Javascript Notes

Part 1, 2, 3

Topics Covered

- Introduction to JavaScript
- Role of JavaScript
- Linking a JavaScript File
- Variables and Datatypes
- Template Literals
- Type Conversion and Coercion
- Truthy & Falsy Values
- Operators
- Statements & Expressions
- Functions

Topics Covered

- Arrays in JavaScript
- Accessing array elements
- Mutating array elements
- Basic array operations
- Introduction to objects
- Dot Vs Bracket Notation
- Object Methods

Topics Covered

- Iteration in Javascript
- Continue and Break
- Looping Backward
- DOM Manipulation
- Handling click events
- High level overview of Javascript
- Call Stack and Memory Heap
- Compilation and Interpretation

(*) What is Javascript?

Javascript is a high-level, object oriented, multi-paradigm programming language.

Programming language → A programming language is a tool that allows us to write code that will instruct a computer to do something.

High-level language → It means we don't have to worry about complex stuff like memory management.

Object-oriented → Based on objects for storing most kind of data.

multi-paradigm → It is so flexible and versatile that we can use different programming styles.

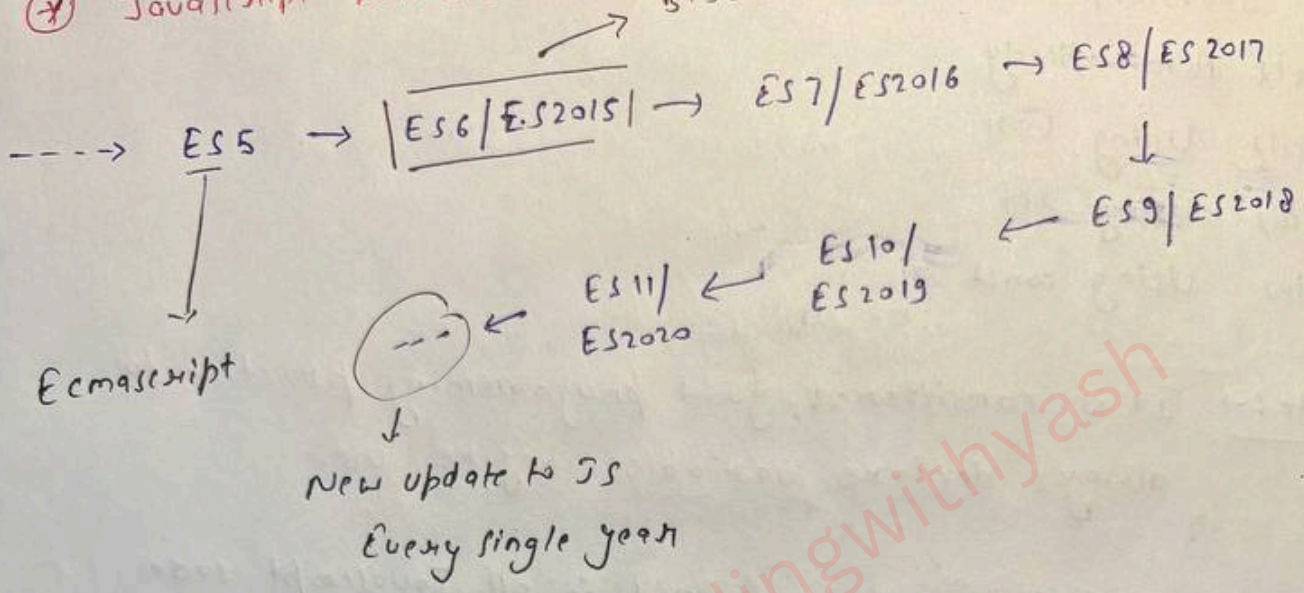
(*) The Role of Javascript in Web development?

(i) Javascript is the world's most popular programming language.

Analogy → HTML (Nouns) + CSS (Adjectives) + Javascript (Verbs)

(ii) Dynamic effects and web applications in the browser.

✳ Javascript Releases



✳ Linking a Javascript file.

We can either write the code in between <script> ..... </script> In HTML file. or else we can directly create a .js file. and then link it to the HTML file before the closing body tag.

ex→ `<script src="script.js"></script>`

Here in the above example we didn't write the actual javascript code, we just write the code in script.js file and then link it to the HTML file.

✳ Values and Variable →

A value is a piece of data, it's the most fundamental unit of information that we have in programming.

Variable are containers for storing data.

ex→ let firstName = "Yash"

firstName → variable name

"Yash" → value.

Javascript variable can be declared in 4 ways:

(i) Automatically
(ii) Using var
(iii) Using let
(iv) Using const

Note:→ It is considered good programming practice to always declare variables before use.

→ The var keyword was used in all Javascript code from 1995 to 2015.

→ The let and const keywords were added to Javascript in 2015.

→ The var keyword should only be used in code written for older browsers.

⊛ Rules for naming a variable.

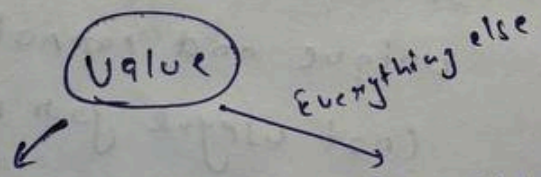
(i) Names can contain letter, digit, underscore & dollar sign.
(ii) Names must begin with a letter
(iii) Names cannot begin with a digit
(iv) Reserved words (ex. new) cannot be used as variable name.
(v) Names can begin with $ and _ symbols.
(vi) It should be meaningful.

Some examples of variable name.

ex→
```
let jonas_matilda = "JM";
let $function = 27;
let person = "Jonas";
let PI = 3.1415;
let myFirstJob = "Programmer";
let myCurrentJob = "Teacher";
```
] meaningful variable names

```
let job1 = "Programmer";
let job2 = "teacher";
```
] not making any meaning, so it is hard to understand their purpose.

⊛ Datatype in Javascript. → Every value in javascript is either an object or a primitive value.



Object

ex→
```
let me = {
    name: 'Jonas'
};
```

Primitive

ex→
```
let firstName = 'Jonas
let age = 30;
```

→ There are 7 primitive datatype

(i) Number → floating point number, used for decimals & integers.

ex→ let age = 25;

(ii) String → Sequence of characters, used for text

ex→ let firstName = 'Jonas';

always enclosed them in quotes no matter single quotes of double quotes.

(iii) Boolean → logical type that can be true or false, used for taking decisions.

ex→ let fullAge = true;

(iv) Undefined → value taken by a variable that is not yet defined. (empty value).

ex→ let children;

(v) Null → Also means empty value.

(vi) Symbol → Introduced in ES2015, value that is unique and cannot be changed. (not useful for now).

(vii) BigInt → Introduced in ES2020, larger integers than the Number type can hold

Note:→ Javascript has a feature of dynamic typing. It means we donot have to manually define the datatype of the value stored in a variable Instead, data types are determined automatically.

In javascript it's the value that has the type not the variable. Variable simply stores the value that has the type.

We can easily find the type of any value using the typeof operator ex→ console.log(typeof 23);

(*) When to user var, let, or const?

(i) Always declare variables.

(ii) Always use const if the value should not be changed.

(iii) Always use const if the type should not be changed (arrays & objects)

(iv) Only use let if you can't use const.

(v) Only use var if you must support old browsers.

(*) Javascript operators.

Javascript operators are used to perform diffrent types of computations.

like→ mathematical operators, logical operators, assignment operators, and many more.

ex→

```
const now = 2037;
const ageJonas = now - 1991
const fName = "Yash";
const lName = "Verma";
console.log("fName + " " + lName); → Yash Verma
```

(*) Some more assignment operators.

→ let x = 10+5; — 15

x+ = 10; // x = x+10;

console.log(x); ⟶ 25

→ x *= 4 // x = x * 4 = 100

(*) Some comparison operator → >, <, >=, <=

console.log(ageJonas >= 18);

Note: Operator precedence means the order in which operators are executed.

Strings and Template literals.

const fName = "Yash";

const job = "Developer";

const birthYear = 1991;

const year = 2037

Now let's try to build a string like → I'm, Yash, a-year old developer.

1st method with using + sign.

const Yash = "I'm" + fName + ', a' + (year - birthyear)
+ 'year old' + job + '!'

Here we can clearly how painful this is to build string like this.

II We can do this much easily with the help of template literal.

```
const yashNew = `I'm ${fName}, a ${year - birthyear}
                                    year old ${job}!;
```

This is backticks

dollar symbol

Here in this example we can clearly see that we don't have to worry about spaces, quotes (single or double) and it is much easier to build strings like this.

→ When we use backtick (` `) then we don't have to worry about which quotation marks we need to use.

ex → `console.log (`Just a regular string...`);`

→ Another use of template literal is to create multiline string

without using backtics

```
console.log ('string with \n
            multiple \n
            lines');
```

with using backticks

```
console.log (`string
            multiple
            lines`);
```

(*) Javascript if, else and else if.

-) conditional statement are used to perform different actions based on different conditions.

```
const age = 15;

if (age >= 18)
    {
    console.log ('sarah can start driving license');
    }
    else
        const yearLeft = 18 - age;
        console.log ('sarah is too young. wait
                    another ${yearLeft} years :)');

    }
```

(*) Type conversion and coercion.

Type conversion is when we manually convert from one type to another. [Explicitly]

Type coercion is when javascript automatically convert types behind the scenes for us. [Implicitly]

(i) Type conversion → let understand this by an example.

for example →

```
const inputYear = "1991";
```
→ string type

```
console.log(inputYear + 18);
```

Now this plus symbol will concatenate it instead of adding because inputYear is a string type and just because of this the output will become 199118

so to fix this we need to convert it into a number first explicity and for this we will use built-in Number() function

```
const inputYear = Number("1991");
console.log(inputYear + 18);
```

Now the output will be 1991 + 18 = 2009

(*) Converting strings to Numbers

The global method Number() (capital N) convert a variable or a value into a number.

examples → Number("3.14") ⟶ 3.14

Number("") ⟶ 0

Number("Yash") ⟶ NaN (not a number)

(*) Converting numbers to string we can use String() (capital S.)

(ii) Type coercion → It happens whenever an operator is dealing with two values that have diggerent types.

In this javascript will convert one of the values to match the other value behind the scenes, so that the operation can be executed.

let's understand with an example

ex → console.log('I am' + 23 + 'years old'); → I am 23 years old.

Here the plus operator that we used, triggers a coercion and convert the number to string automatically.

ex → console.log('10' - '5' - 3);
String String number

Here in this example, the minus operator will trigger an opposite conversion and will convert the strings to number so the output will be → (2)

+ → converts the number to string (concatenate)
- → converts the string to number
* → converts the string to number
/ → converts the string to number.
> → converts the string to number.

Question

let n = '1' + 1; → it will make '11' → string

n = n - 1; → It will make '11' - 1 ⇒ 10

console.log(n); → output 10 Ans

Question '10' - '4' - '3' - 2 + '5'

3 - 2

1 + '5' ⇒ 15

Truthy and falsy values

falsy values are values that are not exactly same false but will become false when we try to convert them into a boolean.

In Javascript we have five falsy values.

→ 0, '', undefined, null, NaN [or including false]

Everything else are our so-called truthy values.

Means the values that will be converted to true are truthy values.

for example → Any number that is not zero

→ Any string that is not an empty string.

Try these out in console and see the result for better understanding

```
console.log(Boolean(0));          → output → false
console.log(Boolean(undefined));  → false
console.log(Boolean('Jonas'));    → True
console.log(Boolean({}));         → True
console.log(Boolean(''));         → false.
```

2.

```
const money = 0;        → falsy value.
if (money)
  {
    console.log("Don't spend it all");
  }
  else
  {
    console.log('You should get a job!');
  }
```

Output will be → You should get a job!

Note:→ when we have only one line of code in our if block or else block then we dont need to put curly braces { }.

✱ Equality operators == Vs ===

== will perform type coercion.

ex → '18' == 18 → It will give true.
string Number
biroz here string is getting converted to a number.

=== will not perform type coercion.

ex → '18' === 18 → It will give false, biroz here type of the
string number value is also getting compared, which is string and number in this case.

✱ So from the above two explanation we can now understand
== will check loosely while === will check tightly
↓ ↓
only the value + type of value.

ex →
```
const favourite = Number(prompt("what's your number"));
console.log(favourite);
console.log(typeof favourite);

if(favourite === 23)
    console.log("cool! 23 is an amazing number!");

else if(favourite === 7)
    console.log("7 is also a cool number");

else
    console.log("number is not 23 or 7");
```

#. != Vs !==

↓ check loosely

↓ check tightly

just as we saw in == and === case, but here its works is opposite. it checks for negative

example

```
if (favourite !== 23)
    console.log ('why not 23');
```

Try this of the previous page code in your console.

(*) Basic Boolean logic → AND, OR & NOT operators

And

B \ A	AND	True	FALSE
	T	T	f
	f	f	f

In case of AND, If both the values are true then the result will be true.

OR

B \ A	OR	T	f
	T	T	T
	f	T	f

In case of OR, If both the values are false, only then the result will be false, else the result will be true

NOT operator just invert the values.

T will become f & vice-versa.

⊛ logical operators in Javascript.

```
const hasDriversLicense = true;  // A
const hasGoodVision = false;  // B

console.log (hasDriversLicense && hasGoodVision);  → false
console.log (hasDriversLicense || hasGoodVision);  → True

console.log (!hasdriversLicense);  → false (because of not operator)

const shouldDrive = hasDriversLicense && hasGoodVision;

if (shouldDrive)
  { console.log ('Sarah is able to drive!');
  }
else
  { console.log ('someone else should drive...');
  }
```

output will be → someone else should drive...

```
const isTired = true;  // C
console.log (hasDriversLicense && hasGoodVision && isTired);
```

output will be false, because hasGoodVision is false.

```
console.log (hasDriversLicense || hasGoodVision || isTired);
```

output will be True, because for an OR operator if any of the condition is True, output will be True.

⊛ Switch Statement in Javascript.

It is an alternative way to select one of many code blocks for execution.

Basically, switch statement is a multibranch selection statement.

Syntax → switch (expression)

```
{
  case x:
      } line of codes      ← this is not a part of syntax
    break;
  case Y:
      line of codes
    break;
      n. number of cases can be there

  default:
      line of codes
}
```


How it works:

(i) The switch expression is evaluated once.

(ii) The value of the expression is compared with the values of each case.

(iii) If there is a match, the associated block of code is executed

(iv) If there is no match, the default code block is executed.

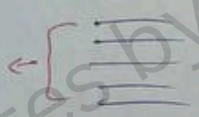
example →

```
const day = 'monday';
switch(day)
{
  case 'monday' : console.log('Today is monday');
                  break;

  case 'tuesday' : console.log('Today is tuesday');
                   break;

  case 'wednesday' : console.log('Today is wednesday');
                     break;
```

Here comes the same code for thursday & friday, so I skip it.



```
  case 'saturday' : console.log('Today is saturday')
                    break;

  default : console.log("Invalid choice");
}
```

Now few things here to pay attention.

(i) case monday will get executed so the output will be Today is monday.

(ii) It is mandatory to use break after every case

(iii) Default is optional and using break after default statement is also optional because the code is automatically going to be terminated.

(iv) If you don't use break, then you will encounter fall through condition.

(v) fall through condition means, one the case is matched it will get execute, so now logically the switch should end but what will happen due to the absence of break, it keep on executing until the break keyword is found or the closing curly braces '}' is found.

Now let's understand this with the previous example.


```
const day = 'monday';
switch (day)
{
    case 'monday': console.log("today is monday");
    case 'tuesday': console.log("Today is tuesday");
    case 'wednesday': console.log("today is wednesday");
    case 'thursday': console.log("Today is thursday");
    case 'friday': console.log("today is friday");
                   break;
    case 'saturday': console.log("today is saturday");
                     break;
    case 'sunday': console.log("today is sunday");
                   break;
    default: console.log("Invalid choice");
}
```

Explanation →

Now in this, day = monday so case monday will get executed but due to the absence of break keyword, Here the fall through condition will occur and the statement will get keeps on executing until a break is found or closing curly braces is found.

so, the output will be.....

Today is monday
Today is tuesday
Today is wednesday
Today is thursday
Today is friday

(#) Difference between statements & expressions.

Expressions are the line of code or simply a piece of code which produces a value.

ex → 3+4
ex → 1991
ex → true && false && !false ← this will also an expression b'coz it produces a boolean value.

Statements are like a bigger piece of code which does not produce a value on itself.

Example

```
if (23 > 10)
{
    const str = '23 is bigger';
}
```

So, this is an if-else statement and we can clearly see on execution it does not produce any value.
All it does is assigning '23 is bigger' to str.

Note: 23 is bigger this string is an expression.

"Statements are like full sentences and expressions are like the words/action that make up the sentences."

imp Note: In a template literal we can only insert expressions, but not statements.

* The Ternary operator (—?—:—);

Another way of writing if-else statements but all in one line.

ex-1

```
const age = 19;
age > 18 ? console.log("Eligible to vote") : console.log("not");
```

age > 18 → condition
? → question mark
console.log("Eligible to vote") → if the condition is true then this part will get executed
: → colon
console.log("not") → will execute if the condition is false.

Now the output will be → Eligible to vote.

example 2

```
const drink = age >=18 ? "Juice" : "water";
console.log(drink);
```

output will be → Juice.

If we have to use if-else statement for this, then the code will be like this.

```
let drink2;
if(age >=18)
  {
   drink2 = "Juice";
  }
  else
    { drink2 = "water";
    }
      console.log(drink2);
```

The benefit of using ternary operator is, we can use it inside a template literal, because ternary operator produces a value, but we cannot use if-else statement inside a template literal.

ex-)

```
console.log(`I like to drink ${age >=18 ? "Juice" : "water"}`);
```

⊛ Functions in Javascript.

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

A function will only get executed only if it get called.

A function can hold one or more lines of codes.

Syntax :

```
function functionName(parameters)
{
    // code to be executed;
}
```

example

```
function logger()        <- no parameters
{
  console.log('my name is Yash');
}

logger();   <- this is called function calling or invoking a function.
```

⊛ We can also pass data to a function and function can also return a value.

Examples

Parameters.

```
function fruitProcessor(apples, oranges)
{
    console.log(apples, oranges);
    const juice = `Juice with ${apples} apples and
                   ${oranges} oranges.`;
    return juice;
}
const appleJuice = fruitProcessor(5,0)
console.log(appleJuice)
```

Passing values to the Parameters.

output will be

5, 0 → because of line no. 2

Juice with 5 apples and 0 oranges.

(*) function Expression Vs function Declaration.

function declaration → A function declared as a separate statement, in the main code flow.

```
//function declaration
function sum(a, b)
{
    return a+b;
}
```

function Expression → A function, created inside an expression or inside another construct. Here, the function is created on the right side of the assignment operator (=).

```
//function expression.
let sum = function(a,b)
    {
      return a+b;
    }
```

Now let's understand with a given example.

This is function declaration.

```
function calcAge1(birthyear)
{
  return 2037 - birthyear;
}
const age1 = calcAge1(1991);
  console.log(age1);
```

For the same situation, let's now create a function expression

```
const calcAge2 = function(birthyear)
    {
      return 2037 - birthyear;
    }
const age2 = calcAge2(1991);
  console.log(age2)
```

Here in the above example we declared a function without a name (Anonymous function) and store that function in a variable, Now the variable will act as a function.

Note :- The big difference between function declaration and function expression is, we can actually call function declaration before they are defined in the code.

But on the other hand we cannot call function expression before defining them in the code.

⋆ Arrow function in Javascript.

parameter

ex → `const calcAge3 = birthyear => 2037 - birthyear;` (this is imp)

`const age3 = calcAge3(1991);`

`console.log(age3);`

ex → `const yearsUntilRetirement = birthyear => {`

`const age = 2037 - birthyear;`

`const retirement = 65 - age;`

`return retirement;`

`}`

`console.log(yearsUntilRetirement(1991));`

Note: we can only omit the return keyword only if we have only one line of code.

ex → When we have multiple parameters.

```
const yearsUntilRetirement = (birthyear, firstName) => {
    const age = 2037 - birthyear;
    const retirement = 65 - age;
    return `${firstName} retires in ${retirement} years`;
}

console.log(yearsUntilRetirement(1991, 'Yash'));
```

Output will be → Yash retires in 19 years.

(*) function calling other functions.

```
function cutFruitPieces(fruit)
{
    return fruit*4;
}

function fruitProcessor(apples, oranges)
{
    const applePieces = cutFruitPieces(apples);
    const orangePieces = cutFruitPieces(oranges);

    const juice = `Juice with ${applePieces} pieces apples and
        ${orangePieces} pieces oranges.`;

    return juice;
}

fruitProcessor(2, 3)
```

function calling another function.

(*) Arrays in Javascript

An array is a special variable, which can hold more than one value.

```
const cars = ["saab", "Volvo", "BMW"];
```

An array can hold many values under a single name, and you can access the values by referring to an index number.

Syntax

```
const array_name = [item1, item2, item3...];
```

It is a common practice to declare arrays with the const keyword.

We can also use the Javascript keyword new for creating arrays.

(ex-)

```
const cars = new Array("saab", "Volvo", "BMW");
```

⊛ Accessing Array Elements

We can access an array element by referring to the index number.

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0]
```

Array index always starts with zero.
[0] is the first element. [1] is the second element.

⊛ To get the actual number of element present in the array we can do this.

```
cars.length
```

array name

length will be counted from 1, 2, 3, ... so on.
But index number will be counted from 0, 1, 2, 3, ...

⊛ To get the last element of array.

```
cars[cars.length - 1]
```

length will be 3

It will become

cars[2] → It will give output → BMW

(*) Mutating / changing elements of an array

ex →
```
const friends = ['michael', 'steven', 'peter'];

friends[2] = 'Jay';

console.log(friends);
```
Output will be → (3) ["michael", "steven", "Jay"]

Note:

Only primitive values are immutable, but an array is not a primitive value, so we can change or mutate an array

But we cannot replace the entire array

ex →
```
const friends = ['Bob', 'Alice'] X It will give error.
```

(*) An array can hold value of diffrent types all at the same time.

ex →
```
const yash = ["Yash', 'verma', '2037-1999', `coder`
```

⊛ Basic Array Operations (methods).

ex → `const friends = ['michael', 'steven', 'peter'];`

`friends.push('Jay');`

(i) Push is a function which is used to add an element to the end of an array.

`console.log(friends);`

output will be → michael, steven, peter, Jay

Push function return the value of length of the new array and we can store the value also like this.

`const newLength = friends.push('Jay');`

`console.log(newLength);`

Output will be 4.

(ii) Unshift → It is used to add elements at the beginning of array, and this also returns a value of the length of new array.

`friends.unshift('John');`

`console.log(friends);`

Output will be → John, michael, steven, peter, Jay

(iii) Pop → This the opposite of Push method, means it will remove the last element of the array

```
friends.pop();
console.log(friends);
```

output will be → John, michael, steven, peter.

This method return the removed element and we can store it like this.

`friends.pop()` → will remove the last element

```
const popped = friends.Pop();
```

Here we can store that element.

```
console.log(popped);
console.log(friends);
```

↳ output → John, michael, steven, Peter

↳ output → Jay

(iv) shift → It is used to remove the first element of an array.

```
friends.shift();
console.log(friends)
```

output will be → michael, steven, Peter

(v) Index of → To find the index of an element

```
console.log(friends.indexOf('Steven'));
```

Output will be → 1

If we try to find the index of an element that doesn't exist →

the output will be → -1.

ex →

```
console.log(friends.indexOf('Bob');
```

Output ⇒ -1

(vi) Includes → Returns true, if the element is present in the array and return false if it is not present.

And this checks strictly

→ `friends.push(23);` → number

friends will become → Michael, Steven, Peter, 23

Now if we check

```
console.log(friends.includes('23'));
```
→ string

So, the output will be false.

Because it is testing with strict equality

(*) Introduction to objects in Javascript.

In object we define key-value pairs.

example →

```
const yash = {
    firstName : 'yash',
    lastName : 'Verma',
    age : 2024 - 1999,
    job : 'coder',
    friends : ['michael', 'Peter', 'steven'],
};
```

→ curly braces will be used

these are value.

These are keys ←

→ semicolon will come here

(*) Dot Vs Bracket Notation.

Dot → `console.log(yash.lastName);`

Bracket → `console.log(yash['lastName']);`

In bracket notation we need to specify a string with the property name and here we can use expression.

(ex→)
```
const namekey = 'Name';
console.log(yashv['last' + namekey]);
```

It will become

```
console.log(yashv['lastName']
```

(*) We can also use dot and bracket notation to add elements to an object.

```
yash.location = 'Delhi';
yash['instagram'] = '@codingwithyash';
```

(*) object methods in Javascript.

ex →

```
const yash = {
    firstName : 'Yash',
    lastName : 'Verma',
    birthYear : 1999,
    job : 'coder',
    friends : ['michael', 'Peter', 'steven'],
    hasDriversLicense : true,
    calcAge : function(birthYear)
        { return 2024 - birthYear; }
};
```

Here we have used function Expression, we cannot use function declaration here.

we can access this like below.

```
yash.calcAge(1999);
```

This keyword is basically equal to the object on which the method is called. or in simple words, it is equal to the object calling the method.

ex →

```
const yash = {
    firstName : 'Yash',
    lastName : 'Verma',
    birthYear : 1999,
    job : 'coder',
    friends : ['michael', 'Peter', 'steven'],
    hasDriversLicense : true,
    calcAge : function()
        { return 2024 - this.birthYear;
        }
};
```

← comma

Here we have used this keyword and because of this we are not having any parameters inside the ()

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

not passing any value.

* Instead of returning we can also create a new key value pair using this keyword.

ex →

```
calcAge : function()
    {
        this.age = 2024 - this.birthYear
    }
```

this will be the new key.

⊛ Iteration in Javascript.

Loops can execute a block of code a number of times.

(i) **for loop.** → for loop keeps running while condition is true.

Syntax → Initilization condition updation

```
for(let rep= 1 ; rep<=10 ; rep++)
  {
  console.log(`Lifting weights repetition
                ${rep}`);

  }
```

example

```
const yash = ['Yash', 'Verma', 2024-1991,
              'Coder',
            ['michael', 'Peter', 'Steven'],
              true ];
```

Empty array ← `const types = [];`

```
for(let i=0; i< yash.length ; i++)
  {
```

Reading array ← `console.log(yash[i], typeof yash[i]);`

filling array ← `types[i] = typeof yash[i];`

```
  }
```

We can also use `types.push(typeof yash[i]);`

(*) Use of continue and break statements.

continue is to exit the current iteration of the loop and continue (move on) to the next one.

Break is use to completely terminate the whole loop

(i) example as per the previous page array.

```
for (let i=0 ; i<yash.length ; i++)
{
  if(typeof yash[i] !=='string') continue;
        console.log (yash[i], typeof yash[i]);
}
```

this line will skill all those value, that are not string

(ii) Example for break

```
for(let i=0; i<yash.length ; i++)
{
  if(typeof yash[i] === 'number') break;
        console.log(yash[i], typeof yash[i]);
}
```

This line will terminate the whole loop as soon as the first number is found.

⊛ Looping backwards of Nested loops in javascript.


```
for( let i = yash.length -1 ; i>=0 ; i--)
    { console.log(yash[i]);
    }
```

Nested loop.

example → We need to create a loop that perform 3 set of 6 repetitions.

```
for( let exercise =1; exercise<4; exercise++)
    {
        console.log(`starting exercise ${exercise}`);
        for( let rep= 1; rep<=6 ; rep++)
            {
                console.log('lifting rep number ${rep}');
            }
    }
```

⊛ While loop in Javascript

The while loop also loops through a block of code as long as a specified condition is true.

Syntax →

```
while (condition)
  {
    // code block to be executed.
  }
```

Ex →

```
let i = 0
while (i < 10)
    { console.log(i);
      i++;
    }
```

⊛ Dom Manipulation in Javascript

With the HTML DOM, Javascript can access and change all the elements of an HTML document.

When a page is loaded, the browser creates a Document Object Model of the page.

⊛ With the object model, javascript gets all the power it needs to create dynamic HTML.

(i) Javascript can change all the HTML elements in the page.

(ii) Javascript can change all the HTML attributes in the page

(iii) Javascript can change all the css styles in the page

(iv) Javascript can remove existing HTML elements and attributes

(v) Javascript can add new HTML elements & attributes

(vi) Javascript can react to all existing HTML events in the page.

(vii) Javascript can create new HTML events in the page.

⊛ Document Object Model → It is a structured representation of HTML Documents. Allows javascript to access HTML element and styles to manipulate them.

⊛ selecting and manipulating elements.

• querySelector('.')

class name
or
Id name

for example. → document.querySelector('.message')

Note:- for other HTML element we use .textContent property to get the actual value, but for the input field we always use .value property

⊛ Handling Click Events

An event is something that happens on the page for example, a mouse click, a mouse moving or a key press

```
document.querySelector('.check').addEventListener('click',
function(){
console.log('Hello');
});
```

Note:- We donot call this function anyw, we only define the function here, it will automatically get called as soon as this event happens.

Note: Whenever we get something from the user Interface, for example an input field it usually always a string.

(*) The text content property sets or return the text content of the specified node, and all its descendants.

Note:→ Unlike CSS, when setting the styles using the style property in Javascript. We have to write the css properties in camelCase notation.

Instead of using a dash - , to separate the words you will have to write in capitals the first letter of each word.

example

backgroundColor ✓ background-color ✗

fontSize ✓ font-size ✗

① An High level Overview of Javascript.

Javascript is a high-level, object oriented, multi-paradigm programming language.

more detailed definition is →

Javascript is a high-level, prototype-based object-oriented, multi-paradigm, interpreted or just-in-time compiled, Dynamic, single threaded, Garbage-collected programming language with first-class functions and a non-blocking event loop concurrency model.

Paradigm means an approach and mindset of structuring code, which will direct your coding style and technique.

Three popular paradigm are → ① Procedural programming
② object-oriented programming
③ functional Programming

② Javascript Enginge and Runtime.

Javascript Engine is simply a computer program that executes Javascript code.

Every browser has its own javascript Engine but probably the most well known engine is Google's V-8 engine

Any Javascript Engine always contains the callstack and Heap

Call stack → The call stack is where our code is actually executed, using something called execution context.

Heap → Heap is an unstructured memory pool which stores all the objects that our application needs.

The javascript Engine does a lot of work for us. But the biggest thing is reading our code and executing it. The two main important things in this step are:-

(i) We need a place to store and write information - data for our app (variables, objects, etc...)

(ii) We need to keep track of what's happening to our code line by line.

This is where a call stack and a memory heap comes in. We need the memory heap as a place to store and write information because at the end of the day all programs just read and write operations - that is to allocate, use and release memory.

⊛ Diggerence between compilation and Interpretation.

Compilation → Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.

Source code —(Step 1: compilation)→ Portable file: machine code —(Step 2: Execution)→ Program Running.

Interpretation → Interpreter runs through the source code and executes it line by line.

Source code —(Step 1: Execution line by line)→ Program Running

⊛ How is javascript code executed ?

→ Execution context is an environment in which a piece of Javascript is executed, stores all the necessary ingormation for some code to be executed.

→ Exactly one global execution context (EC) ⇒ Degault context, created for code that is not inside any junction (top-level).

→ One execution context per junction → for each junction call, a new execution context is created containing all the ingormation that is necessary to run exactly that junction. and same goes for methods as well.

compilation ⟶ creation of global execution context for (top-level code) which are not inside of any function ⟶ Execution of top level code (Inside Global EC)

↓

Execution of functions and waiting for callbacks.

(*) What's Inside Execution context?

(1) Variable Environment

(*) let, const and var declarations

(*) function

(*) Argument object.

(2) Scope chain

(3) this keyword.

Note:- Arrow functions do not get their own argument keywords nor do they get this keyword.

Instead they can use the argument object and this keyword from their closet regular function parent.

Example

```
const name = 'Jones'

const first = () => {
    let a = 1;
    const b = second(7,9);
    a = a+b
    return a;
};

function second(x,y) {
    var c = 2;
    return c;
}
const x = first();
```

Global

name = 'Jones'

first = <function>

second = <function>

x = <unknown>

need to run first() first.

function code

first()

a = 1

b = <unknown>

need to run second() first

second()

c = 2

arguments = [7,9]

Array of passed arguments

available in all "irregular" functions (not arrows).

Follow for more

Thank You !!!