JAVA SCRIPT

- LANGUAGE BASICS

* ADDING INLINE SCRIPT :-

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title> Section2: JAVASCRIPT Language Basics </title>
    </head>
    <body>
        <h1> Section2: JAVASCRIPT Language Basics </h1>
    </body>
    <script>
        console.log("Hello World");
    </script>
</html>
```

→ We use ";" to end that line on code.

* ADDING EXTERNAL SCRIPT :-

Create a new file & save as "script.js" & then what we do is

```
<script src= script.js> </script>
```

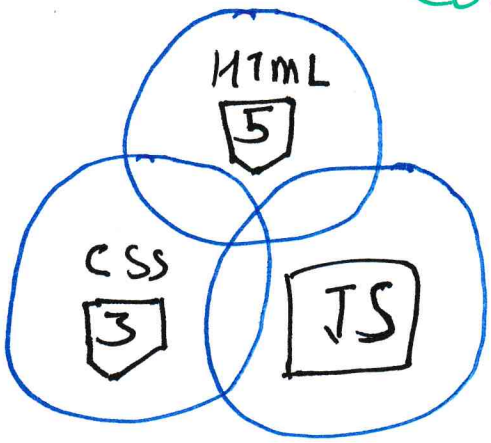
This code we write in .html file &

```
console.log("Himanshu Malik");
```

This code we write in script.js

* Introduction To JavaScript :-

CONTENT
[The para., words, buttons & images]



PRESENTATION
[The style, reference, color]

DYNAMIC EFFECTS / PROGRAMMING

HTML 5 — NOUNS

means "Paragraph"

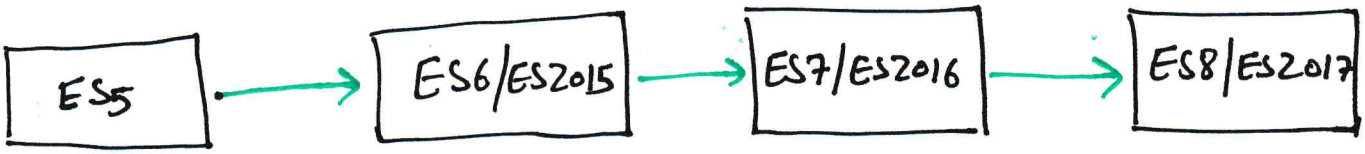
CSS 3 — ADJECTIVE

p{color: red;}
means "Paragraph text is red"

JS — VERBS

p.hide();
means "hide the Paragraph"

* JavaScript Version :-



VARIABLES AND DATA TYPES:-

To, define a variable in javascript,

```
var myName = 'HIMANSHU MALIK';
console.log (myName);
```

We are doing these things in file name "script.js" //

JAVASCRIPT DATA TYPES :-

1. NUMBER
2. STRING
3. BOOLEAN
4. UNDEFINED ⇒ Data type of a variable that doesnot have a value.
5. NULL ⇒ NON-existent.

```
var job;
console.log (job);
```

When we print the answer, the answer will be undefined.

∴ SO that's how it's work.

* Variable Mutation And Type Coercion:-

script.js

```
var name = 'Himanshu_Malik';
var age = 20;
console.log(name + ' ' + age);
```

Result in console will be:-

>> Himanshu_Malik 20

So, this would be possible because of Type Coercion, which itself detects which is string & which is number.

```
var job, isMarried;
job = 'teacher';
isMarried = false;
console.log(name + ' is a ' + job + '. He is ' + age + ' years old ' + 'and
    Married ' + isMarried);
```

And if we put alert instead of putting console.log then, new window notification will pop out instead of coming in console.

var lastName = prompt(`What's Your last Name??`);
console.log(firstName + ' ' + lastName);

So, we use prompt to store the variable.
For example, in the above one we enter the last name Malik so it will come in the console with firstName i.e.

HIMANSHU_MALIK MALIK

BASIC OPERATORS —

⇒ We are writing in (script.js) ⇒ File

```
var year, yearJohn, yearMark;
year = 2020;
yearJohn = year - 20;
yearMark = year + 40;

console.log(yearJohn);
console.log(yearMark);
                                    // Math Operator
console.log(year + 2);
console.log(year - 2);
console.log(year * 2);
console.log(year / 2);
```

```
>> 2000
   2060
   2022
   2018
   4040
   1010
```

// Logic Operator

~~var year, yearJohn, yearMa~~

```
var year, yearJohn, yearMark;
year = 2020;
ageJohn = 38;
ageMark = 40;

yearJohn = year - ageJohn;
yearMark = year - ageMark;
var johnOlder;
console.log(ageJohn
johnOlder = ageJohn < ageMark;
console.log(johnOlder);
```

>> true

// typeof Operator

```
console.log (typeof johnOlder);
console.log (typeof ageJohn);
console.log (typeof "How is it");
.
var x;
console.log (typeof x);
```

>> boolean
number
string
undefined

* Operator Precedence :-

```
var now, yearJohn, fullAge;
now = 2018;
yearJohn = 1989;
fullAge = 18;

var isFullAge = now - yearJohn >= fullAge
Console.log(isFullAge);
```

» true

```
var ageJohn = now - yearJohn;
var ageMark = 35;
var average = (ageJohn + ageMark) / 2;
Console.log(average);
```

So, in this we use grouping precedence first.

» 32

* For check the precedence check the ~~list~~ Table.

Next Page ⇒

```
var x, y;
x = (3+5) * 4-6;
console.log(x);
```

» 26

`** 2` is the same as `2 ** (3 ** 2)`. Thus, doing `(2 ** 3) ** 2` changes the order and results in the 64 seen in the table above.

Remember that precedence comes before associativity. So, mixing division and exponentiation, the exponentiation comes before the division. For example, `2 ** 3 / 3 ** 2` results in 0.8888888888888888 because it is the same as `(2 ** 3) / (3 ** 2)`.

Note on Grouping and Short-Circuiting

In the table below, **Grouping** is listed as having the highest precedence. However, that does not always mean the expression within the grouping symbols `( … )` is evaluated first, especially when it comes to short-circuiting.

Short-circuiting is jargon for conditional evaluation. For example, in the expression `a && (b + c)`, if `a` is "falsy", then the sub-expression `(b + c)` will not even get evaluated, even if it is in parentheses. We could say that the logical disjunction operator ("OR") is "short-circuited". Along with logical disjunction, other short-circuited operators include logical conjunction ("AND"), nullish-coalescing, optional chaining, and the conditional operator. Some more examples follow.

```
a || (b * c);  // evaluate `a` first, then produce `a` if `a` is "truthy"
a && (b < c);  // evaluate `a` first, then produce `a` if `a` is "falsy"
a ?? (b || c); // evaluate `a` first, then produce `a` if `a` is not `null` and not `undefined`
a?.b.c;        // evaluate `a` first, then produce `a` if `a` is `null` or `undefined`
```

Examples

```
3 > 2 && 2 > 1
// returns true

3 > 2 > 1
// Returns false because 3 > 2 is true, then true is converted to 1 in inequality operators, therefore
//  is false. Adding parentheses makes things clear: (3 > 2) > 1.
```

Table ⇒ OPERATOR PRECEDENCE

The following table is ordered from highest (21) to lowest (1) precedence.

Precedence	Operator type	Associativity	Individual operators
21	Grouping	n/a	`( … )`
20	Member Access	left-to-right	`… . …`
	Computed Member Access	left-to-right	`… [ … ]`
	`new` (with argument list)	n/a	`new … ( … )`
	Function Call	left-to-right	`… ( … )`
	Optional chaining	left-to-right	`?.`
19	`new` (without argument list)	right-to-left	`new …`
18	Postfix Increment	n/a	`… ++`
	Postfix Decrement		`… --`
17	Logical NOT	right-to-left	`! …`

	Bitwise NOT		`~ …`
	Unary Plus		`+ …`
	Unary Negation		`- …`
	Prefix Increment		`++ …`
	Prefix Decrement		`-- …`
	`typeof`		`typeof …`
	`void`		`void …`
	`delete`		`delete …`
	`await`		`await …`
16	Exponentiation	right-to-left	`… ** …`
15	Multiplication	left-to-right	`… * …`
	Division		`… / …`
	Remainder		`… % …`
14	Addition	left-to-right	`… + …`
	Subtraction		`… - …`
13	Bitwise Left Shift	left-to-right	`… << …`
	Bitwise Right Shift		`… >> …`
	Bitwise Unsigned Right Shift		`… >>> …`
12	Less Than	left-to-right	`… < …`
	Less Than Or Equal		`… <= …`
	Greater Than		`… > …`
	Greater Than Or Equal		`… >= …`
	`in`		`… in …`
	`instanceof`		`… instanceof …`
11	Equality	left-to-right	`… == …`
	Inequality		`… != …`
	Strict Equality		`… === …`
	Strict Inequality		`… !== …`
10	Bitwise AND	left-to-right	`… & …`
9	Bitwise XOR	left-to-right	`… ^ …`
8	Bitwise OR	left-to-right	`… | …`
7	Nullish coalescing operator	left-to-right	`… ?? …`
6	Logical AND	left-to-right	`… && …`
5	Logical OR	left-to-right	`… || …`
4	Conditional	right-to-left	`… ? … : …`
3	Assignment	right-to-left	`… = …`
			`… += …`
			`… -= …`
			`… **= …`
			`… *= …`
			`… /= …`
			`… %= …`

			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …
			… |= …
2	`yield`	right-to-left	`yield …`
	`yield*`		`yield* …`
1	Comma / Sequence	left-to-right	… , …

Last modified: Jun 5, 2020, by MDN contributors

Related Topics

JavaScript

Tutorials:

- ▸ Complete beginners
- ▸ JavaScript Guide
- ▸ Intermediate
- ▸ Advanced

References:

- ▸ Built-in objects
- ▾ Expressions & operators
  - Arithmetic operators
  - Assignment operators
  - Bitwise operators
  - Comma operator
  - Comparison operators
  - Conditional (ternary) operator
  - Destructuring assignment
  - Function expression
  - Grouping operator
  - Logical operators
  - Nullish coalescing operator
  - Object initializer
  - Operator precedence
  - Optional chaining
  - Pipeline operator
  - Property accessors
  - Spread syntax
  - async function expression
  - await
  - class expression
  - delete operator
  - function* expression

```
x *= 2;
console.log(x);
x += 10;
console.log(x);
x++;
console.log(x);
```

```
>> 52
   62
   63
```

If / Else Statements :-

```
var name = 'John';
var status = 'Single';
if ( status === 'Married') {
    console.log(name + 'You Are Married');
  }
else {
   console.log (name + 'You Are Single');
   }
```

>> John You Are Single

```
var isMarried = true;
if (isMarried) {
    console.log (name + 'Married');
   }
else {
    console.log(name + 'Single');
   }
```

>> John Married

BOOLEAN LOGIC :-

- AND (&&) ⇒ true if ALL are true.
- OR (||) ⇒ true if ONE is true.
- NOT (!) ⇒ inverts true/false value.

```
var firstName = 'John';
var age = 16;

if (age <= 13 && age >= 20) {
    console.log (firstName + ' you r taller');
}
else if (age == 16 || age == 20) {
    console.log (firstName + ' you r 16');
}
else {
    console.log (firstName + ' you r smaller');
}
```

>>

John You r 16

The Ternary Operator And Switch Statements

```
var name = 'John';                    // TERNARY OPERATOR
var age = 20;
age >= 18 ? console.log(name + ' Drinks Beer!!')
: console.log(name + ' Drinks Juice!!');
```

» John Drinks Beer!!

So, in ternary Operator, we can write if & else statements just in one line.

? ⇒ If
: ⇒ else

```
var drink = age >= 18 ? 'beer' : 'Juice';
console.log(drink);
```

» Beer

// Switch Statements

```
var job = 'teacher';
switch(job) {
    case 'teacher':
        console.log(name + ' is a teacher of coding');
        break;
```

```
case 'Driver':
    console.log(name + ' is a Uber Driver');
    break;
default:
    console.log(name + ' is something else');
}
```

>> John is a teacher of coding

So, how this work is let's explain using above example. If switch(job) is equals(case) to teacher then print(console.log).

And we use break to end this. Otherwise it will keep checking.

```
switch (true) {
    case age <=13 && age >= 20:
        console.log(name + ' you r smaller');
        break;
    case age == 20 && age != 20:
        console.log(name + ' you r 16');
        break;
    default:
        console.log(name + ' you r smaller');
}
```

>> You are smaller

Truthy And Falsy Values And Equality Operator

```
// falsy values: undefined, null, 0, '', NaN
// truthy values: NOT falsy values

var height;

[redacted] =
if (height) {
    console.log('[redacted] Defined');
}
else {
    console.log('Not Defined');
}
```

>>
Not Defined

So, if [redacted] we not defined a variable then it is falsy value.

```
// Equality Operator
height = 23;
if(height == '23') {
    console.log('The == operator does type coercion!');
}
```

Javascript converts the integer to string & string to int. in equality operator.
That's called Coercion. And not Javascript does.

So, in Javascript

"==" & "===" they both does the same thing!!

FUNCTIONS :-

```
function calculateAge (birthYear) {
        return 2018 - birthYear;
}

var John = calculateAge (1990);
var Mike = calculateAge (2000);
var Marry = calculateAge (1980);

console.log (John, Mike, Marry);
```

>>

28 18 38

```
function yearsUntilRetirement (year, name) {
        var age = calculateAge (year);
        var retirement = 65 - age;
        console.log (name + ' retires in ' + retirement + ' years');
}

yearsUntilRetirement (1980, 'HM');
```

>>

HM retires in 27 years

Function Statements And Expression:-

```
var whatDoyouDo = function (job, name) {
    switch (job) {
        case 'teacher':
            return name + ' teaches code';
        case' Driver':
            return name + ' Drives Uber';
        case' designer':
            return name + ' is a designer';
        default:
            return name + ' Do nothing';
    }
}

Console.log (whatDo you Do('teacher', 'Akash'));
```

⇒ Return is like break, it will stop it next to loop.

And this is called // Function Expression.

ARRAYS

```
var names = ['John', 'Mark', 'Jane', 'Mary'];
var years = new Array(1990, 1992, 1994);
```

⇒ Another way to define arrays

```
console.log(names[2]);
console.log(names.length);
console.log(years);
```

»

```
(4) ["John", "Mark", "Jane", "Mary"]
4
3 [1990, 1992, 1994]
```

```
// Mutate Array Data
names[1] = 'Ben';
console.log(names);
```

»

```
(4) ["John", "Ben", "Jane", "Mary"]
```

```
names[names.length] = 'Mary';
console.log(names);
```

»

```
(5) ["John", "Mark", "Jane", "Mary", "Mary"]
5
```

// Different Data Types :-

```
Var john = [`John`, `smith`, 1990, false];

john.push('blue'); => .push add the element at the end.
john.unshift('Mr.'); => .unshift add the element at the begining.
console.log(john);
```

>>

["Mr.", "John", "smith", 1990, false, "blue"]

```
john.pop();
john.pop();
```

} => Remove the element from the end.

```
john.shift(); => Removed the element from begining.
console.log(john);
```

>>

["John", "Smith", 1990]

```
console.log (john.indexOf(1990)); => tell the position of value.
```

>>

2

```
var isDesigner = john.indexOf('Designer') === -1 ? `John is
not a Designer` : `John is a Designer`
console.log (isDesigner);
```

>> John is not a Designer

OBJECTS AND PROPERTIES:-

In Arrays, the order matters a lot, but in objects it does not!!

```
// object Literal
var john = {                    ⇒ In arrays we used "[ ]" square brackets.
    name: 'John',               ⇒ Name is the key & John is the value.
    lastName: 'Smith',
    family:['akash', 'dev', 'abit'],  ⇒ we have defined an array in object.
    married: true
}

console.log(john);
```

>>
```
{ --- }
```
You will get all the Property of john.

Now to acess these property we have to use "." dot notation.

```
Console.log(john.name);
```

>>
John

another way to acess the properties is

```
console.log(john["lastName"]);
```

>>
Smith

OBJECTS AND METHODS:-

⇒ Attaching function to an objects are called Methods.

```
var john = {
    name: 'John';
    birthYear: 1990;
    calcAge: function(birthYear){
        return 2020 - birthYear;
    }
};
console.log( john.calcAge(1990));
```

» 30

But as you can see we have already defined birthYear then, to be easy.

```
calcAge: function(){
    return 2018 - this.birthYear;
}

console.log(john.calcAge());
```

⇒ Now what happens is it will check already given one.

» 28

// To stored a result in Johnobject:

```
calcAge: function() {
    this.age = 2018 - this.birthYear;
};
john.calcAge();
console.log(john);
```

>> This will store the value. //

" this" Keyword is very Powerfull.

LOOPS AND ITERATION :-

```
for (i=0; i<=5 ; i++) {
    console.log(i);
  3
```

>>

```
0
1
2
3
4
5
```

```
// i=0 , 0 <=5, true. log i to consol, i++
// this loop ------- will go until
//   i=0, 6 <=5 false, exit loop
```

```
var john = ['DJ', 'JS', 'PY', 'C+'];
for (var i=0; i < john.length; i++) {
      console.log (john[i]);
  };
```

>>

```
DJ
JS
PY
C+
```

While Loop :-

The difference b/w for loop & while loop is that while loop just only have condition.

```
var jone = ['a', 'b', 'c', 'd'];
var i = 0;
for (var i = 0; i < jone.length; i++){
    console.log (jone[i]);
```

Never Study in disturbance Hence Proved

```
while (i < jone.length) {
    console.log (jone[i]);
    i++;
};
```

>>

```
a
b
c
d
```

Continue & BREAK Statements :-

```
var a=["This", 'is', 2020, 'year', 'Hello!'];
for (var i = 0; i < a.length; i++) {
    if (typeof a[i] !== 'string') continue;
    console.log (a[i]);
}
```

⇒ Continue Statement

>>

```
This
is
year
Hello!
```

So, continue statement will not show that thing you want but show you the complete thing without that.

```
for (var i = 0; i<a.length; i++){
    if ( typeof a[i] !== 'string') breaks; => Break statement
     console.log(a[i]);
    };
```

```
>>
This
is
```

So, we will only loop until the string we want without no. as shown in above example.

// Looping Backwards

```
var  jack= ['a', 'b', 'c', 'd'];
for( var i= jack.length-1; i>=0; i-->){
     console.log(jack[i]);
    };
```

```
>>
d
c
b
a
```

Go ahead & Start Coding.

FROM BEGINING

S.E.