* Higher Order Functions :-

→ These functions depends & operates on other function.

→ They take another function as an argument & then execute the logic.

```
function foo() {
    ≡
}

function bar(foo) {
    foo();
}
```

higher order function

Function passed as parameter.

→ So, a function that returns a function or takes other function as argument is called a higher-order function.

* Built-in higher order functions :-

i) map() :-

→ It is a higher-order function for array.

→ It takes another function as an argument & returns an array in which every value is actually populated by calling a function foo with original array elements as argument.

```
function square (element) {
    return element ** 2;
}

const arr = [1, 2, 3];
const resultArray = arr.map(square);
console.log(resultArray); → [1, 4, 9]
```

→ here, map() is a higher order function which takes another function square() as an argument & returns a new array which is stored in resultArray.

→ map() internally iterates over the array, passes every element to the argument function foo() & then stores the returned value from foo() to result array.

→ In the above code, map() iterates over arr & for every element of arr, it calls square().

→ The value returned from square() is stored in resultArray.

→ We can use map() in situations when we have to do an operation on every

element of the array & store the result of each operation.

- Working with index & element using map():-

→ If the argument function of map() takes two params, then the first param is the element & the 2nd param is the index of the original array.

index.js > ...

```
function print(element, index) {
  return `Index: ${index}, Element: ${element}`;
}

let arr = [1, 2, 3];
let resultArr = arr.map(print);

for (let element of resultArr) {
  console.log(element);
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] node "d:\Programming\web-dev\js-workspace\index.js"
Index: 0, Element: 1
Index: 1, Element: 2
Index: 2, Element: 3
```

2) filter():-

→ It is a higher order function that iterates over an array.

→ The argument function that we pass inside the filter(), should always return a boolean. If not, then the result is converted to boolean.

→ If the returned value is true, only then it stores the output to the result array.

index.js > ...

```
function isEven(element) {
  if (element % 2 === 0) return true;
  return false;
}

let arr = [1, 2, 3, 4, 5, 6, 7];
let resultArray = arr.filter(isEven);
console.log(resultArray);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] node "d:\Programming\web-dev\js-workspace\index.js"
[ 2, 4, 6 ]
```

3) reduce() :-

→ For every element of the array it calls the argument function foo() that accumulates the result of further function calls.

index.js > ...

```
1  function sum(previousResult, currentElement) {
2    console.log(`Current element: ${currentElement}`);
3    console.log(`Previous result: ${previousResult}`);
4    return previousResult + currentElement;
5  }
6
7  let arr = [1, 2, 3, 4];
8  let sumOfElements = arr.reduce(sum);
9  console.log("Sum of elements: ", sumOfElements);
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] node "d:\Programming\web-dev\js-workspace\index.js"
Current element: 2
Previous result: 1
Current element: 3
Previous result: 3
Current element: 4
Previous result: 6
Sum of elements:  10
```

→ Starts from index 1, because element in index 0 is stored in previousResult.

- Real world example of reduce() :-

```
function addPrices (prevResult, currValue) {
  let totalPrice = prevResult.price + currValue.price;
  return totalPrice;
}

let cart = [ {price: 50000, product: "Samsung Tab S8"},
             {price: 100000, product: "Macbook Air"}, ];
let totalPrice = cart.reduce(addPrices);
console.log(totalPrice);
```

→ 150000

→ This is how an online shopping cart works.