10. Function

→ Syntax : Function Definition ⇒

```
function funName() {
    // do something
}
```

Function Calling ⇒ `funName();`

e.g.

```
function hello() {
    console.log("hello World");
}
hello();
hello();
hello();
```

Output :

```
hello world
hello world
hello world
```

e.g.

```
function print1to5() {
    for(let i=1; i<=5; i++) {
        console.log(i);
    }
}
print1to5();
```

Output :

```
1
2
3
4
5
```

* Function with Arguments

→ Values we pass to the function

```
function funName( arg1, arg2, arg3 ...) {
        // do something
}
```

eg

```
function printName (name) {
        console.log (name);
}
printName ("parth");
```

output: parth

eg.

```
function printInfo (name, age) {
        console.log(`${name}'s age is ${age}.`);
}
printInfo("parth", 27);
printInfo ("parth");
```

Output:

```
parth's age is 27.
parth's age is undefined.
```

* Return

→ return keyword is used to return some value from the function.

```
function funName ( arg1, arg2, arg3 ... ) {
        // do something
      return  value ;
}
```

eg
```
function  sum (a, b) {
     return  a + b ;
}
let  s  =  sum (3, 4);
console.log (s);
console.log ( sum (3,4) );
console.log ( sum ( sum (2, 3), 3 ) );
```

Output :
```
7
7
8
```

eg Sum of n natural numbers
```
function getSum (n) {
    let sum = 0 ;
    for (let i = 1;  i <= n; i++) {
            sum += i ;
    }
    return sum ;
}
console.log (getSum (10));        ⇒ 55
```

* Scope

→ Scope determines the accessibility of variables, objects, and functions from different parts of code.

- Function Scope
- Block Scope
- Lexical scope

⇒ Function Scope

→ Variables defined inside a function are not accessible (visible) from outside the function

eg
```
function getSum(a,b) {
      let sum = a + b;
}
getSum(1,2);
console.log(sum);
```

Output : error = 'sum' not defined

eg
```
let sum = 54;   ← Global Scope
function getSum(a,b) {                Output : 3
  function scope → let sum = a + b ;           54
      console.log( sum );
}
getSum(1,2);
console.log(sum);
```

⇒ Block Scope

→ Variable declared inside a { } block cannot be accessed from outside the block

→ Block scope applies only for let & const.

eg
```
{ let a = 25 ; }
    console.log(a);
```
Output: error

eg
```
for (let i=1;  i<=5;  i++) {
        //console.log();
    }
    console.log(i);
```

output: error

⇒ Lexical Scope

→ Variable defined outside a function can be accessible inside another function defined after the variable declaration. The opposite is NOT true.

e.g
```
function outerFun() {
        let x =5;
        function innerFun() {
            console.log(x);
        }
        innerFun();
    }
    outerFun();
```
Output : 5

```
e.g. function  outerFun () {
         let x = 5 ;
         function  innerFun () {
              let a = 10;
              console.log(x);
         }
         console.log (a);   ←—— not accessible
         innerFun ();
     }
     outerfun ();
```

output : error

* Function Expressions

→ A different style to write a function (nameless function)

```
const variable = function (arg1, arg2, arg3,...){
        // do or return something
}
```

```
e.g   const sum = function ( a, b ) {
           return a+b;
      }
      console.log (sum (2, 3) ) ;
```

output : 5

* Higher Order Function

→ A function that does one or both of the following:

- take one or multiple function as arguments
- returns a function

⇒ Take one or multiple function as arguments

eg:
```
function multipleGreet (fun, n) {
          for (let i=1; i<=n; i++) {
                fun ();
          }
}
let greet = function () {
      console.log("hello");
}
multipleGreet( greet, 2 );
multipleGreet( function () { console.log("hello");}, 3);
```

Output:
```
hello
hello
hello
hello
hello
```

⇒ Returns a Function

eg
```
function oddEvenTest (request) {
    if( request == "odd" ) {
        return function (n) {
            console.log(! (n%2 == 0));
        }
    } else if ( request == "even") {
        return function (n) {
            console.log (n%2==0);
        }
    } else {
        console.log ("wrong request");
    }
}
let request = "even";
let fun = oddEvenTest (request);
console.log (fun(10) );
console.log (fun (9) );
```

Output:
```
true
false
```

* Methods

→ Actions that can be performed on an object

eg
```
const calc = { add: function(a,b){ return a+b; },
               sub : function(a,b) { return a-b; },
               mul: function(a, b){return a*b;}
};
console.log(calc. add (1,2) );
```

Output : 3

→ Shorthand : No need no use function keyword in object to define function

eg
```
const calc = { add (a, b) { return a+b; },
               sub (a , b) { return a-b; },
               mul(a , b) { return a*b; }
};
```

11. Try & Catch

→ The try statement allows you to define a block of code to be tested for errors while it is being executed.

→ The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

eg
```
try {
    console.log(a);
} catch {
    console.log("variable a is not defined");
}
```

output: variable a is not defined

eg
```
try {
    console.log(a);
} catch(err) {
    console.log(err);
}
```

output: ReferenceError: a is not defined.

12. this Keyword

→ this keyword refers to an object that is executing the current piece of code

eg
```
const obj = {
    name: "parth",
    math: 88,
    phy : 90,
    chem: 89,
    getAvg() { console.log(this);
               let avg = (this.math+ this.phy+ this.chem)/3;
               console.log(avg);
    }
};
function getArg() {
 console.log(this);
}
console.log(obj.getArg());
getArg();
```

Output : { name:'parth', getAvg : f getAvg(), phy:90, math:88, chem:89}
89
→ Window{ window: Window, self: Window, ... }

window object of browser (in-built)

13. Arrow Function

→ not a function but works the same

```
const fun = ( arg1, arg2, ... ) => { // do something };
```

eg
```
const sum = (a , b ) => {
      console.log (a+b);
};
sum(10, 20);
```
Output: 30

For single value no need of parenthesis

eg
```
const sqr = (n ) => {
      console.log (n*n);
};
sqr(10);
```
Output: 100

eg
```
const hello = () => {
      console.log("Hello World");
};
hello();
```
Output: Hello World

→ Implicit (automatic) Return

→ Arrow Function only returns a value then no need to write return keyword.

```
const fun = ( arg1, arg2, ...) => ( value );
```

eg
```
const mul = ( a , b ) => ( a * b );
```

14. Set Timeout & Interval

→ Set Timeout (inbuilt function of Window Object)

```
setTimeout (function , timeout );
               ↑          ↑
           callback    time in
                       milliseconds
```

eg

```
console.log("Hi there");

setTimeout( () => {
                console.log("apna college");
}, 4000);

console.log("Welcome to");
```

Output:

```
Hi there
Welcome to
apna college   ← this will print after 4 second
```

→ Set Interval (inbuilt function of window object)
execution after set interval infinitely

```
setInterval ( function, timeout);
```

eg

```
setInterval( () => { console.log("apna college");
}, 2000 );
```

output:

```
apna college    ← printed after 2 second
apna college    ← printed after 2 second
     ⋮                ⋮
```

→ To stop infinite use clearInterval (id);

eg.
```
let id = setInterval(  () => {
      console. log( "apna  college");
},  2000 );
console.log(id);
setTimeout(  () => {
      ~~console~~ clearInterval(id);
}, 4000);
```

output : 1
```
apna college  ← printed after 2 seconds
apna college  ← printed after 2 seconds
```

15. this with Arrow Function

→ Arrow Function scope (is lexical scope) = parent's scope
→ Function scope = calling object scope

eg
```
const student = {
    name: "parth",
    marks: 90,
    prop: this,     // global scope
    getName: function () {
          console.log(this);
          return this.name;
    },
    getMarks: () => {          parent's scope
          console.log(this);
          return this.marks;
    },
    getInfo1: function () {
          setTimeout( () => {          parent's scope = student
                console.log(this);
          }, 200);
    },
    getInfo2: function() {
          setTimeout( function() {
                console.log(this);     object scope = window
          }, 200);                     calling object of setTimeout
    }
};
student.getName();
student.getMarks();
student.getInfo1();
student.getInfo2();
```

P.T.O. →

Output:
```
{ name: 'parth', marks: 90, ... }
'parth'
Window { window: Window, ... }
undefined
{ name: 'parth', marks: 90, ... }
Window { window: Window, ... }
```

16. Array Methods

* forEach

```
arr.forEach( some function definition or name );
```

eg
```
let arr = [ 1 , 2 , 3 , 4 , 5 ];
let print = function (element) {
        console.log(element);
}
arr.forEach(print);
```

OR

```
arr.forEach( function (element) {
        console.log(element);
});
```

OR

```
arr.forEach( (element) => {
        console.log(element);
});
```

Output :
```
1
2
3
4
5
```

P.T.O. →

~~eg const obj = {~~

* map

```
let newArr = arr.map( some function definition or name)
```

eg
```
let num = [ 1 , 2 , 3 , 4 ];
let double = num.map( (element) => {
        return element * 2;
});
console.log (double);
```

output : (4) [2 , 4 , 6 , 8]

* filter

```
let newArr = arr.filter( some function definition or name);
```

eg
```
let nums = [ 2 , 4 , 1 , 5 , 6 , 2 , 7 , 8 , 9 ];
let even = nums.filter( (num) => (num % 2 == 0) );
console.log(even);
```

output : (4) [2 , 4 , 6 , 8]

* every

→ Returns true if every element of array gives true for some function, else returns false.

```
arr.every( some function definition or name );

eg [1, 2, 3, 4].every( (el) => (el % 2 == 0) );

output : false

eg [2, 4].every( (el) => (el % 2 == 0) );
Output : true
```

* some

→ Returns true if some elements of array gives true for some function, else returns false.

```
arr.some( some function definition or name );

eg [1,2,3,4].some( (el) => (el % 2 == 0) );

output : true

eg [1,3].some( (el) => (el % 2 == 0) );

output : false
```

* reduce

→ Reduces the array to a single value

arr.reduce(reducer function with 2 variables (accumulator, element));

eg
```
[1,2,3,4].reduce( (result,element) => (result+element) );
```

output : 10

```
                         [1, 2, 3, 4]
execution :  (0, 1) => 1
             (1, 2) => 3
             (3, 3) => 6
             (6, 4) => 10
         result   element   result
```

eg
```
[1,10,5,11,3].reduce( (max, el) => {
        if (el > max) {
            return el;
        } ~~eles~~ else {
            return max;
        }
});
```

output : 11

```
                 max  el    max
                  ↓    ↓     ↓
execution :      (0,1)   => 1
                 (1,10)  => 10
                 (10,5)  => 10
                 (10,11) => 11
                 (11,3)  => 11
```

17. Default Parameter, Spread

* Default Parameter

→ Giving a default value to the arguments, which will be value of the variable if there are no value passed when calling the function

```
function fun (a, b=2) { // do something }
```

eg
```
function sum (a, b=3) { return a+b; }
console.log( sum(2) );
console.log( sum(1,5) );
```

output :
```
5
6
```

* Spread

→ Expands an iterable into multiple values

```
function fun (...arr) { // do something }
```

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

output :
```
a b c
```

eg
```
let arr = [1, 2, 3, 5];
console.log( Math.min (... arr) );
console.log( Math.max (... arr) );
console.log( ... arr);
```

output :
```
1
5
1 2 3 5
```

* Spread with Array Literals

eg
```
let arr = [1, 2; 3, 4, 5];
let newArr = [... arr];
console.log(newArr);
```

Output: (5) [1, 2, 3, 4, 5]

eg
```
let chars = [... "hello"];
console.log(chars);
```

output: (5) ['h', 'e', 'l', 'l', 'o']

eg.
```
let odd = [1, 3, 5, 7, 9];
let even = [2, 4, 6, 8, 10];
let nums = [...odd, ...even];
console.log( nums );
```

output: (10] [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]

* Spread with Object Literals

eg
```
let data = { email : "abc@gmail.com",
             password : "abcd"
};
let dataCopy = { ... data, id: 123 };
console.log (dataCopy);

output : { email : 'abc@gmail.com',
           password : 'abcd',
           id: 123 }
```

eg
```
let arr = [ 1 , 2 , 3 , 4 ];
let obj = { ... arr };
console.log ( obj );
                                    key will be automated
                                    begin with 0
output : { 0:1 , 1:2 , 2:3 , 3:4 }
```

eg
```
let obj = { ... "hello" }
console.log ( obj );

output : { 0:'h' , 1:'e' , 2:'l' , 3:'l' , 4:'o' }
```

* Rest

→ Allows a function to take an indefinite number of arguments and bundle them in an array

e.g
```
function sum (...args) {
    return args.reduce( (add, el) => add + el );
}
console.log( sum (1,2,3,4,5));
```

output : 15

inbuilt collection not an array

```
→ function fun () {
      console.log(arguments);
      console.log(arguments.length);
  }
  console.log( fun (2,4,6) );
```

```
output : Arguments (3) [ 2, 4, 6, ... ]
         3
```

* Destructuring Array

→ storing values of array into multiple variables

eg.
```
let names = [ "abc", "cde", "efg", "ghi" ];
let [winner, runnerup, ...others] = names;
console.log (winner, runnerup);
console.log (others);
```

output :
```
abc  cde
(2) [ 'efg' , 'ghi']
```

* Destructuring Object

→ eg.
```
const student = { name: "parth",
                  age: 25 ,
                  class: 10,
                  sub: ["phy", "che"],
                  user: "abc@123",
                  pass: "abcd"
};
                ← name has to be same as key
let { user, pass } = student;
console.log (user);                    // 'abc@123'
                      ← new variable
let { user: username, pass } = student;
console.log (username);                // 'abc@123'

let{ city = "surat" } = student;
console.log (city);                    // surat
```