Function in JS:-

Functions are one of the fundamental building blocks in JavaScript.

A Function definition (also called a function declaration , or function statement) consists of the function keyword ,followed by:

* The name of the function.
* A list of parameters to the function, enclosed in parentheses and separated by commas.
* The JavaScript statements that define the function, enclosed in curly brackets, {c ads}.

For example. The following code defines a simple function named sq.:

Function square(number){

Return number\* number;

}

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

Exsmple:-

function myFunction(p1, p2) {  
  return p1 \* p2;   // The function returns the product of p1 and p2  
}

JavaScript function syntax

A java script unction is define with the function keyword , followed by a name ,followd by parentheses().

Function names can contain letters, digits ,underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separaterd by commas:

(parameter1,parameter2,….)

The code to be executed, by the function,is placed inside curly brackets: {}

Example:-

function name(parameter1, parameter2, parameter3) {  
  // code to be executed  
}

function parameters are listed inside the parentheses () in the function definition.

Function arguments are the values received by the function when it is invoked.

Inside the function , the arguments( the parameters) behave as local variables.

NOTE:- A function is much same as a procedure or a subroutine, in other programming languages.

Function invocation:-

The code inside the function will execute when “something” invokes (calls) the funciomn .

* When as event occurs (when a user clicks a button)
* When it is invoked (called) from JavaScript code
* Automatically (self invoked)

Function Return:-

When JavaScript reaches a return statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will “return” to execute the code after the invoking statement.

Function often compute a return value. The return value is “returned” back to the “caller”:

Example:-

var x = myFunction(4, 3);   // Function is called, return value will end up in x  
  
function myFunction(a, b) {  
  return a \* b;             // Function returns the product of a and b  
}

the result :- 12;

Why function?

we can reuse code: Define the code once, and use it many times.

We can use the same code many times with different arguments arguments, to produce different results.

Example:-

function toCelsius(fahrenheit) {  
  return (5/9) \* (fahrenheit-32);  
}  
document.getElementById("demo").innerHTML = toCelsius(77);

Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calcuations.

Example:-

var x = toCelsius(77);  
var text = "The temperature is " + x + " Celsius";

local Variables:-

variables declared within a JavaScript function, become local to the function.

Local variables can only be accessed from with in the fi=unction.

Example:-

// code here can NOT use carName  
  
function myFunction() {  
  var carName = "Volvo";  
  // code here CAN use carName  
}  
  
// code here can NOT use carName

Since local variables are only recognized inside their functions, variables with the same name can be used in different fuctions.

Local variables are created when a function starts, and deleted when the fi=unction is completed.

Anonymous Function:-

An anonymous function is a functon without a name. An anonymous function is often not accessible after its initial creation.

An anonymous function that display a messg:-

Ex:-

# **Let show =function () {**

**Console.log(‘Anonymous function’);**

**};**

**Show();**

In this example, the anonymous function has no name between the function keyword and parentheses ().

Bcz we need to call the anonymous function later, we assign the function to the show variable.

Using anonymous function as arguments of other function

We often use anonymous function as an arguments of other ftn.

for example:

setTimeout(function () {

console.log(‘Execute later after 1 sec’)

}, 1000);

In ths example , we pass an anonymous function into the setTimeout() function. The settimeout() function this anonymous function one second later.

Note that:- function are the first-class citizens in JavaScript, so we cn pass a function to another as a argument.

Immediately invoke function execution:-

If you want to create a function and execute it immediately after declaration, you can use the anonymous function like:-

(function () {

Console.log(‘IIFE’);

})();

How it works.

First, the following defines a function expression:

(function (){

Console.log(‘immediately invoked function execution’);

})

Second, the trailing parentheses () allow you to call the function:

(function (){

Console.log(‘immediately invoke function execution’);

})();

Arrow function:-

Arrow function expression that provides a shorthand for declaring anonymous function:

For example, this function:-

Let show =function () {

Console.log(‘anonymous functionn’);

};

Now we shorted using the arrow function:-

Let show =() => console.log(‘Anonymous function’) ;

* Anonymous function are function without names.
* Anonymous functions can be used as an argument to other function or as an immediately invoked function execution.

Callback function:-

A callback function is a function passed into another function as an argument, which is then invoke inside the outer function to complete some kind of routione or action.

Example:-

Function greeting (name){

Alert (‘Hello ‘ = name);

}

Function processUserInput (callback) {

Var name =prompt (‘Please enternyour name.’);

}

processUserInputgreeting);

in this exmpl is a synchronous callback, as it is executed immediately.

Note:- callbacks are often used to continue code execution after an asynchronous operation has completed . these are called asynchronous callback. A good example is the callback functions executed inside a .then()

Block chained onto the end of a promise after that promise fulfills or rejects. This structure is used in many modern web APIs, such as Fetch()

VAR LET AND CONST:-

Var:- there are issues associated with variables declared with var , though. That is why it was necessary for new ways to declare variables to emerge. First, let’s get to understand var more before we discuss those issues.

Scope of var:- Scope essentially means these variables are aviable for use .var declarations are globally s coped or function/locally scoped.

Var is function scoped when it is declared with in a function. This means that it is available and can be accessed only with in that function.

# Ex:- var greeter =”hey hi”;

# Function newfunction() {

# Var hello =”hello”;

# }

Here greeter is a globally scoped because it exists out a function while hello is function scoped. So we cannot access the variable hello outside of a function. So if we do this:

# Var tester =”hey hi”;

# Function new function() {

# Var hello =”hello “;

# }

# Comsole.log (hello); //error hello is not defined

We’ll get an error which is an a result of hello not being available outside the function.

Var variables can be re-declare and update:-

This means that we can do this with in the same scope and won’t get an error.

## Var greeter =”hey hi”;

## Var greeter =”say hello instead”;

Hosting of var:-

Hoisting is a js mechanism where variables and function declaration are moved to the top of their scope before code execution. This means that if we do ths:-]

# Console.log (greeter);

# Var greeter =”say hello”;

So var variables are hoisted to the top of their scope and initialized with a value of undefined.

Problem with var:-

There’s a weakness that comes with var .I’ll use the example

# Var greeter =”hey hi”;

# Var times =4 ;

# If (times >3){

# Bar greeter =”say hello instead”;

# }

# Console.log(greeter)// “say hello instead”

So, since time >3 return true greeter is redefined to “say hello instead” . while this is not a problem if you knowingly want greeter to be redefined , it becomes a problem when you don’t realize that a variable greeter has already been defined before.

If you have used greeter in other parts of your code, you might be surprised at the output you might get. This will likely cause a lot of bugs in your code. This is why let  and const are necessary.

LET:-

Let is now preferred for variable declaration. Its no surprise as it comes as an improvement to var declarations .it also solves the problem with var that we just covered .let’s consider why this is do .

Let is block scoped :-

A block is a chunck of code bounded by {}. A block lives in curly braces is a block.

So a variable declared in a block with let is only available for use within that block. Let me explain this with an example:

let greeting = "say Hi";

let times = 4;

if (times > 3) {

let hello = "say Hello instead";

console.log(hello);// "say Hello instead"

}

console.log(hello) // hello is not defined

we see that using hello outside its block (the curly braces where it was defined) returns an error. This because let variables are block scoped.

**Let can be update but not re-declared.**

Just like var, a variable declared with let can be update within its scope. Unlike var, a let variable cannot be re-declared within its scope. So while this will work:

# Let f=greeting =”say hi”;

# Greeting =”say hello instead”;

This will return error:-

let greeting = "say Hi";

let greeting = "say Hello instead"; // error: Identifier 'greeting' has already been declared

however, if the same variable is defined In different scopes, there will be no error:

let greeting = "say Hi";

if (true) {

let greeting = "say Hello instead";

console.log(greeting); // "say Hello instead"

}

console.log(greeting); // "say Hi"

why there is no error? This is bcz both instances are treated as different variables since they have different scopes.

This fact makes let a better choice than var .when using let, you don’t have to bother if you have used a name for a variable before as a variable exists only within its scope.

Also, since a variable cannot be declared more than once within a scope, then the problem discussed earlier that occurs with var does not happen.

Hosting of let:-

Just like b=var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to use a let variable before declaration, you’ll get a reference Error.

**Const:-** variables declared with the const maintain constant values. Const declarations share some similarities with let declarations.

Const declaration are block scoped:-

Like let declaration, const declaration can only be accessed within the block they were declared.

Const cannot be update or re-declared:-

This means that the value of a variable declared with const remains the same with in its scope. It cannot be updated or re-declare. So If we declare a variable with const, we can neither do this:

const greeting = "say Hi";

greeting = "say Hello instead";// error: Assignment to constant variable.

Nor this:

const greeting = "say Hi";

const greeting = "say Hello instead";// error: Identifier 'greeting' has already been declared

every const declaration, therefore, must be initialized at the time of declaration.

This behavior is somehow different when it comes to objects declared with const. While a const object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a const object as this:

const greeting = {

message: "say Hi",

times: 4

}

while we cannot do this:

const greeting = {

words: "Hello",

number: "five"

} // error: Assignment to constant variable.

we can do this:

greeting.message = "say Hello instead";

This will update the value of greeting. message without returning errors.

### Hoisting of const

Just like let, const declarations are hoisted to the top but are not initialized.

So just in case you missed the differences, here they are:

* var declarations are globally scoped or function scoped while let and const are block scoped.
* var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared.
* They are all hoisted to the top of their scope. But while var variables are initialized with undefined, let and const variables are not initialized.
* While var and let can be declared without being initialized, const must be initialized during declaration.

Scope:-

In javascript there are two types of scope:

1. Local scope

2)Global scope

Local javascript:-

Variables declared within a JavaScript function, become LOCAL to the function. Local variables have **Function scope**: They can only be accessed from within the function.

Example:-

// code here can NOT use carName  
  
function myFunction() {  
  var carName = "Volvo";  
  
  // code here CAN use carName  
  
}

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local variables are created when a function starts, and deleted when the function is completed.

Global javascript variables:

A variables declared outside a function, becomes Global.

A global variable has golab scope: All scripts and function on web page can access it.

Example:-

var carName = "Volvo";  
  
// code here can use carName  
  
function myFunction() {  
  
  // code here can also use carName  
  
}

Automatically Global:-

If you assign a value to a variable tht has not been declared, it will automatically become a global variable.

This code example will declare a global variable carname, even if the value is assigned inside a function.

Example:-

myFunction();  
  
// code here can use carName  
  
function myFunction() {  
  carName = "Volvo";  
}

High order function:- A higher order function is a function that takes a function as an argument, or returns a function. Higher order function is in contrast to first order functions, which don’t take a function as an argument or return a function as output.

There are three main functional higher-order functions, which are map, filter and reduce.

Map:- Map*is a collection of elements where each element is stored as a*Key, value*pair.* Map*object can hold both*objects and primitive*values as either key or value.* When we iterate over the map object it returns the key,value pair in the same order as inserted*.*

Syntax:-

## New map([it])

## Parameter:

## It- it is any iterable object whose value are stored as key , value pair,

## If the parameter is not specified then a new map is created is Empty

## Return:-

## A new map object

Propertise:  
Map.prototype.size – It returns the number of elements or the key-value pairs in the map.  
Methods:

1. Map.prototype.set() – It adds the key and value to the Map Object.

Syntax:-

## Map.1set(k,v);

## Parameters:-

## K- key of the element to be added to the map

## v-vale of the element to be added to the map

## returns:-

## it returns a map object

**Map.prototype.has()**– It return a boolean value depending on whether the specified key is present or not

Syntax:-

## Map.1sehas(k)

## Parameters:

## K -key of the element to checked

## Returns:-

## True if the element with the specified key is present or else return false.

**Map.prototype.get()** – It returns the value of the correspondingkey

Syntax:-

## Map1.get(k);

## Paramters:

## k- key , whose value is to be returned

## returns:

## the value associated with key , if it is present in map ,otherwise returns undefined

**Map.prototype.delete() –** It delete’s both the key as well as a value from the map.  
**Syntax:**

## **Map1.delete(k);**

## **Parameters:**

## **K-key which is to be deleted from the map**

## **Returns:**

## **True if the value is found and deleted from the map otherwise , it returns false**

**Map.prototype.clear()** – Removes all the elements from the Map object

Syntax:-

## Map1.clear();

## Paramerters:

## No parameters

## Returns:

## Undefined

Filter:-

The filter() method creates an array filled with all array elements that pass a test (provided as a function).

Note: filter() does not execute the function for array elements without values.

Note: filter() does not change the original array.

## Syntax

array*.*filter*(*function(currentValue, index, arr), this Value)

Reduce:-

The **reduce**() method reduces the array to a single value. The **reduce**() method executes a provided function for each value of the array (from left-to-right). The return value of the function is stored in an accumulator (result/total).

**Note:-**reduce() does not execute the function for array elements without values.

**Note:** This method does not change the original array.

Syntax:-

*array*.reduce(function(total, currentValue, currentIndex, arr), initialValue)

JAVASCRIPTS ARRAY METHODS:-

Converting Arrays to strings:- the JavaScript’s method to string() converts an array to string of (comma separated) array values.

Example:-  
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.toString();

Result :-

Banana,Orange,Apple,Mango

The join() method also joins all array elements into a string.

It behave just like toString() , but in addition you can specify the separator:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.join(" \* ");

Result:-

Banana \* Orange \* Apple \* Mango

Popping and pushing:-

When you work with arrays ,it is easy to remove elements and add new elements

Popping is out of an array , or pushing is insert into an array.

Popping:-

The pop() metyhod removes the last element from an array:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();              // Removes the last element ("Mango") from fruits

the pop() method returns the value that was “popped out “:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
var x = fruits.pop();      // the value of x is "Mango"

Pushing:-

The push() method adds a new element to an array (at the end):

Example

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");       //  Adds a new element ("Kiwi") to fruits

the push() method returns the new array length:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
var x = fruits.push("Kiwi");   //  the value of x is 5

Shifting Elements:- shifting is equivalent to pooping, working on the first element instehd of the last .

The shift() method removes the first array element and “shifts” all other elements lower index.

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();            // Removes the first element "Banana" from fruits

the shift() method returns the string that was “shifted put”:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
var x = fruits.shift();    // the value of x is "Banana"

the unshift() method adds a new element to an array (at the beginning), and “unshifts” older elements:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");    // Adds a new element "Lemon" to fruits

the unshifted() method returns the new array length.

Changing Elements:- Array element are accessed using their index number:

Array **indexes** start with 0. [0] is the first array element, [1] is the second, [2] is the third .

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[0] = "Kiwi";        // Changes the first element of fruits to "Kiwi"

the length property provides an easy way to append a new element to array:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Kiwi";          // Appends "Kiwi" to fruits

Deleting Element:-

Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator delete:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
delete fruits[0];           // Changes the first element in fruits to **undefined**

**using delete may leave undefined holes in the array. Use pop() or shift() instead.**

**Splicing an array:-**

**The splice() method can be used to add new items to an array:**

**Example:-**

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");

**the first parameter (2) defines the position where new elements should be added spliced in).**

**the second parameter (0) defines how many elements should be removed to be added.**

**The splice() method returns an array with the deleted items:**

**Example:-**

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 2, "Lemon", "Kiwi");

Using splice() to Remove elements:-

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);        // Removes the first element of fruits

the first parameter (0) defines the position where new elements should be added (spliced in).

the second parameters are omitted. No new elements will be added.

Merging (Concatenating) Arrays :-

The concat() method creates a new array by merging (concatenating) existing arrays:

Example:- (Merging Two Arrays)

var myGirls = ["Cecilie", "Lone"];  
var myBoys = ["Emil", "Tobias", "Linus"];  
var myChildren = myGirls.concat(myBoys);   // Concatenates (joins) myGirls and myBoys

the concat() method does not change the existing arrays. It always returns arrays. It always returns a new array.

The concat() method cantake any number of array arguments:

Example (merging three arrays)

var arr1 = ["Cecilie", "Lone"];  
var arr2 = ["Emil", "Tobias", "Linus"];  
var arr3 = ["Robin", "Morgan"];  
var myChildren = arr1.concat(arr2, arr3);   // Concatenates arr1 with arr2 and arr3

the concat() method can also take strings as argumennts:-

Example:-(merging an array with values)

var arr1 = ["Emil", "Tobias", "Linus"];  
var myChildren = arr1.concat("Peter");

Slicing an array:-

The slice() method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1(“Orange”):

Example:-

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1);

the slice() method creates a new array. It does not remove any elements from the source array.

This example slice out a part of an array starting form array element 3 (“apple”):

Example:-

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(3);

the slice() method can take two arguments like slice(1,3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

Example:-

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1, 3);

if the end argument is omitted, like in the first examples, the slice() method slices out the rest array.

Example:-

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(2);

Automatic to string():-

JavaScript automatically converts an array to a comma separated string when a primitive value is expected.

This is always the case when you try to output an array.

These two examples will produce the same result:

Example:-

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits;

All JavaScript object have a to string() method.

## Finding Max and Min Values in an Array:-

There are no built-in functions for finding the highest or lowest value in a JavaScript array.

You will learn how you solve this problem in the next chapter of this tutorial.

## **-Complete Array Reference:-**

For a complete reference, go to our [Complete JavaScript Array Reference](https://www.w3schools.com/jsref/jsref_obj_array.asp).

The reference contains descriptions and examples of all Array properties and methods.

Number method And Properties:-

Number method help to work with numbers.

Primitive values (like#.2 or 2013), cannot hanve properties and methods (because they are not objects).

But with javascipt, method and properties are also available to primitive values, because javascript treats primitive values as objects when executing methods and properties.

The toString() method:-

the toString()( method returns as a string. All number methods can be used on any type of numbers(literals, variables,or expressions):

Example:-

var x = 123;  
x.toString();            // returns 123 from variable x  
(123).toString();        // returns 123 from literal 123  
(100 + 23).toString();   // returns 123 from expression 100 + 23

the toExponential() method:-

toExponential()returns a string, with a number rounded and written usig exponential notation.

A parameter defines the number of characters behind the decimal point:-

Example:-

var x = 9.656;  
x.toExponential(2);     // returns 9.66e+0  
x.toExponential(4);     // returns 9.6560e+0  
x.toExponential(6);     // returns 9.656000e+0

the parameter is optional. If you don’t specify it. javascript will not round the number.

The toFixed() method:-

To fixed() return a string, with the number written a specified number of decimals:

Example:-

var x = 9.656;  
x.toFixed(0);           // returns 10  
x.toFixed(2);           // returns 9.66  
x.toFixed(4);           // returns 9.6560  
x.toFixed(6);           // returns 9.656000

tofixed(2) is perfect for working with money.

The toPrecision() method:-

toPrecision() returns a string, with a number written with a specified length:

Exampe:-

var x = 9.656;  
x.toPrecision();        // returns 9.656  
x.toPrecision(2);       // returns 9.7  
x.toPrecision(4);       // returns 9.656  
x.toPrecision(6);       // returns 9.65600

the ValueOf() method:-

ValueOf() returns a number as a number.

Example:-

var x = 123;  
x.valueOf();            // returns 123 from variable x  
(123).valueOf();        // returns 123 from literal 123  
(100 + 23).valueOf();   // returns 123 from expression 100 + 23