Function expression:

Regular function syntax:

```
//defining a function
function greet(){
    console.log("Welcome to masai");
}


//calling the above function
greet();
```

With the function expression, we can hold this function inside a variable, and we can use that variable as a function name.

Example:

```
//defining a function
let greet = function(){
    console.log("welcome to masai");
}

//calling the above function
greet();
```

a. **Arrow function:**

- It is the shorter way to write a function expression.
- It is similar to the Lambda function in Java

Example:

```
// let greet = () => {
//     console.log("Welcome to Masai");
// };


// Inside the function body if only one statement is there then the
curly //bracket is optional but for more than one line it is
mandatory.

let greet = () => console.log("Welcome to masai");

greet();
```

Arrow function with parameter:

Example: write a arrow function to accept a number and print the square of that number.

```
    // let printSquare = (num) =>{

    // // let square = num * num;
    // // console.log("The square of the number",num," is
    ",square);

    // console.log("The square of the number",num," is ",num*num);

    // }


    // let printSquare = (num) => console.log("The square of the
    number",num," is ",num*num);


    //If only one paremeter is there the the small bracket is
    optional, but for more parameters it is mandatory
```

```
let printSquare = num => console.log("The square of the
number",num," is ",num*num);

printSquare(5);
```

Student Activity: Create an arrow function that accepts 2 numbers as a parameter and print the addition of both numbers.

Solution:

```
// let doAddtion = (num1, num2) => {

//      // let result = num1+num2;
//      // console.log("The result is ",result);

//      console.log("The result is ",num1+num2);
// }


let doAddtion = (num1, num2) =>  console.log("The result is
",num1+num2);


//calling the above function
doAddtion(10,50);
```

Arrow function with return type:

Example: write a function which accepts a number as a parameter and returns the square of that number.

Option 1: Normal way

```
let getSquare = (num) =>{

    // let result = num*num;
    // return result;

    return num * num;
}

//call the function and catch the retured value
let x= getSquare(5);
console.log("The result is:",x);
```

Here only one parameter is there then we can remove the small bracket.

Option 2: by removing the small bracket

```
let getSquare = num =>{

    // let result = num*num;
    // return result;

    return num * num;

}

//call the function and catch the returned value
let x= getSquare(5);
console.log("The result is:",x);
```

Here only one statement is there so we can remove the curly braces also
But remember without curly braces, the **return** keyword is not allowed. We can remove the **return** keyword also.

Option 3:

```
let getSquare = num =>  num * num;

//call the function and catch the returned value
let x= getSquare(5);
console.log("The result is:",x);
```

Student Activity: Write an arrow function that accepts a number if the number is positive then return true and if the number is negative then it returns false.

Option1:

```
let checkPositiveOrNegative = (number) => {

    if(number > 0){
        return true;
    }
    else{
        return false;
    }
}


//calling the above function
console.log(checkPositiveOrNegative(-10));
```

Option2:

```
let checkPositiveOrNegative = number => {
    return number > 0;
}


//calling the above function
console.log(checkPositiveOrNegative(-10));
```

Option3:

```
let checkPositiveOrNegative = number => number > 0;


//calling the above function
console.log(checkPositiveOrNegative(-10));
```

Activity: Write an arrow function that accepts an object and prints the details of that object.

```
let printDetailsOfObject = (obj) => {

for(let k in obj){

    console.log(k,"==========",obj[k]);

}

};


let student= {roll: 100, name: "Ram", marks: 850};
printDetailsOfObject(student);

printDetailsOfObject({roll: 102, name: "Ramesh", marks: 840});
```

HOF (High Order functions and callbacks):

- In terms of function parameters, a function in many programming languages can indeed accept various types of values as arguments.
- This includes primitive types like **numbers, strings,** and **booleans**, as well as more complex types like **arrays** and **objects**.
- Moreover, a function can indeed accept another function as an argument. When a function does this, it's referred to as a **higher-order function (HOF)**, and the function being passed as an argument is called a **callback function**.

Example:

```
// Higher-order function that accepts a callback function
function higherOrderFunction(callback) {
   console.log("Inside higher-order function");
   // Execute the callback function
```

```
          callback();
        }

        // Callback function
        let callbackFunction = function() {
          console.log("Inside callback function");
        }


        // Passing the callback function to the higher-order function
        higherOrderFunction(callbackFunction);


  // OR calling the highOrderFunction by supplying the callbackFunction
directly

  higherOrderFunction(function(){
    console.log("inside the callback function");
  });

  // OR calling the highOrderFunction by supplying the callbackFunction as
arrow function

  higherOrderFunction(() => {
    console.log("inside the callback function");
  });
```

Another Example:

```
// Higher-order function (Dadi)
function autnyKiRasoi(callbackFunction) {
    console.log("Namaste! Aunty ki rasoi is open for business!");
    console.log("Time to cook up a storm...");


    // Execute the callback function (family member's task)
    callbackFunction();
    console.log("Aaj ka khaana (Today's meal) is ready! 🍛");

}


// Callback function for another family member's task
let prepareBreakFast = function () {
    console.log("Beta (Son), Preparing the BreakFast !");
}

// Callback function for a family member's task
let prepareLunch = function () {
    console.log("Beti(Daughter) , Preparing the lunch");
}

// Callback function for yet another family member's task
let prepareDinner = function () {
    console.log("Bahu (Daughter-in-law), Preparing the dinner");
}


autnyKiRasoi(prepareBreakFast);
autnyKiRasoi(prepareLunch);
autnyKiRasoi(prepareDinner);
```

Note: Here we need not take 3 different variables to hold all 3 callback function, we can directly pass the function as a callback function to the HOF.

Example:

```
autnyKiRasoi(function(){
    console.log("Beta (Son), Preparing the BreakFast !");
});

autnyKiRasoi(function(){
    console.log("Beti(Daughter) , Preparing the lunch");
});
```

```
autnyKiRasoi(function(){
    console.log("Bahu (Daughter-in-law), Preparing the dinner");
});
```

Student Task: Call the auntyKiRasoi() function 3 times(for breakfast, lunch, dinner) by passing callback function as arrow function to the HOF.

Solution:

```
// Higher-order function (Dadi)
function autnyKiRasoi(callbackFunction) {
    console.log("Namaste! Aunty ki rasoi is open for business!");
    console.log("Time to cook up a storm...");


    // Execute the callback function (family member's task)
    callbackFunction();


    console.log("Aaj ka khaana (Today's meal) is ready! 🍛");

}



// calling the HOF by passing the callback function using arrow function
autnyKiRasoi( () => console.log("Beta (Son), Preparing the BreakFast !"));
autnyKiRasoi( () => console.log("Beti(Daughter) , Preparing the lunch"));
autnyKiRasoi( () => console.log("Bahu (Daughter-in-law), Preparing the
dinner"));
```

Example: HOF passing the value to the callback function

Example1:

```
function highOrderFunction(name, callback) {
    console.log("Inside the HOF");

    //HOF calling the callback function by passing the value
    callback(name.toUpperCase());
}


highOrderFunction("Mahesh", function(n){

//callback function which is accepting the value
 console.log("inside the callback function printing the name",n);
});
```

Example2:

```
      function highOrderFunction(num1,num2, callback) {
    console.log("Inside the HOF");
    console.log("Performing mathematical calculation");


        //HOF is calling the callback function by passing the arguments.
        callback(num1,num2);

        console.log("Mathematical calculation done");
}

// highOrderFunction(10,20, function(n1,n2){
//     console.log("Addition Result", n1+n2);
// })


//Using arrow function
```

```
highOrderFunction(10,20,(n1,n2) =>{
    console.log("Addition Result", n1+n2);
})


highOrderFunction(10,20,(n1,n2) =>{
    console.log("Substraction Result", n1-n2);
})


highOrderFunction(10,20,(n1,n2) =>{
    console.log("Multiplication Result", n1*n2);
})


highOrderFunction(20,5,(n1,n2) =>{
    console.log("Divistion Result", n1/n2);
})
```

Regarding the return type of a higher-order function, yes, it can indeed return a callback function as well. This allows for powerful and flexible programming patterns, where functions can be dynamically created and returned based on certain conditions or calculations.

Example: HOF returning any other function:

```
// Higher-order function that returns a callback function
function createCallbackHOF() {
    // Returning a callback function
    return function() {
      console.log("This is a callback function");
    };
  }

  // Getting the callback function from the higher-order function
  const callback = createCallbackHOF();

  // Executing the returned callback function
  callback();
```

// OR returning the function expression

```
//defining the function
function fun1() {
  console.log("Inside the HOF");

  let f1 = function(){
    console.log("inside the callback function");
  }

  return f1;
}


//calling the HOF function
let cb= fun1();

//calling the cb
cb();
```

// OR return the arrow function.

```
//defining the function
function fun1() {
  console.log("Inside the HOF");

  return () =>{
    console.log("inside the callback function");
  }
}
```

```
//calling the HOF function
let cb= fun1();

//calling the cb
cb();
```

Some of the Important inbuilt HOF in Java-script which takes a callback function as an argument.

Following are some of inbuilt HOF which will work with the list of items(arrays)

```
    Array.forEach();
    Array.sort();
    Array.filter();
    Array.map()
    Array.reduce();
```

Array.forEach:

The **forEach()** method of Array instances executes a provided function/callback once for each array element.

Example:

```
    let arr = ["delhi","mumbai","chennai","kolkata","pune","bangaluru"];

    // arr.forEach(function(item,index,array){


    //     //item: the current item being processed in the array
    //     //index: optional: index of the item inside that array
    //     //array: optional: that array on which this forEach()
    function is called.

    //     console.log(item,"At Index number: ",index, "Of the Array
    ",array);
```

```
      // } );


      //passing callback using arrow function
      arr.forEach((e,i,arr) => {
          console.log(e,"At Index number: ",i, "Of the Array ",arr);
      } );
```

Note: the return type of the forEach() function is undefined.

Example:

```
      let arr = ["delhi", "mumbai", "chennai", "kolkata", "pune",
"bangaluru"];

      let returnedValue = arr.forEach(e => console.log(e.toUpperCase()));

      console.log(returnedValue);
```

<u>**Array.sort();**</u>

Imagine you have a group of your friends lined up to take a photo together. You want them to stand in order based on their height, so the shortest person is on the left and the tallest person is on the right. In real life, you'd ask your friends to move around and swap places until they're in the correct order.

Now, think of JavaScript arrays as a line of friends. Each friend represents an element in the array. The **Array.sort()** method helps you organize this line of friends based on certain criteria, just like organizing your friends by height.

Note: this sort() function will sort the elements in the current array itself

Example:

```
let names = ["Varun","Vivek","Chandan","Amit","Suraj"];

names.sort();

console.log(names); // Output: [ 'Amit', 'Chandan', 'Suraj',
'Varun', 'Vivek' ]
```

Note: the return type of the sort() function is the sorted array itself.

Example:

```
let names = ["Varun","Vivek","Chandan","Amit","Suraj"];

let returndValue= names.sort();
console.log(names); // Output: [ 'Amit', 'Chandan', 'Suraj',
'Varun', 'Vivek' ]

console.log(returndValue);// Output: [ 'Amit', 'Chandan', 'Suraj',
'Varun', 'Vivek' ]

names.sort().forEach(e => console.log(e));
```

Note: The sorting will be in ascending order by default and it will sort them by lexcal order.

Example:

```
let numbers = [5,12,17,3,25,15,2];

numbers.sort();

console.log(numbers); //Output: [ 12, 15, 17, 2, 25, 3, 5]
```

To short them in the proper order we need to pass a callback function to this sort() function.

This callback function is used to determine the order of the elements in the array. The callback function should return a negative, zero, or positive value, depending on the arguments, like:

- a negative value if a should be sorted before b (means a is smaller than b).
- a positive value if a should be sorted after b(means a is greater than b) .
- 0 if a and b are equal and their order doesn't matter(means both a and b are equal).

Example:

```
let numbers = [5,12,17,3,25,15,2];

// numbers.sort(function(a,b){

// if(a > b)
//     return +1;
// else if(a < b)
//     return -1;
// else
//     return 0;
// });

// console.log(numbers); //Output: [2, 3, 5, 12, 15, 17, 25]


//Using shortcut:

// numbers.sort(function(a,b){

//         return a-b;
// });

//     console.log(numbers); //Output: [2, 3, 5, 12, 15, 17, 25]

//Using arrow function

numbers.sort((a,b) => a-b);
```

```
    console.log(numbers); //Output: [2, 3, 5, 12, 15, 17, 25]
```

Example Sorting the array of Student object based on their marks:

```
    let students = [

        { roll: 100, name: "Ram", marks: 750 },
        { roll: 110, name: "Ramesh", marks: 850 },
        { roll: 120, name: "Suresh", marks: 450 },
        { roll: 130, name: "Varun", marks: 950 },
        { roll: 140, name: "Dinesh", marks: 550 },

    ];


    students.sort((a, b) => {

        return a.marks - b.marks;
    })

    //students.sort((a, b) => a.marks - b.marks);


    console.log(students);
```

Array.filter()

JavaScript Array's **filter()** Method is used to create a new array from a given array consisting of only those elements from the given array that satisfy a **condition/test** set by the supplied callback function.

Syntax:

```
    array.filter(function(currentValue, index, arr))
```

1. **function()** Required: A function to run for each array element.
2. **currentValue** Required: The value of the current element.
3. **index** Optional: The index of the current element.

4. **arr** Optional: The array of the current element.

- The filter() method creates a new array filled with elements that pass a test provided by a function.
- The filter() method does not execute the function for empty elements.
- The filter() method does not change the original array.

Example:

```
let ages = [32, 33, 16, 40,50,17];

//boolean condition/test function
function checkAdult(age) {

    // if(age >=18)
    //     return true;
    // else
    //     return false;

  return age >= 18;
}

let result = ages.filter(checkAdult);
console.log(result);
```

//Or passing the callback function directly

// using function expression

```
let ages = [32, 33, 16, 40,50,17];


let filteredArray= ages.filter(function(item, index, array){

    return item >=18;
```

```
      });

      console.log(filteredArray);

//Or Using the arrow function:

      let filteredArray= ages.filter(age => {
          return age >=18;
      });

      console.log(filteredArray);

      //ages.filter(age => age >=18 ).forEach(item => console.log(item));
```

Example: Filtering all the even numbers from a number array

```
let numbers = [10,11,15,17,18,19,25,44,32];



// let filteredArrays= numbers.filter( item => {

//      // if(item % 2 ===0)
//      //      return true;
//      // else
//      //      return false;

//      return item % 2 === 0;
// });



let filteredArrays= numbers.filter(item => item % 2 ===0);

console.log(filteredArrays);
```

StudentActivity: filtering and printing all the students whose marks greater than 600.

```
let students = [


    { roll: 100, name: "Ram", marks: 750 },
    { roll: 110, name: "Ramesh", marks: 850 },
    { roll: 120, name: "Suresh", marks: 450 },
    { roll: 130, name: "Varun", marks: 350 },
    { roll: 140, name: "Dinesh", marks: 550 },


];
```

```
students.filter(student => student.marks > 600).forEach(student =>
console.log(student));
```

Array.map():

This map() function is used to get a transformed array from an original array.

This map() function iterates over an array, applying a callback function to each element, and returns a new array with the results. The map() method does not change the original array, and does not execute the function for empty elements.

- It returns a new array.
- The length of the output array is same as the length of the input array.
- It takes in a callback function.
- The callback function is called for each item in the input array.
- In every iteration we have access to the current item.
- What ever you return from the callback function, becomes the item of the output array.

Syntax:

```
    array.map(function(currentItem, index, arr))
```

Example: From a numbers array return a new array with the square of all element.

```
let numbers = [2, 4, 5, 8, 6, 9, 7];

// let transformedArray= numbers.map(function(item,index, arr){

//          return item * item;

//        //    return item**2;

// });

// console.log(transformedArray);
// console.log(numbers); //not modified


// using arrow function

let transformedArray = numbers.map(item => item * item);

console.log(transformedArray);
```

Example: Transform a names array to the length array of each name.

```
let names= ["Rahul","Amit","Venkatesh","Dinesh","Chandan","Manoj"];

let lenghtArray= names.map(name => name.length);

console.log(names);
console.log(lenghtArray);
```

Example: Potato to gold:

```
let potatos= ["potato","potato","potato","potato","potato"];

let golds= potatos.map(p => "Gold");

console.log(golds);
```

Example: Given a Student object array get the array of each student marks.

```
let students = [

    { roll: 100, name: "Ram", marks: 750 },
    { roll: 110, name: "Ramesh", marks: 850 },
    { roll: 120, name: "Suresh", marks: 450 },
    { roll: 130, name: "Varun", marks: 350 },
    { roll: 140, name: "Dinesh", marks: 550 },
];


let allMarks= students.map(student => student.marks);
console.log(allMarks);
allMarks.forEach(marks => console.log(marks) );
```

Example: Transform the above student array, to put one extra entry **result**, which represent the student is pass or fail, if the marks is greater than 600 then result should be pass or result should be failed.

Solution:

```
let students = [

    { roll: 100, name: "Ram", marks: 750 },
    { roll: 110, name: "Ramesh", marks: 850 },
    { roll: 120, name: "Suresh", marks: 450 },
    { roll: 130, name: "Varun", marks: 350 },
    { roll: 140, name: "Dinesh", marks: 550 },
];
```

```
let transformedStudents = students.map(student => {

    if (student.marks > 600) {
        let newStudent = { roll: student.roll, name: student.name, marks:
student.marks, result: "Passed" };
        return newStudent;

    }
    else {
        let newStudent = { roll: student.roll, name: student.name, marks:
student.marks, result: "Failed" };
        return newStudent;
    }

});

console.log(students);
console.log(transformedStudents);
```

Student Activity: Given a products array, add one extra entry to each product called **discountedPrice**, whose value should be 10% discounted price.

```
let products = [
    { name: 'Laptop', price: 1000 },
    { name: 'Phone', price: 500 },
    { name: 'Tablet', price: 300 }
  ];
```

Hint: to apply 10% discount: price* 0.9

Student Activity: Given a student array, convert each student the employee,
Roll number should become the employeeId, and marks will become employee salary (marks * 1000)

```
let students = [

    {roll: 11, name: "Rahul", marks: 500},
    {roll: 12, name: "Anjali", marks: 600},
    {roll: 13, name: "Aditi", marks: 700},
    {roll: 14, name: "Vikash", marks: 550},
];


let employees= students.map(student => {

    let employee={
           employeeId: student.roll,
           employeeName: student.name,
           salary: student.marks*1000
      }

    return employee;
});

console.log(employees);
```

Array.reduce();

The reduce() method of Array instances performs an operation, specified by a user-provided "reducer" callback function, on each element of the array sequentially. It passes the return value from the calculation on the preceding element to the next element. Ultimately, the result of running the reducer on all elements of the array is a single value.

When the callback function is initially executed, there's no preceding return value to pass. If provided, an initial value can be used in its place. Otherwise, the element at index 0 of the array serves as the initial value, and the iteration starts from the next element (index 1 instead of index 0).

Syntax:

***array*.reduce(*function(accu, currentValue, index, arr)*, *initialValue*)**

- **function:** Required, A function to be run for each element in the array.
- **accu:** Required, The initialValue, or the previously returned value of the function.
- **currentValue:** Required, The value of the current element.
- **index:** Optional, The index of the current element.
- **arr**: Optional, The array the current element belongs to.

How the reduce() function works:

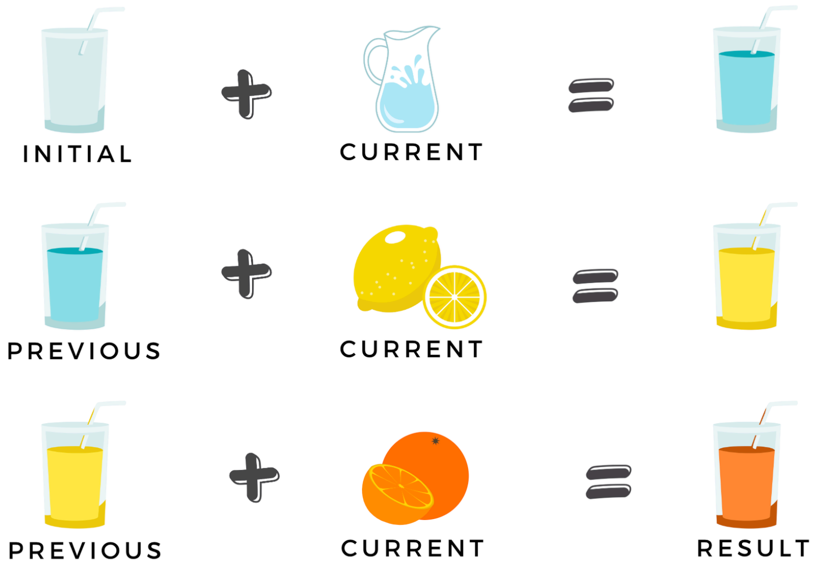
Imagine that you have a glass which is nice and empty initially

At the same time your have three ingredients right in-front of you

The end result you are looking for a glass of fresh water + lemon + orange juice

Possible way to solve:

You can use the following process to accomplish the task



This exactly is how Javascript's array.reduce works.

Example:

Assume this array **[2,4,1,7,8]** of numbers, you need to use the same iterative approach as our juice example (array.reduce() method) to find the total of all the items (numbers) in the array.

Answer the following questions:

1. How many times would be the reducer-function invoked? In other words, how many iterations would happen?
2. What should be the initial value?
3. What should be returned from the first iteration?
4. What will be the previous value (accumulated value) for the second iteration?
5. What will be the current value in the second iteration?
6. What will be the previous (accumulated) value in the third iteration?

Solution:

```
let arr = [2,4,1,7,8];

// 0 + 2 + 4 + 1 + 7 + 8

let sumWithInitial = arr.reduce( (acc, item) => acc + item, 0);
//here 0 is the initial value

console.log(sumWithInitial);
// Expected output: 22
```

Without initial value

```
let arr = [2,4,1,7,8];

// 2 + 4 + 1 + 7 + 8
```

```
      let sumWithInitial = arr.reduce( (acc, item) => acc + item);

      console.log(sumWithInitial);
      // Expected output: 22
```

Example: find the total marks from the student array

```
let students = [


    { roll: 100, name: "Ram", marks: 750 },
    { roll: 110, name: "Ramesh", marks: 850 },
    { roll: 120, name: "Suresh", marks: 450 },
    { roll: 130, name: "Varun", marks: 350 },
    { roll: 140, name: "Dinesh", marks: 550 },
];

let totalMarks = students.reduce((total, student) => {
    return total + student.marks;
  }, 0);

  console.log(totalMarks); // Output: 2950
```

Example: We are receiving data from backend in the following format:

```
let developers_array = [
  {
    first: "John",
    last: "Doe",
    dept: "FE",
    commits: 10
  },
  {
    first: "Jane",
    last: "Doe",
    dept: "BE",
    commits: 15
  },
  {
    first: "James",
    last: "bond",
    dept: "BE",
    commits: 8
  }
];
```

But at the frontend, we need to use it in the following way:

```
{
  BE: ["Jane", "James"]
  FE: ["John"]
}
```

Solution:

```
let developers_array = [
    {
        first: "John",
        last: "Doe",
        dept: "FE",
        commits: 10
    },
    {
        first: "Jane",
        last: "Doe",
        dept: "BE",
        commits: 15
    },
    {
        first: "James",
        last: "bond",
        dept: "BE",
        commits: 8
    }
];
```

```
let developersByDept = developers_array.reduce((acc, item) => {

    if (item.dept === "BE") {
        acc.BE.push(item.first);
    }
    if (item.dept === "FE") {
        acc.FE.push(item.first);
    }

    return acc;

}, { BE: [], FE: [] });


console.log(developersByDept);
```