**INTRODUCTION**

JavaScript is a cross-stage, question situated scripting dialect used to make site pages intelligent (e.x. having complex liveliness, interactive catches, popup menus, and so forth.). There are additionally further developed server side forms of javascript, for example, Node.Js which enable you to add more usefulness to a site than basically downloading documents, (for example, realtime joint effort between numerous PCs). Inside a host situation (for instance, a web program), JavaScript can be associated with the objects of its condition to give automatic control over them.

**JAVASCRIPT FUNDAMENTALS**

1. **Code Structure**

TO-DO: Analyze well the building blocks of every code.

1. *Statements*

* These are language structure builds and charges that perform activities. An example articulation **alert('Web, Technologies!')** which demonstrates the message. Another announcement can be isolated with a semicolon (a). Normally every announcement is composed on a different line which turns out to be more discernable (b).

1. **alert(‘Web’); alert(‘Technologies’);**
2. **alert(‘Web’);**

**alert(‘Technologies’);**

1. *Semicolons*

* May be precluded by and large when a line break happens as an “implicit” semicolon and known as automatic semicolon insertion.

1. **alert(‘Web’);**
2. **alert(‘Technologies’);**
3. *Comments*

* This can be put into wherever of the content, they don't influence the execution in light of the fact that the motor basically overlooks them.

NOTE:

One-line remarks begin with 2 forward slash characters

1. Multiline remarks begin with a forward slash and a mark/\* and end with a reference bullet and a forward slash \*/.

//Occupies its very own line

alert('Web');

**//Follows the statement**

**alert(‘Technologies’);**

1. **/\* Sample of a multiline comment. \*/**
2. **alert(‘Web’);**

**alert(‘Technologies’);**

1. **The modern mode (“use strict”)**

* It had been so until 2009 when ECMAScript 5 (ES5) appeared. It added new features to the tongue and changed a bit of the present ones. To keep the old code working, most modifications are off as per normal procedure. One needs to engage them unequivocally with a remarkable order "utilize strict".

**"use strict" OR ‘use strict'**

* The whole content works the "advanced" way, in the event that it is situated on the highest point of the content

Syntax:

“use strict”;

// Works on modern way

Conditions:

1. Make sure that the “use strict” is at the top

* This must be applied so that the strict mode will be enabled.

No strict mode:

alert(“sample code”);

// “use strict” IGNORED since it is not located at the top

“use strict”

// strict mode is not activated

NOTE: Comments may only appear above the “use strict”

1. No way for canceling the “use strict”

**Always “use strict”**

1. The "use strict" mandate changes the motor to the "modern" mode, changing the conduct of some implicit highlights. We'll see the points of interest as we examine.
2. The strict mode is empowered by "use strict" at the best. Likewise there are a few dialect highlights like "classes" and "modules" that empower strict mode naturally.
3. The strict mode is bolstered by every modern program.
4. It's constantly prescribed to begin contents with "use strict".

**3. Variables**

* JavaScript application needs to work with data.
* Online shop - It may incorporate sold products and shopping basket
* Visit applications - Includes messages, clients et cetera

A variable is a "named stockpiling" for crude information. Making a variable in JavaScript, you have to utilize "let" watchword.

let message;

We would now be able to put an example information into it by applying the task administrator (=):

let message;

message = 'Hi'; // string will be put away

The string is directly saved into the memory territory related with the variable. We can get to it utilizing the variable name:

let message;

message = ‘Hello’; //string will be stored

alert(message); // displays the variable content

To be compact we can combine the variable announcement and task into a single line:

let message = ‘Hello’; //defining the variable and assigned value

alert(message); // Hello

We can likewise proclaim numerous factors in a single line:

let user = ‘Maria’, age = 20, message = ‘Hi’;

That may appear to be shorter, however it's not suggested. For better coherence, please use one line for every factor. The multi-line variation is somewhat more, yet less demanding to read:

let user = ‘Maria’;

let age = 20;

let message = ‘Hi’;

Others write many variable like this:

let user = ‘Maria’,

age = 20,

message = ‘Hi’;

1. **var instead of let**

You may find another keyword in older scripts: **var** instead of **let**

var message = ‘Hello’;

This catchphrase is 'nearly' the same as let. In an 'old-school' mold, it additionally proclaims a variable.

**Real-life Application**

We can without much of a stretch handle the idea of a "variable" in the event that we envision it as a "container" for information, with a particularly named sticker on it.

For example, the variable message can be envisioned as a crate named "message" with the esteem "Hello!" in it:



You can put any an incentive into the case. The esteem can be changed in commonly as required.

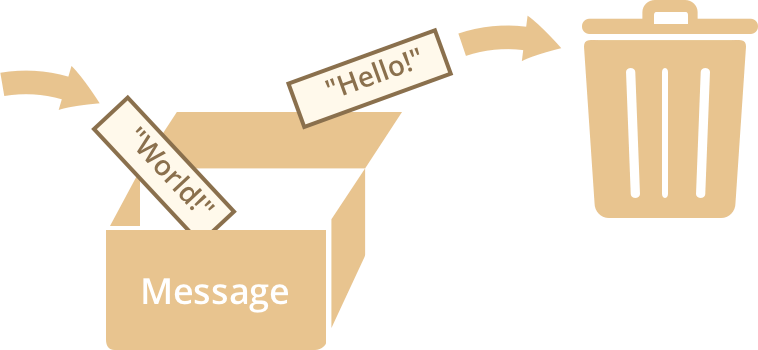
let message;

message = ‘Hello!’;

message = ‘World!; //changed value

alert(message);

As the value is changed, old data is being removed from the variable:



Declaring two variables and copying of data from one into the other can also be done.

let hello = ‘Hello world’;

let message;

// copy ‘Hello world’ from helo into message

message = hello;

//2 factors now hold similar information

alert(hello);

alert(message); //Hello World!;

**b. Functional Languages**

* In such language, once the value is put away in the "box", it's there for eternity. In the event that we have to store something different, the language compels us to make another case (announce another variable).

**c. Variable naming**

Limitations:

1. The name contains just letters, digits, images $ and \_;
2. First character must not be a digit.

Valid names:

let userName;

let test123;

Dollar sign ($) and underscore (\_) can likewise be utilized as a part of names. They are general images with no uncommon significance.

let & = 1; // variable with the name “$”

let \_ = 2; // variable with the name “\_”

alert($ + \_); // 3

Incorrect variable names:

let 1a; // can’t begin with a digit

let my-name; // a hyphen ‘-’ is not permitted in the name

**d. Case-sensitivity**

Variables name Hello and HeLLo are two different variables.

**e. Non-english letters are permitted, yet not prescribed**

let имя = '...';

let 我 = '...';

**f. Reserved names**

* They are utilized by the dialect itself including let, class, return, function

let let = 5; // cannot name a variable “let”, error!

let return = 5; // cannot name it “return”, error!

**g. A task without use strict**

* Ordinarily, we need to portray a variable before using it. In any case, in the old conditions, it was in truth possible to make a variable by an irrelevant assignment of the esteem, without let. This still works now if we don't put utilize strict.

// note: no “use strict”

num = 10; // variable “num” is created if didn’t exist

alert(num); //10

**4. Constants**

* Declaring a constant variable, one can utilize const rather than let:

const myDateOfBirth = “11.12.1998”

It can't be changed in light of the fact that an endeavor to do it would cause an error.

const myDateOfBirth = “11.12.1998”

myDateOfBirth = “01.01.2011” // ERROR! Reassigning the constant is not allowed.

1. Capitalized constants

* There is a wide practice to use constants as accepted names for difficult to-review regards that are known before execution. Such constants are named using capital letters and underscores.

const COLOR\_RED = “#F00”;

const COLOR-GREEN = “#0F0”;

const COLOR\_BLUE = “#00F”;

// when picking a certain color

let color = COLOR\_RED;

alert(color); // #F00

**5. Data types**

* It can be a string and afterward get a numeric value.

//no error

let message = “web technologies”;

message = 09876;

1. Number

* Serves for whole number and drifting point numbers

**Operation for numbers:** multiplication (\*), division (/), addition (+), subtraction (-)

let n = 321;

n = 23.23;

Special numeric values: Infinity, -Infinity, NaN

* Infinity - it is a mathematical infinity loop, also a special value which is greater than other number.

alert( 1/0 ); //Infinity

alert(Infinity); //Direct saying

* NaN - This is an aftereffect of a wrong/indistinct scientific task and speaks to computational mistake.

alert( “not a number” / 2 ); // Division is erroneous

alert( “not a number” / 2 + 5 ); // NaN

1. String

let str = “Hi”;

let str2 = 'Single statements';

let phrase = 'can implant ${str};

There are 3 types of quotes:

1. Double quote: “Hello”
2. Single quote: ‘Hello’
3. Backticks `Hello`

c. Boolean (Logical Type)

* Two values (true and false)

let nameFieldChecked = false; // no, name field is not checked

let ageFieldChecked = true; // yes, age field is checked

d. “Null” value

* It doesn’t belong to any type of the above-mentioned. It is just a special value meaning “nothing”, “empty”, or “unknown value”

let age = null;

e. Undefined value

* It makes its own, just like null meaning “value is not assigned”

let x =123;

x = undefined;

alert(x); //undefined

f. Objects and Symbols

* Object: It is special and called “primitive” which has a value that contains only a single thing (string/number). Conversely, it is used to store accumulations of information and more perplexing elements.
* Symbol: Used for creating a unique identifiers.

g. Typeof operator

* Returns the sort of the contention. It is helpful when we need to process estimations of various sorts in an unexpected way, or simply need to make a speedy check.

Supports two types of sentence structure:

1. Operator: typeof x
2. Function style: typeof(x)

The call to typeof x restores a string with the sort name:

typeof undefined // “undefined”

typeof 0 // “number”

typeof true // “boolean”

typeof “foo” // “string”

typeof Symbol(“id”) // “symbol”

typeof Math // “object” (1)

typeof null // “object” (2)

typeof alert // “function”

**6. Type conversions**

* Operators and functions that automatically convert a value to its right type.

1. toString

* Happens when you need the during the form of a value and is mostly obvious. A false becomes “false”, null become “null” and so forth.

let value = true;

alert(typeof value); //boolean

value = String(value); // string value is true

alert(typeof value); //string

1. toNumber

* Occurs in numerical capacities and articulations consequently

Eg. at the point when division is connected to non-numbers

alert( “6” / “2” ); // 3, strings are changed to numbers

Eg. utilizing a Number(value) function

let str = “1234”;

alert(typeof str); // string

let num = Number(str); // converted to the number 1234

alert(typeof num); // number

1. toBoolean

* It is the simplest among others. t occurs in coherent tasks however can be performed physically with the call of Boolean(value).

Conversion rule:

* Values are "unfilled" like 0, a void string, invalid, unclear, and NaN turns out to be false.
* Other become true or correct.

alert(Boolean(1)); // true

alert(Boolean(0)); // false

alert(Boolean(“hello”)); // true

alert(Boolean(“”)); // false

NOTE: String with zero (0) is true

alert (Boolean(“0”)); // true

alert(Boolean(“ ”)); // spaces are also true (non-empty string)

**7. Operators**

* Addition (+) , Multiplication (\*) , Subtraction (-) , Division (/) etc.

**Terms: “unary” , “binary” , “operand”**

* Unary - if it has a single operand

Eg. unary negation - reverses the operation of the number

let x = 1;

x = -x;

alert( x ); // -1 (negation applied)

* Binary - it is has two operands, the same minus exist in binary form

let x - 1, y = 3;

alert(y - x); // 2 (binary minus structs values)

**Connection of string, binary +**

Usually the in addition to administrator + wholes numbers yet in the event that the parallel + is connected to strings,it links them.

let s = “my” + “string”;

alert(s); // mystring

Note that any of the operands is a string then the other is changed over to a string as well.

alert( ‘1’ + 2 ); // “12”

alert( 2 + ‘1’ ); // “21”

alert( 2 + 2 + ‘1’ ); “41” not “221”

alert( 2 - ‘1’ ); // 1

alert( ‘6’ / ‘2’ ); // 3

**Conversion of numbers, unary +**

Plus + exists in 2 forms (binary and unary form). The unary in addition to or in addition to administrator + is connected to a solitary esteem, and does not do anything with numbers, but rather if the operand isn't a number, at that point it can be changed over.

// no effects on the numbers

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// converts non-numbers

alert( +true); // 1

alert( +”” ); // 0

**Administrators priority**

Priority Name Sign

16 Unary plus +

16 unary negation -

14 multiplication \*

14 division /

13 addition +

13 subtraction -

3 assignment =

**Assignment (=)**

* This administrator dependably restores an esteem. It is clear for the vast majority of them like an expansion + or an augmentation \* and takes after that administer too.

The call x = value writes the value into x and then returns.

let a = 1;

let b = 2;

let c = 3 - (a = b + 1);

alert(a); // 3

alert(c); // 0

**Remainder (%)**

* Despite of its look, it doesn't have a connection to percents. The consequence of a % b is the rest of the whole number division of a by b.

alert( 5 % 2 ); // 1 is a remainder of 5 divided by 2

alert( 8 % 3 ); // 2 is a remainder of 8 divided by 3

**Exponentiation (\*\*)**

* It is a current expansion to the dialect.

Eg. For the normal b, the consequence of a \*\* b is an increased without anyone else's input b times.

alert( 2 \*\* 2 ); // 4 ( 2 \* 2 )

alert( 2 \*\* 3 ); // 8 ( 2 \* 2 \* 2 )

**Increment/decrement**

* **Increment ++** expands a variable by 1

let counter = 2;

counter++; // works the same as counter = counter + 1

alert(counter); // 3

* **Decrement --** diminishes a variable by 1

let counter = 2;

counter--;

alert(counter); // 1

**Bitwise operators**

* They regard contentions as 32-bit whole number numbers and work on the level of their double portrayal.

List of operators:

* And ( & )
* Or ( | )
* Xor ( ^ )
* Not ( ~ )
* Left shift ( << )
* Right shift ( >> )
* zero-fill right shift ( >>> )

**Modify -in-place**

* Often need to apply an administrator to a variable and store the new outcome in it

For example:

Let n = 2;

n = n + 5;

n + n \* 2;

Notations /Documentations can be shortened using operators += and \*=

let n = 2;

n += 5; // now n = 7 (same as n = n + 5)

n \*=2; // now n = 14 (same as n = n \* 2)

alert(n); // 14

**Comma**

* One of the most uncommon and unusual administrators and used to compose shorter code.

let a = (1+2, 3+4)

alert(a); // 7 (result of 3+4)

**Note:** Comma has a very low priority

**8. Comparisons**

1. greater/less than a > b , a < b
2. greater/less than or equals a >= b , a <= b
3. Equality check a == b
4. Not equal a != b

**Boolean is the result**

* A comparison returns a value.

True - means “yes”, “correct”, or “truth”

False - means “no”, “wrong” or “lie”

alert( 2 > 1 ); // true (correct)

alert( 2 == 1 ); // false (wrong)

**String comparison**

* It is greater than the others, hey are called “dictionary” or “lexicographical” order is used. Strings are compared letter-by-letter.

alert( ‘Z’ > ‘A’ ); // true

alert( ‘Thought’ > ‘Taught’ ); //true

Note: Not a genuine lexicon, but rather Unicode arrangement

**Correlation of various types**

* When contrasting esteems that has a place with various kinds, it will be changed over to numbers.

alert( ‘3’ > 2 ); // true, string becomes a number 2

alert( ‘04’ == 4 ); true, string ‘01’ becomes a number 1

Note: An interesting outcome

* It is equivalent in the event that it has two values.
* One of them is valid and the other is false, and both are boolean.

**Strict uniformity**

* Customary uniformity check == has an alternate. It can't vary 0 from false

alert( 0 == false ); // true

alert( ‘ ’ == false ); // genuine/ true (empty string)

**Comparison with null and undefined**

* There is a non-natural conduct when invalid or indistinct are contrasted and different values.

1. Strict uniformity check ==

alert(null === undefined); // false

1. Non-strict check ==

alert(null == undefined); // true

1. Maths and different correlations < > <= >=

* Values null/undefined are changed over to a number: null becomes 0, while undefined becomes NaN.

**Peculiar outcome: null vs 0**

alert( null > 0); // false

alert( null ==0); // false

alert( null >= 0); // true

**Incomparable undefined**

* This must not collaborate in examinations by any means

alert( null > 0); // false

alert( null < 0); // false

alert( undefined == 0); // false

**9. Interaction**

1. Alert

Syntax: alert(message);

Eg. alert(“Hello!”);

1. Prompt

* Accepting two arguments

result = prompt(title[, default]);

1. Confirm

* It shows a modal window with a question and two buttons (OK and CANCEL)
* Accepting two arguments

result = prompt(title[, default]);

**10. Logical operators**

* Three logical operators in JavaScript: || (or) , && (and) , ! (not)

1. **|| (OR)**

* Represented with two vertical line symbols

result = a | b;

Four possible logical operations:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

1. **&& (AND)**

* Represented with two ampersands &&. In classical programming AND returns true if both operands are truthy or otherwise false

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

Example of if:

let hour = 12;

let minute = 30;

if (hour == 12 && minute == 30) {

alert( 'Time is 12:30' );

}

1. **! (NOT)**

* Boolean NOT operator is represented with an exclamation sign !

result = ! value;

The operator accepts a single argument and does the following:

* Converts the operand to boolean type: true/false.
* Returns an inverse value.

alert( !true ); // false

alert( !0 ); // true

A double NOT !! is sometimes used for converting a value to boolean type:

alert( !!"non-empty string" ); // true

alert( !!null ); // false

**11. Loops: while, do..while, for**

* The syntax of these three loops are the same with java programming language.

SYNTAX:

while (condition) {

// code

// so-called "loop body"

}

do {

// loop body

} while (condition);

for (begin; condition; step) {

// ... loop body ...

}

**FUNCTIONS**

Syntax:

function name(parameters, delimited, by, comma) {

/\* code \*/

}

**FUNCTION DECLARATION**

We can create a function using the function declaration.

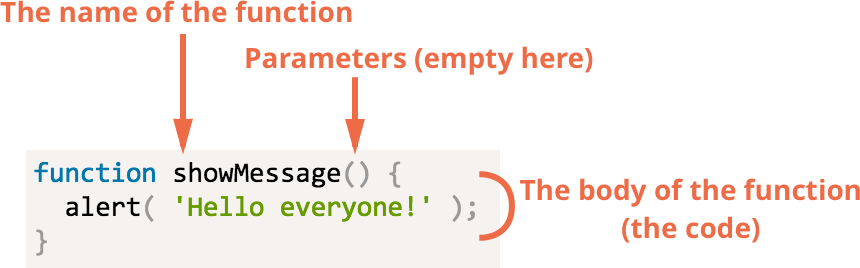
The function declaration looks like this:

function showMessage(){

alert( ‘Web Systems and Technologies’);

}

First is the function keyword, then the name of the function, then a sort of parameter between the parentheses, lastly the code of the function, likewise named the “function body” between wavy or curly braces.



New function: showMessage( )

* It executes the code of function and the main purpose of function is to avoid duplication of code.

function showMessage( ) {

aler(‘Web Technologies’);

}

showMessage( );

showMessage( );

**LOCAL VARIABLES**

* It is only visible inside that function if a variable is declared inside

function showMessage( ) {

let message = "Hello, this is a sample message!"; // local variable

alert( message );

}

showMessage( ); // Hello, this is a sample message!

alert( message ); // <-- Errors since the variable is local

**OUTER VARIABLES**

* Accessed when a function is an outer variable. This is only used when there is no local variables.

let userName = 'Maria';

function showMessage() {

let message = 'Hi, ' + userName;

alert(message);

}

showMessage(); // Hi, Maria

* The function can be modified and has a full access to the outer variable

let userName = 'Maria';

function showMessage() {

userName = "Peter"; // (1) changed the outer variable

let message = 'Hi, ' + userName;

alert(message);

}

alert( userName ); // Maria before the function call

showMessage();

alert( userName ); // Peter is the value that was modified by the function

NOTE: Global variables are declared outside and visible from any other functions.

**PARAMETERS:**

* Function arguments can pass the arbitrary data to function when using a parameter.

Example of a function that has two parameters (from and text)

function showMessage(from, text) { // arguments: from, text

alert(from + ': ' + text);

}

showMessage('Bernie', 'Thank You!'); // Bernie: Thank You! (\*)

showMessage('Benny', "You’re welcome."); // Benny: You’re welcome. (\*\*)

* A certain values are duplicated to local variables from and text when a function is called in line (\*) and (\*\*) then it uses them.

**DEFAULT VALUES**

* A value will become undefined if a parameter is not given.

For example, the example function showMessage(from, text) is called with a single argument.

showMessage(“Benny”);

It has no error since it has no text in it so it’s assumed that text == undefined.

function showMessage(from, text = "no text provided") {

alert( from + ": " + text );

}

showMessage("Benny"); // Benny: no text provided

NOTEl Default parameters old-style are a century editions of JavaScript which does not support default parameters. There are other ways in supporting them like:

1. Explicit check that is being undefined

function showMessage(from, text) {

if (text === undefined) {

text = 'no text provided;

}

alert( from + ": " + text );

}

1. || operator

function showMessage(from, text) {

// text gets the "default" value if it is incorrect

text = text || 'no text provided;

...

}

**RETURNING A VALUE**

* As an outcome, the function will return a value back into the calling code

function sum(x, y) {

return x + y;

}

let result = sum(13, 2);

alert( result ); // 15

NOTE: Function that has an empty return or returns undefined without it

function doNothing() { /\* empty \*/ }

alert( doNothing() === undefined ); // true

* Then never add a new line between a return and the value

return

(some + long + expression + or + whatever \* f(a) + f(b))

**NAMING A FUNCTION**

* They are accurately describe what the function does that it why functions are actions.

Functions that starts with:

* “get…” - a value is returned
* “calc…” - something is being calculated
* “create…” - something is being created
* “check…” - something is being checked and returned as a boolean, etc.

NOTE: One function - one action

* It should exactly do what is being suggested by the name, nothing more.

Examples in breaking the rule:

* getAge - if it displays an alert message with the age which would be bad
* createForm - if it modifies the document, adding a form should only create it and return
* checkPermission - if it presents the access granted/denied message and must only shows he check and the result will be returned.

Ultrashort function names

* Functions are being utilized very often

**FUNCTIONS == COMMENTS**

* It must be short and do exactly the same thing. If it is huge, it might be worth it to break the function into smaller functions.

1. Used a label

function showPrimes(x) {

nextPrime: for (let i = 2; i < x; i++) {

for (let y = 2; y < i; y++) {

if (i % y == 0) continue nextPrime;

}

alert( i ); // prime

}

}

b. Additional function: isPrime(x)

function showPrimes(x) {

for (let z = 2; z < x; z++) {

if (!isPrime(z)) continue;

alert(z); // prime

}

}

function isPrime(x) {

for (let z = 2; z < x; z++) {

if ( x % z == 0) return false;

}

return true;

}

* This is easier to be understood because instead of the code piece, we can see the name of the action (isPrime) which is sometimes refer to as self-describing.