546 classes we have set i,e para1 p1.
2547 * If only one class would have been set then it would have over
2548 * rided the existing class.
2549 */ -->
2550 document.getElementById('description').setAttribute('class', 'para1 p1');
2551
2552
2553 <!-- /**Used to set styling on element stored
2554 in title variable. */ -->
2555 title.style.background = "green";
2556 title.style.borderRadius = "5px";
2557 title.style.padding = "5px";
2558
2559
2560 <!-- /** Difference between innerText and textContent,
2561 *
2562 * suppose in html "description" is the id
2563 * of a p tag which is having some text
2564 * and inside there is also a span which is
2565 * having some more text.
2566 * Now in the span we have given an style of
2567 * display none, so this addition more
2568 * text inside span will not be
2569 * visible.
2570 *
2571 * So if we do document.getElementById("description").
2572 * innerText so it will give
2573 * only the text which will be visible after
2574 * the styles have been applied.
2575 *
2576 * however document.getElementById("description").
2577 * textContent will even show the
2578 * text within the span which will not be shown
2579 * on browser due to styling.
2580 */ -->
2581
2582 const description = document.getElementById("description");
2583
2584 console.log(description.innerText);
2585 console.log(description.textContent);
2586
2587
2588
2589 <!-- /**Used to get the innerHtml */ -->
2590 console.log(description.innerHTML);
2591
2592
2593
2594 <!-- /**Query selector is used to get elements
2595 based on id or class or
2596 even direct tag names.

```

```

2597     * In case of presence of multiple elements it will
2598     * give you the first by default
2599     */ -->
2600
2601 <!-- //selecting query by tag name directly -->
2602 console.log(document.querySelector('h2'));
2603
2604 <!-- //selecting query by id -->
2605 console.log(document.querySelector('#sub-header2'));
2606
2607 <!-- //selecting query by class name -->
2608 console.log(document.querySelector(".header3"));
2609
2610 let ul = document.querySelector('ul');
2611 let li = ul.querySelector('li');
2612 li.style.color = "red";
2613
2614
2615
2616 <!-- /**Query selector all is used to give a nodeList of
2617     all the elements which are
2618     getting matched based
2619     * upon id, class or even the tag name itself.
2620     *
2621     * Node list may somewhat looks similar to array but
2622     * however its not as it does
2623     * not include some basic array
2624     * operations like map, filter , reduce etc and thus
2625     * its different
2626     */ -->
2627
2628 let myLi = document.querySelectorAll('li');
2629 myLi[0].style.color = "red";
2630 myLi[1].style.color = "green";
2631 myLi[2].style.color = "blue";
2632
2633 myLi.forEach((item) => {
2634     item.style.backgroundColor = 'white';
2635 })
2636
2637 //conversion of node list to array
2638 myLiArray = [...myLi];
2639 console.log(myLiArray);
2640
2641
2642
2643 <!-- /**Accessing DOM elements using
2644     relationships */ -->
2645
2646 let parentObj = document.querySelector('.parent');
2647
2648 <!-- //gives html collection from parentObj -->
2649 let childHtmlCollection = parentObj.children;
2650
2651 <!-- //iterating over html collection
2652     using for of loop -->
2653 for (let item of childHtmlCollection) {
2654     item.style.color = "orange";
2655     item.style.padding = "2px";
2656 }
2657
2658 <!-- //getting the first child from an parentObj -->
2659 let firstChild = parentObj.firstElementChild;
2660 let lastChild = parentObj.lastElementChild;
2661
2662 <!-- //reaching the parent from children -->
2663 parentObj = firstChild.parentElement;

```



```
2664
2665 <!-- //reaching siblings from children -->
2666 let sibling = firstChild.nextElementSibling;
2667
2668
2669
2670 <!-- /**
2671  * Child nodes are nodes which are present inside an element for ex:
2672  * if we see
2673  * parent class in html then we can say
2674  * classes like child1, child2 .... etc are its child nodes,
2675  * however enters, comments etc
2676  * are also taken into consideration
2677  * while considering child nodes.
2678  */ -->
2679
2680 let mainNode = document.querySelector(".parent");
2681 console.log(mainNode.childNodes);
2682
2683
2684
2685 <!-- /**Creating a element in document */ -->
2686
2687 <!-- // creating a new div and giving it some id, class
2688 name some styling and some -->
2689 text and then appending it to document.
2690
2691 let div = document.createElement("div");
2692 div.className = "customDiv";
2693 div.id = Math.round(Math.random() * 10 + 1);
2694 div.style.backgroundColor = "blue";
2695 div.style.color = "black";
2696 let textNode = document.createTextNode("This is a custom div via script")
2697 div.appendChild(textNode);
2698
2699 document.body.appendChild(div);
2700
2701
2702 <!-- // appending more li into a ul via a function which
2703 takes langName as parameter and use
2704 that as text in li. -->
2705
2706 document.querySelector(".language1").style.background = "#212121"
2707
2708 function createListElements(languageName) {
2709     let ulLanguageList = document.querySelector(".languageList");
2710     let li = document.createElement('li');
2711     let textNode = document.createTextNode(languageName);
2712     li.appendChild(textNode);
2713     ulLanguageList.appendChild(li);
2714 }
2715
2716 createListElements("java");
2717 createListElements("typescript");
2718 createListElements("python");
2719
2720
2721
2722
2723 <!-- /**Editing an existing element in document */ -->
2724
2725 <!-- //editing an existing list item in ul via creating a new li
2726 and then replacing the old with new. -->
2727 let individualLang = document.querySelector('.languageList :nth-child(2)');
2728 let newLi = document.createElement('li');
2729 newLi.appendChild(document.createTextNode("maje lo"));
2730 individualLang.replaceWith(newLi);
```

```

2731
2732
2733 <!-- // editing an list item by directly changing the
2734 outerHTML of it. -->
2735 let firstChildList = document.querySelector('.languageList :first-child');
2736 firstChildList.outerHTML = '<li>changed and edited item</li>'
2737
2738
2739
2740 <!-- /**Deleting an existing element in document */ -->
2741 let lastChildList = document.querySelector('.languageList :last-child');
2742 lastChildList.remove();
2743
2744
2745
2746 <-----
>
2747
2748 **Events**
2749
2750 <!--HTML>
2751
2752 <!DOCTYPE html>
2753 <html lang="en">
2754
2755 <head>
2756     <meta charset="UTF-8">
2757     <meta name="viewport" content="width=device-width, initial-scale=1.0">
2758     <title>Document</title>
2759 </head>
2760
2761 <body>
2762     <ul id="cityList">
2763     </ul>
2764 </body>
2765 <script src="01_Events.js"></script>
2766
2767 </html>
2768
2769
2770 <!--SCRIPT>
2771
2772 let cities = [
2773     {
2774         name: "Delhi",
2775         id: "city1"
2776     },
2777     {
2778         name: "Srinagar",
2779         id: "city2"
2780     },
2781     {
2782         name: "Bangalore",
2783         id: "city3"
2784     },
2785     {
2786         name: "Gurugram",
2787         id: "city4"
2788     }
2789 ]
2790
2791
2792 <!-- /**Click event on list items */ -->
2793
2794 function addCities(cities) {
2795     let parentUl = document.querySelector('#cityList');
2796     cities.forEach((item) => {

```

```

2797         let element = document.createElement('li');
2798         element.textContent = item.name;
2799         element.id = item.id;
2800         element.style.cursor = "pointer";
2801
2802         parentUl.appendChild(element);
2803
2804         element.addEventListener('click', () => {
2805             alert(`welcome to details of ${item.name}`)
2806         });
2807     })
2808 }
2809
2810 addCities(cities);
2811
2812
2813
2814 <!-- /**
2815  * The above is a example of Event propagation, in which we
2816  * handled click event in the same way
2817  * there can be multiple events which
2818  * can be handled like keyboard up, down etc.
2819  *
2820  * Now there are two main types of event propagation
2821  * 1) Bubbling
2822  * 2) Capturing
2823  *
2824  * when we said:
2825  * element.addEventListener('click', () => {
2826  *     // write some code
2827  * },false)
2828
2829     this boolean parameter false is default false
2830     and refers to event bubbling.
2831
2832     Lets understand this with an example
2833
2834     we had a parent which was ul and had child like li
2835     when we passed the parameter false in addEventListener,
2836     then its called event bubbling
2837     and flow goes from child -> parent -> grandparent.
2838
2839     So in below example we had event listener of click on
2840     both parent and child.
2841     When parent is click is fine , parent click event got
2842     propagated, when child is clicked
2843     the in case of event bubbling first child event gets
2844     triggered and then parent
2845     event will get triggered.
2846
2847     How ever in case of event capturing if we would have
2848     given true as parameter in
2849     both the cases, then
2850     first parent event would have propagated and then child
2851     event will get propagated.
2852  *
2853  */ -->
2854
2855 let parentUl = document.querySelector('#cityList');
2856 parentUl.addEventListener('click', () => {
2857     console.log('parent ul clicked')
2858 }, false); // mark true for event capturing
2859
2860 let element = document.querySelector('#city2');
2861 element.addEventListener('click', () => {
2862     console.log('child item clicked')
2863 }, false) // mark true for event capturing

```

```

2864
2865
2866 <!-- /**Stop propagation, if we don't want that event to
2867 propagate further either via
2868 bubbling or capturing then we
2869 * do a stop propagation
2870 */ -->
2871
2872 element = document.querySelector('#city3');
2873 element.addEventListener('click', (e) => {
2874     console.log('child item clicked');
2875     e.stopPropagation();
2876 }) // now in this case the event will not bubble up till the parent
2877
2878
2879 <!-- /**Make gurgaon list item disappear /delete**/ -->
2880 let gurgaon = document.querySelector('#city4');
2881 gurgaon.addEventListener('click', (e) => {
2882     gurgaon.remove();
2883     gurgaon.stopPropagation();
2884 })
2885
2886
2887
2888

```

<

----->

```

2889
2890 **Async js**
2891
2892 <!-- /**
2893  * Js is fundamentally synchronous in nature, means it can only
2894  * perform task one by one and is
2895  * single threaded in nature.
2896  * However this resembles is only in context of js engines.
2897  *
2898  * Generally js now as days aren't used with its simply engines
2899  * however either uses web api's/browser
2900  * or run time env like node which
2901  * gives it a feel of asynchronous like promises, fetch api's,
2902  * callbacks, setTimeouts, setInterval etc.
2903  *
2904  * Lets understand how js becomes asynchronous, so
2905  * if we use web api's/browser
2906  * that have DOM api's, setTimeout, setInterval etc.
2907  * (can be seen in windows object)
2908  *
2909  * Or uses node js env which doesn't have DOM api'
2910  * and rest everything so how flow happens.
2911  *
2912  *
2913  * Once global execution context is created and function
2914  * calls start getting executed, these
2915  * function calls are pushed to call stacks and formulate
2916  * a functional execution context.
2917  *
2918  * Say we have a function which has a setTime ,
2919  * setTime is asynchronous in nature,
2920  * so such is pushed to webApi,
2921  * inside webApi there is a register callback which pushes
2922  * this call to a taskQueue, once
2923  * it is popped from the task queue , it is
2924  * again pushed back into the call stack of js global
2925  * execution context and thus is performed.
2926  * This pushing and management happens via
2927  * event loop.
2928  *

```

```

2929     * In this way async calls are handled.
2930     *
2931     * In case of fetch this function when passed to web api, and sent to
2932     * register callback is sent to a different queue which is different
2933     * from task queue and is having higher priority while the rest process remains same.
2934     *
2935     * Consider diagram
2936     * ![alt text](image.png)
2937     *
2938     *
2939     */ -->
2940
2941                                     <!--HTML>
2942
2943 <!DOCTYPE html>
2944 <html lang="en">
2945
2946 <head>
2947     <meta charset="UTF-8">
2948     <meta name="viewport" content="width=device-width, initial-scale=1.0">
2949     <title>Document</title>
2950 </head>
2951
2952 <body class="body">
2953     <h1>We are learning async js</h1>
2954     <button id="stop">Stop</button>
2955
2956     <h2>Start game of changing background</h2>
2957     <button id="startBgGame">start</button>
2958     <button id="stopBgGame"> stop</button>
2959 </body>
2960 <script src="asyncJs.js"></script>
2961
2962 </html>
2963
2964 </html>
2965
2966                                     <!--SCRIPT>
2967
2968 const hexCodeValid = 'ABCDEF123456789';
2969
2970 <!-- /**
2971     * Time out function of web api/browser can be seen in
2972     * browser prototype,
2973     * which is changing the inner html of
2974     * h1 tag.
2975     */ -->
2976
2977 const timer = setTimeout(() => {
2978     document.querySelector('h1').innerHTML = "Gaurav's learning of js";
2979 }, 2000);
2980
2981
2982 <!-- /**
2983     * Clear time out function to stop the setTimeOutFunction if not
2984     * executed within its given time frame and
2985     * button is clicked.
2986     */ -->
2987 document.querySelector('#stop').addEventListener('click', () => {
2988     clearTimeout(timer);
2989 })
2990
2991
2992
2993
2994 const body = document.querySelector('.body');
2995 let interval;

```

```

2996
2997 <!-- /**
2998  * Setting an interval using setInterval inside a
2999  * function and taking the
3000  * reference of that
3001  * function in intervalGame and taking the reference
3002  * of this setInterval inside
3003  * a variable interval.
3004  */ -->
3005
3006 const intervalGame = function () {
3007     let color;
3008     interval = setInterval(() => {
3009         color = '';
3010         for (let i = 0; i < 6; i++) {
3011             color += hexCodeValid[Math.floor(Math.random() * 14)];
3012         }
3013         body.style.backgroundColor = `#${color}`;
3014         console.log(color);
3015     }, 1000);
3016 }
3017
3018 <!-- /**Use a event of click and pass the
3019     main function's
3020     reference for this event
3021     in which background color
3022     * start changing on interval of 1 sec
3023     * once button is clicked
3024 */ -->
3025
3026 document.querySelector('#startBgGame').addEventListener('click', intervalGame);
3027
3028
3029 <!-- /**Use a button to stop the changing of background once
3030 the button is clicked by clearInterval
3031 * and passing reference of the setInterval function.
3032 */ -->
3033
3034 document.querySelector('#stopBgGame').addEventListener('click', () => {
3035     clearInterval(interval);
3036     body.style.backgroundColor = 'white'
3037 })
3038
3039
3040 <----->
3041
3042 **Promises**
3043
3044
3045 <!-- /**
3046  * 1.
3047  *
3048  * Promises are object in js which is used to handle
3049  * asynchronous operation in js,
3050  * It takes one parameter as a callBack function which
3051  * has 2 parameters i,e
3052  * resolve and reject.
3053  *
3054  * In order to consume a promise we can use (then)
3055  * which is associated with the
3056  * resolved state of the promise which
3057  * we are dealing with.
3058  */ -->
3059
3060 let promise1 = new Promise((resolve, reject) => {
3061     setTimeout(() => {
3062         console.log("Async function completed");

```

```

3063         resolve();
3064     }, 1000)
3065 })
3066
3067 promise1.then(() => {
3068     console.log("Promise is resolved");
3069 })
3070
3071
3072 <!-- /**
3073  * 2.
3074  *
3075  * Getting some data through asynchronous process
3076  * and then setting it to a promise
3077  * and consuming that data.
3078  *
3079  * Here instead of placing the promise inside a variable
3080  * and then consuming it we have
3081  * simply, placed a (then) directly
3082  * with the promise, the place where promise is resolved
3083  * is where we are setting the data which
3084  * needs to be consumed and then
3085  * simply (then) , which takes a callback we are saying res as a
3086  * param and this res can be used
3087  * to consume data which is sent by
3088  * the resolve state of the promise.
3089  */ -->
3090
3091 new Promise((resolve, reject) => {
3092     setTimeout(() => {
3093         resolve({ userName: "bhatt", mobileNumber: "987654xxxx" });
3094     }, 1000)
3095 }).then((res) => {
3096     console.log(res);
3097 })
3098
3099
3100 <!-- /**
3101  * 3.
3102  *
3103  * Catching errors in promises.
3104  *
3105  * Errors while promise creation is created via reject
3106  * state where we can pass the
3107  * error which we want
3108  * to consume in case of error.
3109  *
3110  * Like then and resolve are associated to each other,
3111  * in the same way reject and catch are
3112  * associated to each other.
3113  * Catch also takes an callback as an argument and it can
3114  * have a parameter which is sended via
3115  * reject and can be accessed inside
3116  * the consumption.
3117  */ -->
3118
3119 new Promise((resolve, reject) => {
3120     let error = true;
3121     setTimeout(() => {
3122         if (!error) {
3123             resolve({ userName: "bhatt", mobileNumber: "987654xxxx" });
3124         } else {
3125             reject("Oops something went wrong");
3126         }
3127     }, 1000)
3128 }).then((res) => {
3129     console.log(res);

```

```

3130 }).catch((err) => {
3131     console.log(err);
3132 })
3133
3134
3135 <!-- /**
3136  * 4.
3137  *
3138  * Chaining in promises
3139  *
3140  * The res.userName returned via first then is
3141  * consumed by the second
3142  * then and further it logged
3143  * that returned value, such phenomena is called chaining.
3144  */ -->
3145
3146 new Promise((resolve, reject) => {
3147     let err = false;
3148     setTimeout(() => {
3149         if (!err) {
3150             resolve({ userName: "bhatt", mobileNumber: "987654xxxx" })
3151         } else {
3152             reject("Oops something went wrong");
3153         }
3154     }, 1000)
3155 }).then((res) => {
3156     return res.userName;
3157 }).then((userName) => {
3158     console.log(` Hello ${userName} to the application`)
3159 }).catch((error) => {
3160     console.log(error);
3161 })
3162
3163
3164 <!-- /**
3165  * 5.
3166  *
3167  * Finally in promises.
3168  *
3169  * finally is always executed either is promise is
3170  * rejected or resolved.
3171  * Thus finally can be used to say do some clean up stuff.
3172  */ -->
3173
3174 new Promise((resolve, reject) => {
3175     let err = true;
3176     setTimeout(() => {
3177         if (!err) {
3178             resolve({ userName: "bhatt", mobileNumber: "987654xxxx" })
3179         } else {
3180             reject("Oops something went wrong in finally promise example");
3181         }
3182     }, 1000)
3183 }).then((res) => {
3184     return res.userName;
3185 }).then((userName) => {
3186     console.log(` Hello ${userName} to the application`)
3187 }).catch((error) => {
3188     console.log(error);
3189 }).finally(() => {
3190     console.log('It will be executed either promise is resolved or rejected.')
3191 })
3192
3193
3194 <!-- /**
3195  * 6.
3196  *

```



```

3197     * Consuming promises using async and await and not
3198     * using then and catch.
3199     */ -->
3200
3201     let promiseA = new Promise((resolve, reject) => {
3202         let err = true;
3203         setTimeout(() => {
3204             if (!err) {
3205                 resolve({ userName: "bhattuu", mobileNumber: "987654XXXX" });
3206             } else {
3207                 reject('Oops something went wrong in consuming promise via async and await')
3208             }
3209         }, 1000)
3210     })
3211
3212
3213     async function consumePromise() {
3214         try {
3215             let response = await promiseA;
3216             console.log(`Hello ${response.userName} to the application`);
3217         } catch (err) {
3218             console.log(err);
3219         }
3220     }
3221
3222     consumePromise();
3223
3224
3225     <
-----
>
3226
3227     **Fetch**
3228
3229     <!-- /**
3230      * Fetch is js is used to make some network calls and
3231      * fetch data from the server.
3232      * It also gives a promise and thus this promise needs
3233      * to be consumed.
3234      */ -->
3235
3236     <!-- /**
3237      * 1. Consumption of fetch promise using then and catch
3238      */ -->
3239
3240     function getData() {
3241         fetch('https://api.coindesk.com/v1/bpi/currentprice.json')
3242             .then((res) => {
3243                 return res.json();
3244             }).then((res) => {
3245                 console.log(res);
3246             }).catch((err) => {
3247                 console.log('error :', err)
3248             })
3249     }
3250
3251     // getData();
3252
3253
3254     <!-- /**
3255      * 2. Consumption of fetch promise using async and await
3256      */ -->
3257
3258     async function getBitcoinData() {
3259         try {
3260             let resp = await fetch('https://api.coindesk.com/v1/bpi/currentprice.json');
3261             resp = await resp.json();

```

```

3262         console.log(resp);
3263     } catch (err) {
3264         console.log('error :', err)
3265     }
3266 }
3267
3268 getBitcoinData();
3269
3270
3271 <!-- /**
3272  * How does fetch works internally.
3273  *
3274  * So lets understand with an image regarding asynchronous
3275  * process
3276  * Consider diagram
3277  * (image.png) in 07_AsyncJs
3278
3279  Asynchronous operations like setTimeout, setInterval etc
3280  are send from js
3281  engine to web api where a register
3282  callback which pushes this call to a taskQueue.
3283
3284  However in case of fetch once it gets send from js
3285  engine this register
3286  callback does not send it to taskQueue, but
3287  sends it to a different priority queue.
3288
3289  This queue is in priority as compared to normal
3290  taskQueue and event loop
3291  is responsible for managing and pushing
3292  these task back to call stack inside js engine
3293  for execution in
3294  functional execution context.
3295
3296  Once a fetch is fired two things happens,
3297  a) Creation of memory space for data
3298  b) Sending fetch call to web api/Node
3299
3300  Inside memory space a data is created which interacts
3301  with two different arrays based on
3302  situation of resolve or reject.
3303
3304  These two arrays are onFulfilled and onRejection
3305
3306  In case of network able to hit the request irrespective
3307  its a 404 or not found, but if this
3308  call happens then its a state of
3309  resolve and whatever response is coming either 404 even
3310  will be pushed to onFulfilled array,
3311  in case of network not able to hit request results in
3312  pushing of error in onRejection.
3313
3314  This data is formulated using these two array and
3315  then is available in global
3316  execution context to be used by us.
3317
3318  */ -->
3319
3320
3321 <
-----
----->
3322
3323 **Prototype**
3324
3325 <!--
3326  * Lets understand what we mean by js is a

```

```

3327 * prototype based language,
3328 * in js everything is a object.
3329 * By everything we mean everything i,e arrays, string etc.
3330 * Why?
3331 *
3332 * Js does prototype inheritance.
3333 *
3334 * let arr = [1,2,3]
3335 * If you do a console.log(arr);
3336 *
3337 * we will see array in browser, if we expand
3338 * it you will see 1 placed
3339 * at 0th index and so on..
3340 * but in last you will see a prototype,
3341 * if you expand it you can see multiple
3342 * keys whose values are some functions.
3343 *
3344 * like
3345 *
3346 * map: function(){.....}.
3347 *
3348 * Now it means arrays are treated as objects
3349 * and has keys which has
3350 * values like functions.
3351 *
3352 * Now the parent of array is Object as in
3353 * last of that prototype we will
3354 * see another prototype which is object
3355 * and then it shows all properties of an object.
3356 *
3357 *      So      array -> object
3358 *
3359 * Similarly string -> object
3360 *
3361 * So everything in js is having a prototype
3362 * inheritance from object.
3363 * In the similar way any function which we create is
3364 * also having prototype
3365 * inheritance from object and is basically
3366 * an object only.
3367 *
3368 * Thus to this multiply function we can add keys like
3369 * name and even methods
3370 * like value which is a function.
3371 * This multiply function is a constructor function.
3372 *
3373 * This new keyword helps us to create an instance of
3374 * an object or constructor
3375 * function and bind all the prototypes
3376 * of that object or constructor function to that instance.
3377 * -->
3378
3379 function multiply(num1, num2) {
3380     this.num1 = num1;
3381     this.num2 = num2;
3382 }
3383
3384 multiply.prototype.name = 'Multiply method';
3385 multiply.prototype.value = function () {
3386     return this.num1 * this.num2;
3387 }
3388
3389 const mul = new multiply(1, 2);
3390 console.log(mul.value());
3391 console.log(mul.name);
3392
3393

```

```

3394
3395 <!-- /**
3396  * Just like say we have length property in string
3397  * let str = "abc";
3398  * str.length //3
3399  *
3400  * In same manner To any instance of string ,
3401  * define a true length property which helps to get the
3402  * trueLength of that string.
3403  * For ex: str = "      Abc      ", trueLength = 3, so we
3404  * should be able to get true length by
3405  * getting rid of all spaces at front or at back.
3406  *
3407  * The property should be named as trueLength only.
3408  *
3409  * Ex:
3410  * let myName = "      Gaurav"
3411  * console.log(myName.trueLength) // 6
3412 */ -->

```

```

3413
3414 Object.defineProperty(String.prototype, 'trueLength', {
3415   get: function () {
3416     return this.trim().length
3417   }
3418 })
3419
3420 let str = "      Gaurav";
3421 console.log(str.length);
3422 console.log(str.trueLength);
3423
3424

```

3425 <----->

```

3426
3427 **Oops**
3428
3429 <!-- /**
3430  * Javascript is object oriented from ES6,
3431  * however javascript is always
3432  * a prototype based language and
3433  * thus under the hood it will always remain a
3434  * prototype based language ,
3435  * although some syntactical sugars
3436  * may give us a feel of js being object
3437  * oriented.
3438  *
3439  * Lets understand what we mean by js is
3440  * a prototype based language,
3441  * in js everything is a object.
3442  * By everything we mean everything
3443  * i,e arrays, string etc.
3444  * Why?
3445  *
3446  * Js does prototype inheritance.
3447  *
3448  * let arr = [1,2,3]
3449  * If you do a console.log(arr);
3450  *
3451  * we will see array in browser,
3452  * if we expand it you will see 1 placed
3453  * at 0th index and so on..
3454  * but in last you will see a prototype,
3455  * if you expand it you can see
3456  * multiple keys whose values are some functions.
3457  *
3458  * like
3459  *
3460  * map: function(){.....}.

```

```

3461  *
3462  * Now it means arrays are treated as objects and
3463  * has keys which has
3464  * values like functions.
3465  *
3466  * Now the parent of array is Object as in last
3467  * of that prototype we
3468  * will see another prototype which is object
3469  * and then it shows all properties of an object.
3470  *
3471  *      So      array ->  object
3472  *
3473  * Similarly string -> object
3474  *
3475  * So everything in js is having a prototype
3476  * inheritance from object.
3477  */ -->
3478
3479
3480 <!-- /**
3481  *
3482  *This is an example of constructor function.
3483  this refers to the current execution context.
3484  So inside the constructor function
3485  if I log this so it will be an
3486  global object in context to the function userData.
3487  Inside that global object I m
3488  setting userName, isLoggedIn and
3489  loginCount property.
3490  And the values are being set to
3491  the arguments which is being passed
3492  to this function.
3493
3494  But there is a main problem here,
3495  suppose i create another user user1
3496  with different values , so it will override the
3497  values which we being passed for user.
3498  Because now the current execution context
3499  is a different instance and
3500  thus we need new keyword, this helps us creation
3501  of different instances.
3502  */ -->
3503
3504 function userData(userName, isLoggedIn, loginCount) {
3505     // console.log(this);
3506     this.userName = userName;
3507     this.isLoggedIn = isLoggedIn;
3508     this.loginCount = loginCount;
3509     return this;
3510 }
3511
3512 const user = userData("Gaurav", true, 5);
3513 console.log(user.userName);
3514
3515 const user2 = userData("Bhatt", false, 4);
3516 console.log(user.userName); // bhatt gets printed so this has
3517                             over rided user values.
3518
3519
3520 <!-- /**
3521  * In order to solve this issue we need a new keyword
3522  * The new keyword in JavaScript is used to create an instance
3523  * of an object from a constructor function or class.
3524  */ -->
3525
3526 const user3 = new userData("Panda", true, 12);
3527 const user4 = new userData("Mango", false, 12);

```

```

3528 console.log(user3.userName);
3529 console.log(user4.userName);
3530
3531
3532
3533 <!-- /**
3534  * Using classes in js
3535  *
3536  * So even classes are internally constructor functions only,
3537  * only syntactical sugar gives us a feel of
3538  * class however this class Person is equivalent to a constructor
3539  * function defined like
3540  *
3541  * function Person(name, age, mobileNumber) {
3542  *     this.name = name;
3543  *     this.age = age;
3544  *     this.mobileNumber = mobileNumber;
3545  * }
3546  *
3547  * Person.prototype.greet = function() {
3548  *     console.log(`Hi ${this.name},
3549  *                 age: ${this.age},
3550  *                 your mobile number is ${this.mobileNumber}`);
3551  * };
3552
3553  So js although syntactically gives us a feel of
3554  classes but internally
3555  under the hood it will always remain a prototype
3556  based language
3557  */ -->
3558
3559 class Person {
3560     constructor(name, age, mobileNumber) {
3561         this.name = name;
3562         this.age = age;
3563         this.mobileNumber = mobileNumber;
3564     }
3565
3566     getDetails() {
3567         return `Hi ${this.name}, age: ${this.age},
3568             your mobile number is
3569             ${this.mobileNumber}`;
3570     }
3571 }
3572
3573 let p1 = new Person("Gaurav", 21, 9354377);
3574 let p2 = new Person("Panda", 34, 3876578976);
3575
3576 console.log(p1.getDetails());
3577 console.log(p2.getDetails());
3578
3579 <!-- /**
3580  * Inheritance
3581  */ -->
3582
3583 class User {
3584     constructor(userName) {
3585         this.userName = userName;
3586     }
3587
3588     greetUser() {
3589         console.log(`Welcome ${this.userName}`);
3590     }
3591 }
3592
3593 <!-- /**Extends keyword is user to have
3594 inheritance

```

```

3595     * and super keyword is used to call
3596     * constructor of parent class
3597     */ -->
3598     class Teacher extends User {
3599         constructor(userName, email, mobileNumber) {
3600             super(userName);
3601             this.email = email;
3602             this.mobileNumber = mobileNumber;
3603         }
3604     }
3605
3606     let teacher = new Teacher("Gaurav", "bh@hjj", 98789);
3607     teacher.greetUser();
3608
3609
3610
3611     <!-- /**Static keyword
3612     *
3613     * Static is used when we need to make a
3614     * property/methods belong only to the class and not
3615     * to the instances of the class or the child
3616     * or classes which inherit it.
3617     */ -->
3618
3619     class Universe {
3620         static god = "God";
3621
3622         constructor(planetName, area) {
3623             this.planetName = planetName;
3624             this.area = area;
3625         }
3626
3627         static getGodName() {
3628             return this.god;
3629         }
3630     }
3631
3632     let u = new Universe("Earth", 1200);
3633     // console.log(u.getGodName()); // error not a function because its inaccessible
3634     // to instances of the class
3635
3636     console.log(Universe.getGodName());
3637     console.log(u.god); // undefined
3638     console.log(Universe.god);
3639
3640
3641     <----->
3642
3643     **Call & Bind**
3644
3645         <!--HTML>
3646
3647         <!DOCTYPE html>
3648         <html lang="en">
3649
3650         <head>
3651             <meta charset="UTF-8">
3652             <meta name="viewport" content="width=device-width, initial-scale=1.0">
3653             <title>Document</title>
3654         </head>
3655
3656         <body>
3657             <button id="btn">Click me</button>
3658         </body>
3659         <script src="01_Call&Bind.js"></script>
3660
3661         </html>

```

```

3662
3663                                     <!--SCRIPT>
3664
3665 <!-- /**
3666  * In js whenever a function is called its taken into a
3667  * call stack
3668  * and then its execution happens,
3669  * once its execution gets completed its taken out of the
3670  * call stack
3671  * and its function execution ends there along with its execution
3672  * scope.
3673  *
3674  * Now lets suppose i have a scenario in which i am setting
3675  * some properties ,
3676  * however on one of the
3677  * property i need to call another function which
3678  * sets this property,
3679  * as soon as the subFunction call is complete,
3680  * its taken out of call stack
3681  * and its execution context ends, thus
3682  * we will not be able to set the that property.
3683  *
3684  * This scenario is handled by call.
3685  */ -->
3686
3687 function setUsername(userName) {
3688     this.userName = userName;
3689 }
3690
3691 function setDetails(userName, email, mobileNumber) {
3692     setUsername(userName);
3693     this.email = email;
3694     this.mobileNumber = mobileNumber
3695 }
3696
3697 const res = new setDetails("Gaurav", "bhatt@we.com", 98656789876);
3698 console.log(res); // userName will not be there
3699
3700 <!-- /**
3701  *
3702  *The problem can be solved via call
3703  */ -->
3704 function setUsername1(userName) {
3705     this.userName = userName;
3706 }
3707
3708 <!-- /**
3709  *
3710  * here we have passed our execution context of setDetails1
3711  * function to setUsername1 function
3712  * as soon as setUsername1 gets executed and its
3713  * function execution ends ,
3714  * still since it was working
3715  * with execution context of setDetails, so setDetails
3716  * will now have its userName
3717  *
3718  * So we use a call when we need to invoke the function
3719  * immediately with a specific context.
3720  */ -->
3721 function setDetails1(userName, email, mobileNumber) {
3722     setUsername1.call(this, userName);
3723     this.email = email;
3724     this.mobileNumber = mobileNumber
3725 }
3726
3727 const res1 = new setDetails1("Gaurav", "bhatt@we.com", 98656789876);
3728 console.log(res1);

```



```

3729
3730
3731
3732 <!-- /**
3733  * Call and bind both are used to pass a specific execution
3734  * context
3735  * to some other function.
3736  * However we use call when we need to immediately invoke
3737  * the other function with a
3738  * specific execution context.
3739  *
3740  * We use bind when we need to invoke a function with a
3741  * specific execution context
3742  * later say during some event.
3743  */ -->
3744
3745
3746 <!-- /**
3747  * Here constructor is having an current execution
3748  * context in which userName is there,
3749  * but handleClick is not having the current execution
3750  * context where userName is
3751  * present
3752  * since we don't need to invoke the function handleClick
3753  * immediately and only once
3754  * the button is clicked
3755  * i,e on click event thus its an classic case of bind,
3756  * if we would have need to
3757  * immediately invoke the handleClick
3758  * function we would have used call, however in this
3759  * case we need bind.
3760  */ -->
3761 class Bind {
3762     constructor() {
3763         this.userName = "Gaurav";
3764         // document.querySelector('#btn').
3765         addEventListener('click', this.handleClick);
3766         // this will result in undefined userName
3767
3768         document.querySelector('#btn').
3769         addEventListener('click', this.handleClick.bind(this));
3770     }
3771
3772     handleClick() {
3773         console.log(`Button clicked by ${this.userName}`);
3774     }
3775 }
3776
3777 let obj = new Bind();
3778
3779
3780 <
-----
-->
3781
3782 **Getter & Setter**
3783
3784 class Person {
3785     constructor(name, password) {
3786         this.name = name;
3787         this._password = password; // _ variable intended to be private
3788     }
3789
3790     set password(val) {
3791         this._password = val;
3792     }
3793

```

```

3794     get password() {
3795         return this._password;
3796     }
3797 }
3798
3799 let p1 = new Person("Gaurav", "abc");
3800 console.log(p1.password);
3801
3802 <!-- /**
3803  * when we invoke the property password since it was a getter
3804  * so get
3805  * function gets invoked
3806  * automatically and thus we don't invoke get
3807  * function using functionCall().
3808 */ -->
3809
3810
3811 <----->
3812
3813 **Lexical scoping**
3814
3815 <!-- /**
3816  * Lexical scoping in js is defined as the concept in which
3817  * the scope members of the outer function is available to
3818  * the inner function as well.
3819  *
3820  * In js in global execution context once a function is
3821  * called it gets loaded in
3822  * the execution context
3823  * and since in this case innerFunction is getting called
3824  * from outerFunction thus,
3825  * innerFunctions execution
3826  * context will also get loaded on top of outerFunction
3827  * executional context.
3828  *
3829  * This inner function's executional context also shares
3830  * the memory of its outer
3831  * parent function and thus
3832  * have access to its scope members as well.
3833  *
3834  * This concept is lexical scoping.
3835 */ -->
3836
3837 function outerFunction() {
3838     let name = "Gaurav";
3839     function innerFunction() {
3840         console.log(name);
3841     }
3842     innerFunction();
3843 }
3844
3845 outerFunction();
3846
3847
3848 <----->
3849
3850 **Clousers**
3851
3852
3853 <!-- /**
3854  * In this use case, inner function is inside the outer function,
3855  * and due to lexical scoping it has access to
3856  * the scope members of the outer function.
3857  *
3858  * Now outer function actually returns the inner function reference.
3859  *
3860  * So when we say

```

```
3861  *
3862  * myFunc = outerFunction();
3863  *
3864  * so myFunc goes reference of innerFunction and after this line,
3865  * the functional execution context
3866  * of outer function gets destroyed from the stack.
3867  *
3868  * Now when we called myFunc how it gave me the name variable as
3869  * name variable was in scope of
3870  * outer function whose execution context has ended.
3871  *
3872  * the reason behind this is when inner function is returned,
3873  * the entire lexical scope of the inner function as
3874  * well as entire lexical scope of
3875  * the outer function is also
3876  * present thus inner function still has access to
3877  * scope member of outer function
3878  * and this phenomena is called
3879  * as closer.
3880  *
3881  * A closure in JavaScript is a feature that
3882  * allows a function to "remember" and
3883  * access its lexical scope (the scope in which it was declared) even when
3884  * the function is executed outside that scope.
3885  */ -->
3886
3887  function outerFunction() {
3888      let name = "Gaurav";
3889      function innerFunction() {
3890          return name;
3891      }
3892      return innerFunction;
3893  }
3894
3895  let myFunc = outerFunction();
3896  console.log(myFunc());
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
```