# JavaScript

**How can use JavaScript in code?**

1. JavaScript code use between **<script> ….. </script>** tag.

<script>

document.getElementById ("main").innerHTML = "JavaScript";

</script>

**\*\* Old JavaScript examples may use a type attribute: <script type="text/javascript">.**

**\*\* The type attribute is not required now. JavaScript is the default scripting language in HTML.**

1. You can place any number of scripts in an HTML document.
2. Scripts can be placed in the **<body>** or in the **<head>** section of an HTML page, or in both.
3. Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display.
4. For using external script file, file name uses in **src** (source) attribute of a **<script>** tag:

**<script src="myScript.js"> </scrip>**

**\*\* External script files cannot contain <script> tags.**

**JavaScript Output:**

1. Writing into an HTML element, using innerHTML.
2. Writing into the HTML output using document.write().

\*\* Using document.write() after an HTML document is loaded, will delete all existing HTML:

1. Writing into an alert box, using window.alert().
2. Writing into the browser console, using console.log().

**JavaScript Statements:**

1. JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

1. Semicolons separate JavaScript statements.
2. JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.

# JavaScript Syntax

JavaScript syntax is the set of rules, how JavaScript programs are constructed:

1. **JavaScript Values Syntex:** The JavaScript syntax defines two types of values:
2. **Fixed values (Literal**)

\*\*Numbers are written with or without decimals: ---------- 10, 10.58

\*\*Strings are text, written within double or single quotes. ---------- “shuvo”

1. **Variable values (Variables)**

\*\*variables are used to store data values.

\*\*JavaScript uses the keywords var, let and const to declare variables.

\*\*An equal sign is used to assign values to variables

\*\* All JavaScript identifiers are case sensitive.

\*\* Hyphens are not allowed in JavaScript. They are reserved for subtractions.

1. **JavaScript Operators:** JavaScript uses arithmetic operators ( + - \* / ) to compute values:
2. **JavaScript Expressions:** An expression is a combination of values, variables, and operators, which computes to a value.
3. **JavaScript Keywords:** JavaScript keywords are used to identify actions to be performed
4. **JavaScript Comments:** Code after double slashes // or between /\* and \*/ is treated as a comment.
5. **JavaScript Character Set**: JavaScript uses the Unicode character set.

# JavaScript Variables

1. Variables are containers for storing data (storing data values).
2. All JavaScript variables must be identified with unique names.
3. These unique names are called identifiers.
4. JavaScript identifiers are case-sensitive.
5. It's a good programming practice to declare all variables at the beginning of a script.
6. var
7. let
8. const
9. nothing

### Let:

1. If you think the value of the variable can change, use let.
2. Variables defined with let cannot be Redeclared in same block.
3. Variables defined with let must be Declared before use.
4. Variables defined with let have Block Scope.

### Const:

1. If you want a general rule: always declare variables with const
2. Variables defined with const cannot be Redeclared.
3. Variables defined with const cannot be Reassigned.
4. Variables defined with const have Block Scope.
5. JavaScript const variables must be assigned a value when they are declared:

Use const when you declare:

1. A new Array
2. A new Object
3. A new Function
4. A new RegExp

The keyword const is a little misleading. It does not define a constant value. It defines a constant reference to a value. its means identifier

You can NOT:

1. Reassign a constant value
2. Reassign a constant array
3. Reassign a constant object

You CAN:

1. Change the elements of constant array
2. Change the properties of constant object

### Value = undefined:

variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input. A variable declared without a value will have the value undefined.

# JavaScript Operators

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Conditional Operators
6. Type Operators

# Data Types

1. JavaScript variables can hold different data types: numbers, strings, objects.
2. JavaScript evaluates expressions from left to right. Different sequences can produce different results:
3. JavaScript has dynamic types. This means that the same variable can be used to hold different data types:
4. Number
5. String
6. Boolean
7. Object
8. Undefined

\*\* JavaScript has dynamic types. This means that the same variable can be used to hold different data

# JavaScript Events

1. HTML events are "things" that happen to HTML elements.
2. When JavaScript is used in HTML pages, JavaScript can "react" on these events.
3. An HTML event can be something the browser does, or something a user does.

**Here are some examples of HTML events:**

1. An HTML web page has finished loading
2. An HTML input field was changed
3. An HTML button was clicked

**Common HTML Events**

1. **Onchange**: An HTML element has been changed
2. **Onclick**: The user clicks an HTML element
3. **Onmouseover**: The user moves the mouse over an HTML element
4. **Onmouseout**: The user moves the mouse away from an HTML element
5. **Onkeydown**: The user pushes a keyboard key
6. **Onload**: The browser has finished loading the page

**JavaScript Event Handlers**

1. Event handlers can be used to handle and verify user input, user actions, and browser actions:
2. Things that should be done every time a page loads
3. Things that should be done when the page is closed
4. Action that should be performed when a user clicks a button
5. Content that should be verified when a user inputs data

**Many different methods can be used to let JavaScript work with events:**

1. HTML event attributes can execute JavaScript code directly
2. HTML event attributes can call JavaScript functions
3. You can assign your own event handler functions to HTML elements
4. You can prevent events from being sent or being handled

# Strings

1. You can use single or double quotes
2. You can use quotes inside a string, if they don't match the quotes surrounding the string
3. Comparing two JavaScript objects always returns false.

### String Search

1. String indexOf()
2. String lastIndexOf()
3. String search ()
4. String match ()
5. String matchAll()
6. String includes ()
7. String startsWith()
8. String endsWith()

### Escape Character

1. \’ ------ ' ------ Single quote
2. \" " Double quote
3. \\ \ Backslash
4. \b Backspace
5. \f Form Feed
6. \n New Line
7. \r Carriage Return
8. \t Horizontal Tabulator
9. \v Vertical Tabulator

\*\* Do not create Strings objects.

### String Methods

* All string methods return a new string. They don't modify the original string.
* Strings are immutable: Strings cannot be changed, only replaced.

1. **Length**: The length property returns the length of a string:

let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

let length = text.length;

1. **slice (start, end):**

* extracts a part of a string and returns the extracted part in a new string.
* The method takes 2 parameters: start position, and end position (end not included).
* JavaScript counts positions from zero. First position is 0. Second position is 1.
* If a parameter is negative, the position is counted from the end of the string.
* slices out a portion of a string from position -12 to position -6.

let text = "Apple, Banana, Kiwi";

let part = text.slice(7, 13);

Ans: Banana

1. **substring (start, end):**

* substring () is like slice ().
* The difference is that start and end values less than 0 are treated as 0 in substring ()
* If you omit the second parameter, substring () will slice out the rest of the string.

1. **substr (start, length):**

* second parameter specifies the length of the extracted part.
* If you omit the second parameter, substr () will slice out the rest of the string.
* If the first parameter is negative, the position counts from the end of the string.

1. **replace ():**

* The replace () method replaces a specified value with another value in a string.
* The replace () method does not change the string it is called on.
* The replace () method returns a new string.
* The replace () method replaces only the first match.
* the replace () method is case sensitive

1. **toUpperCase** (): A string is converted to upper case with toUpperCase ().
2. **toLowerCase** (): A string is converted to lower case with toLowerCase ()
3. **concat**():

* joins two or more strings.
* can be used instead of the plus operator. These two lines do the same.

1. **trim** (): The trim() method removes whitespace from both sides of a string:
2. **trimStart**(): removes whitespace only from the start of a string.
3. **trimEnd**() : removes whitespace only from the end of a string.
4. **padEnd**(): pads a string with another string:
5. **charAt**()
6. **split()**

\*\*All string methods return a new string. They don't modify the original string.

### Template Literals

1. **Back-Tics Syntax**

Template Literals use back-ticks (``) rather than the quotes ("") to define a string:

let text = `He's often called "Johnny"`;

let text =`The quick

brown fox

jumps over

the lazy dog`;

1. **Interpolation**

Template literals provide an easy way to interpolate variables and expressions into strings.

The syntax is:  **${...}**

let firstName = "John";

let lastName = "Doe";

let text = `Welcome ${firstName}, ${lastName}!` ;

1. **tagged template literals ----------------- (ES6)**
2. Tags allow you to parse template literals with a function.
3. The first argument of a tag function contains an array of string values.
4. The remaining arguments are related to the expressions.

**Syntax:**

TYPICAL FUNCTION

function greet (string, …values) {

// do something

};

TAG FUNCTION

greet `I'm ${name}. I'm ${age} years old. `

**Example:**

function useless(strings, ...values) {

return 'I render everything useless.';

}

let name = 'Benedict';

let occupation = 'being awesome';

let sentence = useless `Hi! I'm ${ name } and I'm busy at ${ occupation }.` ;

console.log(sentence);

# JavaScript Numbers

1. JavaScript Numbers are Always 64-bit Floating Point
2. Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

**Number Methods**

1. **toString() :** Returns a number as a string

let x = 123;  
x.toString();  
(123).toString();  
(100 + 23).toString()

1. **toExponetial():** Returns a number written in exponential notation

let x = 9.656;  
x.toExponential(2);  
x.toExponential(4);  
x.toExponential(6);

1. **toFixed():** Returns a number written with a number of decimals

let x = 9.656;  
x.toFixed(0);  
x.toFixed(2);  
x.toFixed(4);  
x.toFixed(6);

1. **toPrecision():** Returns a number written with a specified length

let x = 9.656;  
x.toPrecision();  
x.toPrecision(2);  
x.toPrecision(4);  
x.toPrecision(6);

1. **ValueOf():** Returns a number as a number

let x = 123;  
x.valueOf();  
(123).valueOf();  
(100 + 23).valueOf();

1. **Number ():** 
   1. Returns a number converted from its argument.
   2. If the number cannot be converted, NaN (Not a Number) is returned.
   3. Number () can also convert a date to a number.

Number(true);

Number(false);

Number("10");

Number("  10");

Number("10  ");

Number(new Date("1970-01-01"))

1. **parseFloat():** Parses its argument and returns a floating point number

parseFloat("10");  
parseFloat("10.33");  
parseFloat("10 20 30");

1. **parseInt():** Parses its argument and returns a whole number

parseInt("-10");  
parseInt("-10.33");  
parseInt("10");

# Date Objects

1. new Date() creates a new date object with the current date and time:
2. 6 numbers specify year, month, day, hour, minute, second:
3. 5 numbers specify year, month, day, hour, and minute:
4. 3 numbers specify year, month, and day:

# Math Object

1. **Math.E** - returns Euler's number.
2. **Math.PI** - returns PI.
3. **Math.SQRT2** - returns the square root of 2.
4. **Math.SQRT1\_2 -** returns the square root of ½.
5. **Math.LN2**  - returns the natural logarithm of 2.
6. **Math.LN10** - returns the natural logarithm of 10.
7. **Math.LOG2E** - returns base 2 logarithm of E.
8. **Math.LOG10E**
9. **Math.round(x)** - Returns x rounded to its nearest integer.
10. **Math.ceil(x)** - Returns x rounded up to its nearest integer.
11. **Math.floor(x)** - Returns x rounded down to its nearest integer.
12. **Math.trunc(x)-** Returns the integer part of x (new in ES6).
13. **Math.random()**

# Conditional statements

1. **If statement:**
   * Use if to specify a block of code to be executed, if a specified condition is true
   * Use else to specify a block of code to be executed, if the same condition is false
   * Use else if to specify a new condition to test, if the first condition is false
   * Use switch to specify many alternative blocks of code to be executed
2. **for:**
3. **for( x in y ) loop:**

const numbers = [45, 4, 9, 16, 25];  
let txt = "";  
for (let x in numbers) {  
  txt += numbers[x];  
}

\*\*\* Do not use for in over an Array if the index order is important.

1. **forEach():**

The forEach() method calls a function (a callback function) once for each array element.

const numbers = [45, 4, 9, 16, 25];  
let txt = "";  
numbers.forEach(myFunction);  
function myFunction(value, index, array) {  
  txt += value;  
}

Note that the function takes 3 arguments:

1. The item value
2. The item index
3. The array itself
4. **for/of:**

JavaScript for of statement loops through the values of an iterable object.

const cars = ["BMW", "Volvo", "Mini"];  
let text = "";  
for (let x of cars) {  
  text += x;  
}

1. **while:**
2. **do/while:**
3. **Switch statement:**

The continue statement (with or without a label reference) can only be used to skip one loop iteration. the break statement "jumps out" of a loop.

switch(expression) {

case x:

// code block

break;

case y:

// code block

break;

default:

// code block

}

# Array

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

1. JavaScript, arrays use numbered indexes.
2. Last array index is [ array.length - 1];
3. Arrays are a special type of objects.
4. Array indexes start with 0.

### Create array:

Method-1:

const array\_name = [item1, item2, ...., ….];

Method-2:

const cars = [];

cars[0]= "Saab";

cars[1]= "Volvo";

Method-3:

Const car = new Array(“shuvo”,26); // no need to use new array method

**Accessing Array Elements:**

const cars = ["Saab", "Volvo", "BMW"];

let car = cars [0]; // first element

console.log(cars) // full array

let car = cars[cars.length - 1]; // last index

### Array method & property:

* **Length:** The length property of an array returns the length of an array.

const p = cars.length;

* **Array.isArray():** to know the variable is array or object use Array.isArray() methods.

Array.isArray(cars);

* **Instanceof ():** The instanceof operator returns true if an object is created by a given constructor

cars instanceof Array;

* **toString() :** method  converts an array to a string of (comma separated) array values.

const fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log(fruits.toString());

* **join()** : method also joins all array elements into a string. But we can specify the separator;

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.join(" \* ");

* **pop() :** removes the last element from an array.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();

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

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");

* **shift ():** removes the first array element and "shifts" all other elements to a lower index.
  + 1. const fruits = ["Banana", "Orange", "Apple", "Mango"];  
       fruits.shift();
* **unshift ():** adds a new element to an array at the beginning , and "unshifts" older elements:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");

* **delete():** 
  1. Array elements can be deleted using the JavaScript operator delete.
  2. Using delete leaves undefined holes in the array.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
delete fruits[0];

* **includes () :** 
  1. The includes() method returns true if an array contains a specified value.
  2. The includes() method returns false if the value is not found.
  3. The includes() method is case sensitive.

Syntax:

array.includes (element, start);

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.includes("Banana", 3);

* **concat():** creates a new array by merging (concatenating) existing arrays. The concat() method can also take strings as arguments.

const G = ["Cecilie", "Lone"];  
const B = ["Emil", "Tobias", "Linus"];  
const C = G.concat(B);

const D = G.concat(B, C); // three array concat

* **splice (added, removed, elements1, elements2):**
  1. adds new items to an array.
  2. Splice() method work in original array.
  3. The 1st defines the position where new elements should be added (spliced in).
  4. The 2nd parameter defines how many elements should be removed.
  5. The rest of the parameters ("Lemon”, "Kiwi") define the new elements to be added.

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

* **slice(removed):**
  1. The slice() method slices out a piece of an array into a new array.
  2. The slice() method creates a new array.
  3. The slice() method does not remove any elements from the source array.
  4. The method then selects elements from the start argument, and up to (but not including) the end argument.
* s**ort ():**
  1. **String Sort:** methos sort the array alphabetically.
     + 1. const fruits = ["Banana", "Orange", "Apple", "Mango"];  
          fruits.sort();
  2. **Nemuric Sort:** By default, the sort() function sorts values as strings. Because of this, the sort() method will produce incorrect result when sorting numbers.
     + 1. const points = [40, 100, 1, 5, 25, 10];  
          points.sort(function(a, b){return a - b});
* **Random Order:**
  + 1. const points = [40, 100, 1, 5, 25, 10];  
       points.sort(function(){return 0.5 - Math.random()});
* **reverse() :** reverses the elements in an array.
* **Math.max()**
* **Math.min()**

# Iteration

1. **ForEach():** method calls a function (a callback function) once for each array element.

**const numbers = [45, 4, 9, 16, 25];  
 let txt = "";  
 numbers.forEach(myFunction);  
 function myFunction(value, index, array) {  
  txt += value + "<br>";  
 }**

myFunction(value, index, array) takes 3 arguments:

1. The item value
2. The item index
3. The array itself
4. **map():**
5. The map() method creates a new array by performing a function on each array element.
6. The map() method does not execute the function for array elements without values.
7. The map() method does not change the original array.

**const numbers1 = [45, 4, 9, 16, 25];**

**const numbers2 = numbers1.map(myFunction);**

**function myFunction(value, index, array) {**

**return value \* 2;**

**}**

1. **filter():**

The arr.filter() method is used to create a new array from a given array consisting of only those elements from the given array which satisfy a condition set by the argument method.

const numbers = [45, 4, 9, 16, 25];  
const over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {  
  return value > 18;  
}

**syntax:**

array.filter( callback(element, index, arr ) , thisValue )

1. callback: This parameter holds the function to be called for each element of the array.
2. element: The parameter holds the value of the elements being processed currently.
3. index:  is optional, it holds the index of the current Value element in the array starting from 0.
4. arr: This parameter is optional; it holds the complete array on which Array.
5. thisValue: This parameter is optional, it holds the context to be passed as this to be used while executing the callback function. If the context is passed, it will be used like this for each invocation of the callback function, otherwise undefined is used as default.

the callback function does not use the index and array parameters, so they can be omitted:

**const numbers = [45, 4, 9, 16, 25];  
const over18 = numbers.filter(myFunction);  
function myFunction(value) {  
   return value > 18;  
}**

1. **reduce()**
2. The reduce() method runs a function on each array element to produce (reduce it to) a single value.
3. The reduce() method works from left-to-right in the array.
4. The reduce() method does not reduce the original array.

**const numbers = [45, 4, 9, 16, 25];  
let sum = numbers.reduce(myFunction, 100);  
function myFunction(total, value) {  
   return total + value;  
}**

**Syntax:**

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

1. function(total, currentValue, index, arr): It is the required parameter and used to run for each element of array. It contains four parameter which are listed below:
2. total: It is required parameter and used to specify the initialValue, or the previously returned value of the function.
3. currentValue: It is required parameter and used to specify the value of the current element.
4. currentIndex: It is optional parameter and used to specify the array index of the current element.
5. arr: It is optional parameter and used to specify the array object the current element belongs to.
6. initialValue: It is optional parameter and used to specify the value to be passed to the function as the initial value.
7. **reduceRight()**

The reduceRight() works from right-to-left in the array.

const numbers = [45, 4, 9, 16, 25];  
let sum = numbers.reduceRight(myFunction);  
  
function myFunction(total, value, index, array) {  
   return total + value;  
}

1. **every()**
2. **some()**
3. **indexOf()**
4. **lastIndexOf()**
5. **find()**
6. **findIndex()**
7. **Array.from()**
8. **Keys()**
9. **entries()**
10. **includes()**

# Destructuring ---- ES6

when we pass those to a function, it may need not be an object/array as a whole. It may need individual pieces. Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient.

1. **Destructing Arrays:**

const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays

const [x, y, z] = arrValue;

console.log(x); // one

console.log(y); // two

console.log(z); // three

1. **Object destructuring**

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { name, realName } = hero;

console.log(name);

console.log(realName);

# Spread Operator --------ES6

The JavaScript spread operator **( ... )** allows us to quickly copy all or part of an existing array or object into another array or object.

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

**Assign the first and second items from numbers to variables and put the rest in an array.**

Const numbers = [1, 2, 3, 4, 5, 6];

const [one, two, ...rest] = numbers;

**We can use the spread operator with objects too:**

const myVehicle = {

brand: 'Ford',

model: 'Mustang',

color: 'red'

}

const updateMyVehicle = {

type: 'car',

year: 2021,

color: 'yellow'

}

const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle }

# Rest parameter ----- ES6

1. ES6 provides a new kind of parameter so-called rest parameter that has a prefix of three dots (...)
2. The rest parameter syntax allows a function to accept an indefinite number of arguments as an array
3. The rest parameters must be at the end

**Syntax:**

function fn (a, b, ...args) {

//......................................

}

**Example:**

function myFun(a, b, ...manyMoreArgs) {

console.log("a", a);

console.log("b", b);

console.log("manyMoreArgs", manyMoreArgs);

}

myFun ("one", "two", "three", "four", "five", "six");

# Maps

1. A Map holds key-value pairs where the keys can be any datatype.
2. A Map remembers the original insertion order of the keys.
3. Creating a Map by passing an Array to the new Map() constructor:

const fruits = new Map ([

  ["apples", 500],

  ["bananas", 300],

  ["oranges", 200]

]);

Method of Maps :

1. **set() method:** Adding elements to a Map with the set () method:

**const fruits = new Map ();**

**fruits.set("apples", 500);**

**fruits.set("bananas", 300);**

**fruits.set("oranges", 200);**

1. **get() Method:** this gets the value of a key in a Map:

**fruits.get("apples");**

1. **size Property**: **fruits.size;**
2. **delete () Method: fruits.delete("apples");**
3. **has() Method : fruits.has("apples");**
4. **forEach() Method:** This method calls a function for each key/value pair in a Map:

**let text = "";**

**fruits.forEach (function(value, key) {**

**text += key + ' = ' + value;**

**})**

1. **entries() Method:**  this method returns an iterator object with the [key, values] in a Map:

**let text = "";**

**for (const x of fruits.entries() ) {**

**text += x;**

**}**

# Modules ------ES6

1. JavaScript modules rely on the import and export statements.
2. JavaScript modules allow you to break up your code into separate files.
3. **Export:**

You can export a function or variable from any file. You can create named exports two ways

1. In-line individually:

export const name = "Jesse";  
export const age = 40;

1. All at once at the bottom:

const name = "Jesse"

const age = 40

export { name, age }

1. Default Exports

const message = () => {

const name = "Jesse";

const age = 40;

return name + ' is ' + age + 'years old.';

};

export default message;

1. **Import:**

You can import modules into a file in two ways, based on if they are named exports or default exports.

1. Import from named exports

import { name, age } from "./person.js";

1. Import from default exports

import message from "./message.js";

# Errors

1. **try and catch:**
2. The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
3. The**catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

try {

Block of code to try  
}  
catch(err) {

  Block of code to handle errors  
}

1. **throw Statement:**

When an error occurs, JavaScript will normally stop and generate an error message. The throw

Statement allows you to create a custom error. The technical term for this is: JavaScript will throw an exception (throw an error).

**try {**

**If (x == "") {**

**throw "empty";**

**}**

**If (isNaN(x)) throw "not a number";**

**}**

**catch(err) {**

**message.innerHTML = "Input is " + err;**

**}**

1. **finally, Statement:**

The finally statement defines a code block to run regardless of the result.

**try {**

**Block of code to try**

**}**

**catch(err) {**

**Block of code to handle errors**

**}**

**finally {**

**Block of code to be executed regardless of the try / catch result**

**}**

# Scope

Scope determines the accessibility of variables, objects, and functions from different parts of the code

**Block scope:**

1. Variables declared inside a { } block cannot be accessed from outside the block:
2. Variables declared with the var keyword can NOT have block scope.

**{**

**let x = 2;**

**}  
// x can NOT be used here**

**Function scope:**

1. Variables declared within a JavaScript function, become LOCAL to the function.
2. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.
3. Local variables are created when a function starts, and deleted when the function is completed.

**function myFunction() {**

**let carName = "Volvo";**

**}**

**Global scope:**

1. A variable declared outside a function, becomes GLOBAL.
2. A global variable has Global Scope:
3. All scripts and functions on a web page can access it.
4. JavaScript, the global scope is the JavaScript environment.
5. In HTML, the global scope is the window object.
6. Global variables defined with the var keyword belong to the window object.
7. Global variables defined with the let keyword do not belong to the window object:

**let carName = "Volvo";  
function myFunction() {**

**// code here can also use carName**

**}**

\*\*\*\*\*\*

1. In "Strict Mode", undeclared variables are not automatically global.
2. Do NOT create global variables unless you intend to.

**The Lifetime of JavaScript Variables**

1. The lifetime of a JavaScript variable starts when it is declared.
2. Function (local) variables are deleted when the function is completed.
3. In a web browser, global variables are deleted when you close the browser window (or tab)

# Hoisting

1. Hoisting is JavaScript's default behavior of moving declarations to the top.
2. a variable can be used before it has been declared.

x = 5; // Assign 5 to x

var x; // Declare x

**LET:**

1. Variables defined with let is hoisted to the top of the block, but not initialized as undefined .
2. Using a let variable before it is declared will result in a ReferenceError.
3. The variable is in a "temporal dead zone" from the start of the block until it is declared:

**Const :**

1. Using a const variable before it is declared, is a syntax errror, so the code will simply not run.

**Initializations are Not Hoisted :**

1. JavaScript only hoists declarations, not initializations.

**var x = 5;**

**console.log(x + y)  
var y = 7;**

1. In this code all x and y variable are only hoists not initializations

**var x;**

**var y;**

**x = 5;**

**console.log(x + y);**

**y=6**

# Use Strict – ES5

**"use strict"** Defines that JavaScript code should be executed in "strict mode". Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

1. Using a variable, without declaring it, is not allowed:
2. Using an object, without declaring it, is not allowed:
3. Deleting a variable (or object) is not allowed.
4. Deleting a function is not allowed.
5. Duplicating a parameter name is not allowed:
6. Octal numeric literals are not allowed:
7. Octal escape characters are not allowed:
8. Writing to a read-only property is not allowed:
9. Writing to a get-only property is not allowed:
10. Deleting an undeletable property is not allowed:
11. The word eval cannot be used as a variable:
12. The with statement is not allowed
13. **“This”** keyword in functions behaves differently in strict mode

# Object

1. Objects are variables too. But objects can contain many values.
2. Object values are written as name: value pairs (name and value separated by a colon).
3. Dates are always objects
4. Math are always objects
5. Regular expressions are always objects
6. Arrays are always objects
7. Functions are always objects

Objects are always objects

**const person = {**

**firstName:"John",**

**lastName:"Doe",**

**age:50,**

**eyeColor:"blue"**

**};**

## There are different ways to create new objects:

1. **Create a single object, using an object literal.**

**Example 1:**

**const person = {**

**Age: 50,**

**eyeColor:"blue"**

**};**

**Example 2:**

**const person = {};  
person.age = 50;  
person.eyeColor = "blue";**

1. **Create a single object, with the keyword new.**

**const person = new Object ();  
person.age = 50;  
person.eyeColor = "blue";**

1. **Define an object constructor, and then create objects of the constructed type.**
2. **Create an object using Object.create().**

* Object.create() method is used to create a new object with the specified prototype object and properties.
* Object.create() method returns a new object with the specified prototype object and properties.
* Object.create() is used for implementing inheritance.
* Here all child object inheritance parent all method and prototype.

**Syntax:**

**Object.create(prototype , [properties-Object])**

**Parameters Used:**

1. **prototype** : It is the prototype object from which a new object must be created.
2. **Properties-Object** : It is optional parameter. It specifies the enumerable properties to be added to the newly created object.

**Return Value:** it returns a new object with the specified prototype object and properties.

**const city = {**

**name: "dhaka",**

**age : 36**

**}**

**const mym =  Object.create(city); // no result**

**const mym = Object.create(city,name)**

\*\*\*\*\* Objects are mutable: They are addressed by reference, not by value.

## Object.create vs new Object :

**Object.create:**

**let p = function Person(name, age){**

**let person =  Object.create(p.prototype);**

**person.name = name;**

**return person;**

**}**

**p.prototype = {**

**eat(){**

**console.log("eat ");**

**},**

**sleep(){**

**console.log("sleep");**

**}**

**}**

**const sakib = p('sakib');**

**sakib.prototype={ name:"shuvo" };**

In object create method, when we want inherit parents object, we need to create object and return value.

**New Object:**

**let p = function Person(name, age){**

**person.name = name;**

**}**

**p.prototype = {**

**eat(){**

**console.log("eat ");**

**},**

**sleep(){**

**console.log("sleep");**

**}**

**}**

**const sakib = new p('sakib');**

**sakib.prototype={ name:"shuvo" };**

create an object with new keyword not need no assign protype . prototype create when object.

## JavaScript Object Properties and method

1. Properties are the values associated with a JavaScript object.
2. A JavaScript object is a collection of **unordered** properties.
3. Properties can usually be changed, added, and deleted, but some are read only.

**Syntax for accessing the property of an object -----**

1. objectName.property
2. objectName["property"]
3. objectName[expression]
4. **Property in loop:**

**for (let x in person) {**

**txt += person[x];**

**}**

1. **Add property: person.nationality = "English";**
2. **Deleting Property: delete person.age;**
3. **Nested Object:**

**myObj = {**

**name: "John",**

**age:30,**

**cars: {**

**car1:"Ford",**

**car2:"BMW",**

**car3:"Fiat"**

**}**

**}**

1. **Object called:**

**myObj.cars.car2;**

**myObj.cars["car2"];**

**myObj["cars"]["car2"];**

**myObj[p1][p2];**

1. **Nested Arrays and Objects:**
2. Values in objects can be arrays
3. values in arrays can be objects

**const myObj = {**

**name: "John",**

**age: 30,**

**cars: [**

**{ name: "Ford", models:["Fiesta", "Focus", "Mustang"] },**

**{ name: "BMW", models:["320", "X3", "X5"] },**

**{ name: "Fiat", models:["500", "Panda"] }**

**]**

**}**

**for (let i in myObj.cars) {**

**x += "<h1>" + myObj.cars[i].name + "</h1>";**

**for (let j in myObj.cars[i].models) {**

**x += myObj.cars[i].models[j];**

**}**

**}**

## This:

1. In an **object** method, **this** refers to the object.
2. **Alone**, this refers to the global object.
3. In a **function**, this refers to the **global** object.
4. In a **function**, in strict mode, this is **undefined**.
5. In an **event**, this refers to the element that received the **event**.
6. Methods like **call**(), **apply**(), and **bind**() can refer **this** to any **object**

## Accessing Object Methods:

1. You will typically describe fullName() as a method of the person object, and fullName as a property.
2. The fullName property will execute when it is invoked with **().**
3. If you access the fullName **property**, without **()**, it will return the **function definition**:

objectName.methodName()

## Display Objects

1. **Displaying the Object Properties by name:**

const person = {

  name: "John",

  age: 30,

  city: "New York"

};

document.getElementById("demo").innerHTML = person.name;

1. **Displaying the Object Properties in a Loop**

for (let x in person) {

txt += person[x] + " ";

};

**\*\*\* person.x** will not work (Because **x** is a variable).

1. **Displaying the Object using Object.values():**

Any JavaScript object can be converted to an array using

const myArray = Object.values(person);

1. **Displaying the Object using JSON.stringify()**

Any JavaScript object can be stringified (converted to a string) with the JavaScript function.

const person = {

  name: "John",

  age: 30,

  city: "New York"

};  
  
let myString = JSON.stringify(person);

## Object Constructors function:

1. A constructor is a special function that creates and initializes an object instance of a class.
2. In JavaScript, a constructor gets called when an object is created using the new keyword.
3. The purpose of a constructor is to create a new object and set values for any existing object properties.
4. It is considered good practice to name constructor functions with an upper-case first letter.

**Without parameter:**

function User () {

this.name = 'Bob';

}

var user = new User ();

**With parameter:**

function Person (first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

const m2 = new Person ("John", "Doe", 50, "blue");  
const m2 = new Person ("Sally", "Rally", 48, "green");

m2.firstName

1. cannot add a property to an object constructor. But add in object.
2. You cannot add a new method to an object constructor the same way you add a new method to an existing object.

**Constructor Function Vs Object Literal**

1. Object Literal is generally used to create a single object. The constructor function is useful if you want to create multiple objects
2. Each object created from the constructor function is unique. You can have the same properties as the constructor function or add a new property to one object

## Object Prototypes

1. JavaScript is a prototype-based language.
2. Whenever we create a function using JavaScript, JavaScript engine adds a prototype property inside a function.
3. **Prototype property** is basically an object (also known as Prototype object).
4. We can attach methods and properties in a prototype object
5. Which enables all the other objects to inherit these methods and properties.
6. All JavaScript objects inherit properties and methods from a prototype.

<script>

function Person(name, job, yearOfBirth){

    this.name= name;

    this.job= job;

    this.yearOfBirth= yearOfBirth;

}

Const shuvo = new Person(“suvo”,”job”,1323)

</script>

dcdc

### Property adds:

The JavaScript prototype property allows you to add new properties to object constructors:

function Person(first, last, age, eyecolor) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}  
  
Person.prototype.nationality = "English";

### Method adds:

The JavaScript prototype property also allows you to add new methods to objects constructors:

function Person(first, last, age, eyecolor) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}  
  
Person.prototype.name = function() {

  return this.firstName + " " + this.lastName;

};

Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.

# Function

1. JavaScript functions are defined with the function keyword.

2. You can use a function declaration or a function expression.

**Function Declarations:**

**function** **functionName**(**parameters**) {

   // code to be executed

}

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

**function** **myFunction**(a, b) {

return a \* b;

}

**myFunction**();

**Function Expressions**:

1. Function can be store in variable .
2. After a function expression has been stored in a variable, the variable can be used as a function:
3. **the function above is an anonymous function (a function without a name).**
4. Functions stored in variables do not need function names. They are always invoked (called) using the variable name.
5. The function above ends with a semicolon because it is a part of an executable statement.

const x = function (a, b) {return a \* b};

let z = x(4, 3);

**Constructor**

Functions can also be defined with a built-in JavaScript function constructor called Function ().You don’t have to use the function constructor.

const myFunction = new Function ("a", "b", "return a \* b");  
let x = myFunction (4, 3);

**Self-Invoking Functions**

1. A self-invoking expression is invoked (started) automatically, without being called.
2. Function expressions will execute automatically if the expression is followed by **().**
3. Function not necessary any name.

( **Function** () {  
  **let** x = "Hello!!”; // I will invoke myself  
}) ();

## Function Parameters

1. JavaScript function definitions do not specify data types for parameters.
2. JavaScript functions do not perform type checking on the passed arguments.
3. JavaScript functions do not check the number of arguments received.

function functionName(parameter1, parameter2, parameter3) {

  // code to be execute

}

**Default Parameters:**

If a function is called with missing arguments (less than declared), the missing values are set to undefined.

function myFunction(x, y) {

  if (y === undefined) {

y = 2;

}

}

function myFunction (x, y = 2) {

// function code

}

## The Arguments

1. JavaScript functions have a built-in object called the arguments object.
2. Arguments are the real values passed to (and received by) the function.
3. **arguments** name array receives all the argument passed in function.

**x** = **findMax**(1, 123, 500, 115, 44, 88); // argument value……

**function** **findMax**() {

**let** **max** = -Infinity;

**for** (let i = 0; i < **arguments**.length; i++) {

**if** (**arguments**[i] > max) {

**max** = arguments[i];

     }

  }

**return** max;

}

1. JavaScript arguments are passed by value: The function only gets to know the values, not the argument's locations.
2. If a function changes an argument's value, it does not change the parameter's original value.
3. Changes to arguments are not visible (reflected) outside the function.

# Arrow Functions ES-6

1. Arrow functions allows a short syntax for writing function expressions.
2. You don't need the function keyword, the return keyword, and the curly brackets.
3. Arrow functions are not hoisted. They must be defined before they are used.
4. Arrow functions do not have their own this. They are not well suited for defining object methods.
5. Arrow functions aren't suitable for call, [apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) and [bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) methods,
6. Arrow functions cannot be used as constructors.

// ES5

var x = function(x, y) {  
  return x \* y;  
}

// ES6

const x = (x, y) => x \* y;

const x = (x, y) => { return x \* y };

Advanced syntax:

1. To return an object literal expression requires parentheses around expression:

(params) => ({ foo: "a" }) // returning the object { foo: "a" }

1. Rest parameters are supported, and always require parentheses:

(a, b, ...r) => expression

1. Default parameters are supported, and always require parentheses:

(a=400, b=20, c) => expression

1. Destructuring within params is supported, and always requires parentheses:

([a, b] = [10, 20]) => a + b ; // result is 30

({ a, b } = { a: 10, b: 20 }) => a + b; // result is 30

1. Arrow functions used as methods

'use strict';

const obj = { // does not create a new scope

i: 10,

b: () => console.log(this.i, this),

c() {

console.log(this.i, this);

},

}

obj.b(); // prints undefined, Window { /\* … \*/ } (or the global object)

obj.c(); // prints 10, Object { /\* … \*/ }

# Function Methods

1. call (),
2. apply (),
3. bind ()
4. **call () method :**
5. In JavaScript all functions are object methods.
6. With the call() method, you can write a method that can be used on different objects.

const person = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

const person1 = {

firstName:"John",

lastName: "Doe"

}

const person2 = {

firstName:"Mary",

lastName: "Doe"

}

person.fullName.call(person1);

**The call() Method with Arguments:**

const person = {

fullName: function(city, country) {

return this.firstName + " " + this.lastName + "," + city + "," + country;

}

}

const person1 = {

firstName:"John",

lastName: "Doe"

}  
person.fullName.call(person1, "Oslo", "Norway");

1. **apply() Method:**
2. The apply() method is similar to the call()
3. With the apply() method, you can write a method that can be used on different objects.
4. The call() method takes arguments separately.
5. The apply() method takes arguments as an array.
6. The apply() method is very handy if you want to use an array instead of an argument list

const person = {

fullName: function(city, country) {

return this.firstName + " " + this.lastName + "," + city + "," + country;

}

}  
const person1 = {

firstName:"John",

lastName: "Doe"

}  
  
person.fullName.apply(person1, ["Oslo", "Norway"]); // array in apply()

person.fullName.call(person1, "Oslo", "Norway"); // call method

1. **bind() method :**

With the bind() method, an object can borrow a method from another object.

const person = {

firstName:"John",

lastName: "Doe",

fullName: function () {

return this.firstName + " " + this.lastName;

}

}  
  
const member = {

firstName:"Hege",

lastName: "Nilsen",

}  
  
let fullName = person.fullName.bind(member);

1. Closures

Global variables can be made local (private) with closures.

# Asynchronous

1. JavaScript functions are executed in the sequence they are called.
2. Not in the sequence they are defined.

function myFirst() {

  myDisplayer("Hello");

}

function mySecond() {

   myDisplayer("Goodbye");

}  
myFirst();

mySecond();

mySecond();

myFirst();

Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation, and then display the result.

**We can do it as below:**

function myDisplayer(some) {

  document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2) {

  let sum = num1 + num2;

  return sum;

}

let result = myCalculator(5, 5);

myDisplayer(result);

**Other Way:**

function myDisplayer(some) {

  document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2) {

  let sum = num1 + num2;

  myDisplayer(sum);

}

myCalculator(5, 5);

The problem with the first example above, is that you have to call two functions to display the result. The problem with the second example, is that you cannot prevent the calculator function from displaying the result.

### How JavaScript Asynchronous Works:

1. when JavaScript code run in browser, all code converts into machine language. Then browser interpret all the code.
2. Browser have two things.
   * + 1. runtime
       2. engine
3. **Runtime:**

Runtime is the environment in which a programming language executes. JavaScript’s runtime majorly constitutes three things namely JavaScript Engine, Web API, Call stack. JavaScript can work with asynchronous code as well as synchronous code.

The unique feature of JavaScript’s runtime is that even though JavaScript’s interpreter is single-threaded, it can execute multiple codes at a time using concurrent fashion in a non-blocking way. This enables asynchronous behavior. As the interpreter is not multithreaded, it rules out parallelism.

1. **JavaScript Engine:**

JavaScript engine can be considered as the heart of the runtime. It is the place where each code is executed. JavaScript engine constitutes of Heap storage and call stack. Let’s understand each of those.  JavaScript is a single-threaded language. This means it has only one call stack and one memory heap. Hence, it can only execute one code at a time. In other words, the code is executed in an orderly fashion. It must execute one code in the call stack before moving to the next code to be executed. There are two types of code tasks in JavaScript, asynchronous code which runs and gets executed after certain loading, synchronous, which gets executed instantaneously.

**Heap :**

It is the place where all the objects and data are stored. This is similar to the heap storage we see on various other languages like C++, Java, etc. It contains the store of the data related to all the objects, arrays, etc. that we create in the code.

**Call Stack:**

It is the place where the code is stacked before the execution. It has the properties of a basic stack (first in last out). Once a coding task is stacked into the call stack, it will be executed. There is an event loop that takes place and this is the one that makes the JavaScript interpreter smart. It is responsible for concurrent behavior.

**Web API:**

JavaScript has the access to different web API’s and it adds a lot of functionality. For example, JavaScript has the access to the DOM API, which gives access to the DOM tree to JavaScript. Using this, we can make changes to the HTML elements present on the browser. Also, you can think of the timer, which gives it access to the time-related functions, etc. Also, the geolocation API which gives it access to the location of the browser. Like this, JavaScript has the access to various other APIs.

**Callback Queue:**

This is the place where asynchronous code is queued before passing to the call stack. The passing of the code task from the callback queue to the call stack is taken care of by the event loop. In addition to this, there is also a micro tasks queue.

### JavaScript Callbacks

A function passed as an argument to another function is called callbacks. Callbacks are just the name of a convention.

When doing a complex task, we break that task down into smaller steps. To help us establish a relationship between these steps according to time (optional) and order, we use callbacks.

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}  
  
function myCalculator (num1, num2, myCallback) {

let sum = num1 + num2;

myCallback(sum);

}  
myCalculator (5, 5, myDisplayer);

\*\*\*\* When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong ~~: myCalculator(5,5, MyDisplayer);~~

**Callback Hell:**

Callback hell is where there are multiple nested callbacks.

const makeBurger = () => {

getBeef(function(beef) {

cookBeef(beef, function(cookedBeef) {

getBuns(function(buns) {

putBeefBetweenBuns(buns, beef, function(burger) {

// Serve the burger

});

});

});

});

};

### Asynchronous

1. Functions running in parallel with other functions are called asynchronous
2. In the real world, callbacks are most often used with asynchronous functions.

Asynchronous function:

1. setTImeout()
2. setInterval()

**setTimeout() function :**

setTimeout(myFunction, 3000);

function myFunction () {

document.getElementById("demo").innerHTML = "I love You !!";

}

**setInterval() function :**

console.log(“dlkj”)

setInterval(myFunction, 1000);  
  
function myFunction() {

console.log(“function”);

}

Console.log(“finish”)

### Promises

You write a function **A ()** that fetch all the data from other website. After fetch data you show all the data in a table. If data not fetch then table not show.

Here we learn two terms –

* **producing code**: that does something and takes time. Here **A ()** is producing code.
* **consuming code:** that wants the result of the “producing code” once it’s ready. Here when data fetch is complete **table will show.**

1. A promise is a special JavaScript object that links the “producing code” and the “consuming code” together.
2. Promises is an object which is invented to solve the problem of callback hell and to better handle our tasks.
3. You must use a Promise method to handle promises.

The Constructor Syntax for A Promise Object Is:

let promise = new Promise (function (resolve, reject) {

// executor (the producing code, "A()")

});

1. The function passed to new Promise is called the **executor**. When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result
2. Its arguments **resolve** and **reject** are callbacks provided by JavaScript itself. Our code is only inside the executor.
3. When the executor obtains the result, be it soon or late, doesn’t matter, it should call one of these callbacks:
   1. resolve(value) — if the job is finished successfully, with result value.
   2. reject(error) — if an error has occurred, error is the error object.



Promises object has two properties:

* + - * 1. state
        2. result

|  |  |  |
| --- | --- | --- |
| **State value** | **Means** | **Result** |
| Pending | initial state, neither fulfilled nor rejected | Undefined |
| Fulfilled | meaning that the operation was completed successfully. | Result value |
| Rejected | meaning that the operation failed. | Error value |



**Example:**

let promise = new Promise (function (resolve, reject) {

setTimeout (() => resolve("done"), 1000);

});

We can see two things by running the code above:

1. The executor is called automatically and immediately (by new Promise).
2. The executor receives two arguments: **resolve and reject**. These functions are pre-defined by the JavaScript engine, so we don’t need to create them. We should only call one of them when ready.

**Resolve called:**

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done"), 1000);

});

After one second of “processing”, the executor calls resolve("done") to produce the result. This changes the state of the promise object:

That was an example of a successful job completion, a “fulfilled promise”.

**Reject called:**

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

The call to reject(...) moves the promise object to "rejected" state:

**Promise Consumers:**

A Promise object serves as a link between the executor (the “producing code”) and the consuming functions, which will receive the result or error.

Consuming functions can be registered using the methods-

**1) .then**

**2) .catch**

**then ():**

then() is invoked when a promise is either resolved or rejected. It may also be defined as a career which takes data from promise and further executes it successfully.

then() syntax –

.then( **peram\_1** , **peram\_2** );

then() method takes two functions as parameters.

1. **Param\_1:** It is a function. First function is executed if promise is resolved and a result is received.
2. **Parram\_2:** It is also a Function. Second function is executed if promise is rejected .

If we’re interested only in successful completions, then we can provide only one function argument to .then:

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert);

**catch()**

catch() is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error

.catch( **param\_1** )

**Param\_1: it is a function.**  Function to handle errors or promise rejections.

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

promise.catch(alert);

We can use then() as like catch.

.then( null, param\_2);

Here param\_2 take error as like catch().

**Finally() :**

1. The .finally() handler performs cleanups like stopping a loader, closing a live connection, and so on.
2. The finally() method will be called irrespective of whether a promise resolves or rejects.
3. It passes through the result or error to the next handler which can call a .then() or .catch() again.
4. A finally handler has no arguments. In finally we don’t know whether the promise is successful or not.
5. A finally handler also shouldn’t return anything. If it does, the returned value is silently ignored.

let loading = true;

loading && console.log('Loading...');

promise = getPromise(ALL\_POKEMONS\_URL);

promise.finally(() => {

loading = false;

console.log(`Promise Settled and loading is ${loading}`);

}).then((result) => {

console.log({result});

}).catch((error) => {

console.log(error)

});

**Promise Chain**

The  promise.then() call always returns a promise. This promise will have the state as pending and result as undefined. It allows us to call the next .then method on the new promise.

let promise = getPromise(ALL\_POKEMONS\_URL);

promise.then(result => {

let onePokemon = JSON.parse(result).results[0].url;

return onePokemon;

}).then(onePokemonURL => {

console.log(onePokemonURL);

return getPromise(onePokemonURL);

}).then(pokemon => {

console.log(JSON.parse(pokemon));

}).catch(error => {

console.log('In the catch', error);

});

**How to Handle Multiple Promises**

1. Promise.all
2. Promise.any
3. Promise.allSettled
4. Promise.race
5. Promise.resolve
6. Promise.reject

### JavaScript Async

1. "async and await make promises easier to write"
2. async makes a function return a Promise
3. await makes a function wait for a Promise

Async Syntax:

async function myFunction() {

return "Hello";

}

myFunction().then(

  function(value) {myDisplayer(value);},

  function(error) {myDisplayer(error);}

);

Await Syntax:

1. The await keyword can only be used inside an async function.
2. The await keyword makes the function pause the execution and wait for a resolved promise before it continues:

# JSON

1. JSON is a format for storing and transporting data.
2. JSON is often used when data is sent from a server to a web page.
3. JSON stands for JavaScript Object Notation
4. JSON is a lightweight data interchange format
5. JSON is language independent \*
6. JSON is "self-describing" and easy to understand

**JSON Example:**

{

"employees": [

 { "firstName":"John", "lastName":"Doe" },

{ "firstName":"Anna", "lastName":"Smith" },

{ "firstName":"Peter", "lastName":"Jones" }

]

}

**JSON Syntax Rules:**

1. Data is in name/value pairs
2. Data is separated by commas
3. Curly braces hold objects
4. Square brackets hold arrays

**JSON strings into JavaScript objects:**

Use the JavaScript built-in function JSON.parse() to convert the string into a JavaScript object:

let text = '{ "employees" : [' +

'{ "firstName":"John" , "lastName":"Doe" },' +

'{ "firstName":"Anna" , "lastName":"Smith" },' +

'{ "firstName":"Peter" , "lastName":"Jones" }

]

}';

const obj = JSON.parse(text);

**Converting An Object into a JSON string:**

Use the JavaScript built-in function JSON.stringify() to convert object into a JSON string

# Class

1. Use the keyword class to create a class.
2. Always add a method named constructor():
3. Then add any number of methods.

class ClassName {

  constructor() { ... }

  method\_1() { ... }

  method\_2() { ... }

  method\_3() { ... }

}

class Car {

  constructor(name, year) {

    this.name = name;

    this.year = year;

  }

  age() {

    let date = new Date();

    return date.getFullYear() - this.year;

  }

}

let myCar1 = new Car("Ford", 2014);  
let myCar2 = new Car("Audi", 2019);

### Class Inheritance:

To create a class inheritance, use the extends keyword.

class Car {

constructor(brand) {

    this.carname = brand;

  }

  present() {

    return 'I have a ' + this.carname;

  }

}  
class Model extends Car {

  constructor(brand, mod) {

    super(brand);

    this.model = mod;

  }

  show() {

    return this.present() + ', it is a ' + this.model;

  }

}

The super () method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

### Getters and Setters:

It can be smart to use getters and setters for your properties, especially if you want to do something special with the value before returning them, or before you set them.

To add getters and setters in the class, use the get and set keywords.

class Car {

  constructor(brand) {

    this.carname = brand;

  }

  get cnam() {

    return this.carname;

  }

  set cnam(x) {

    this.carname = x;

  }

}

even if the getter is a method, you do not use parentheses when you want to get the property value.

# Web API

* It can extend the functionality of the browser
* It can greatly simplify complex functions
* It can provide easy syntax to complex code

### Form validate API

**Method:**

1. **setCustomValidity():** Sets the validation Message property of an input Element.
2. **checkValidity():** Returns true if an input element contains valid data.

**Example:**

<input id="id1" type="number" min="100" max="300" required>

<button onclick="myFunction()">OK</button>

<p id="demo"></p>

<script>

function myFunction() {

  const inpObj = document.getElementById("id1");

  if (!inpObj.checkValidity()) {

    document.getElementById("demo").innerHTML = inpObj.validationMessage;

  }

}

</script>

**Validation DOM Properties:**

1. **validity :** Contains boolean properties related to the validity of an input element.
2. **validationMessage :** Contains the message a browser will display when the validity is false.
3. **willValidate:** Indicates if an input element will be validated.

**Validity Properties:**

1. **customError**: Set to true, if a custom validity message is set.
2. **patternMismatch**: Set to true, if an element's value does not match its pattern attribute.
3. **rangeOverflow**: Set to true, if an element's value is greater than its max attribute.
4. **rangeUnderflow**: Set to true, if an element's value is less than its min attribute.
5. **stepMismatch**: Set to true, if an element's value is invalid per its step attribute.
6. **tooLong** : Set to true, if an element's value exceeds its maxLength attribute.
7. **typeMismatch**: Set to true, if an element's value is invalid per its type attribute.
8. **valueMissing**: Set to true, if an element (with a required attribute) has no value.
9. **Valid:** Set to true, if an element's value is valid.

**Example-1:**

<input id="id1" type="number" max="100">

<button onclick="myFunction()">OK</button>

<p id="demo"></p>

<script>

function myFunction() {

  let text = "Value OK";

  if (document.getElementById("id1").validity.rangeOverflow) {

    text = "Value too large";

  }

}

</script>

### History API

The Web History API provides easy methods to access the windows.history object.

**Example :**

<button onclick="myFunction()">Go Back</button>  
  
<script>

function myFunction() {

 window.history.back();  
}

</script>

**Method && Properties:**

1. [length](https://www.w3schools.com/jsref/prop_his_length.asp): Returns the number of URLs in the history list
2. [back()](https://www.w3schools.com/jsref/met_his_back.asp): Loads the previous URL in the history list
3. [forward()](https://www.w3schools.com/jsref/met_his_forward.asp): Loads the next URL in the history list
4. [go()](https://www.w3schools.com/jsref/met_his_go.asp): Loads a specific URL from the history list

### Storage API

The Web Storage API is a simple syntax for storing and retrieving data in the browser.

**There are two type of storage:**

**A) localStorage Object:**

1. The localStorage object provides access to a local storage for a particular Web Site. It allows you to store, read, add, modify, and delete data items for that domain.
2. The data is stored with no expiration date, and will not be deleted when the browser is closed.
3. The data will be available for days, weeks, and years.

**Example:**

localStorage.setItem("name", "John Doe");

localStorage.getItem("name");

B**) sessionStorage Object:**

1. The sessionStorage object is identical to the localStorage object.
2. The difference is that the sessionStorage object stores data for one session.
3. The data is deleted when the browser is closed.

**Example:**

sessionStorage.setItem("name", "John Doe");

sessionStorage.getItem("name");

**Method and Property:**

1. **key**(n): Returns the name of the nth key in the storage
2. **length**: Returns the number of data items stored in the Storage object
3. **getItem**(keyname): Returns the value of the specified key name
4. **setItem**(keyname, value): Adds a key to the storage, or updates a key value (if it already exists)
5. **removeItem**(keyname): Removes that key from the storage
6. **clear** (): Empty all key out of the storage

### Workers API

A web worker is a JavaScript running in the background, without affecting the performance of the page.

### Fetch

The fetch () method in JavaScript is used to request data from a server. The request can be of any type of API that return the data in JSON or XML.

Syntax:

fetch (**param\_1, param\_2**)

. then ( response => response.json() )

. then ( data => console.log(data) );

**Param\_1:** the URL to access.

**Param\_2:** It is an array of properties. It is an optional parameter.

**Example:**

const data = { username: 'example' };

         let options = {

            method: 'POST',

            headers: {

                 'Content-Type': 'application/json;

charset=utf-8'

            },

            body: JSON.stringify(data)

         }

let fetchRes = fetch ( **"http://dummy.restapiexample.com/api/v1/create**" **,** **options** );

fetchRes.then(res => res. json () )

. then ( d => { console.log(d) })

**Async Await:**

With Async Await method with fetch() method to make promises in a more concise way. Async functions are supported in all modern browsers.

Syntax:

async function funcName(url){

const response = await fetch(url);

var data = await response.json();

}

# DOM – 65 (yahoo)

## DOM targeting method

**getElementById**:

**getElementByTagName**:

**getElementByclass**: it target all tag with same class name.

**querySelector**:

1. we can target the value with help CSS selector.
2. It target only first node.

**Id(#) select:**

Document.querySelector(“#main”).innerHTML;

**Class(.) selector:**

Document.querySelector(“.main”).innerHTML;

**querySelectorAll**:

1. we can target the value with help CSS selector.
2. It target all node.
3. It return a array

**Id(#) select:**

Document.querySelectorAll(“#main”).innerHTML;

**Class(.) selector:**

Document.querySelectorAll(“.main”).innerHTML;

## Get and set

In JavaScript, we can get/set 3 things with help of DOM.

1. HTML
2. Text
3. Attribute

### GET :

Method for get value-

1. **innerText:**  it return all text only inside the tag.

**Code** :

**<div>**

**<p>The text content of the button element is:</p>**

**<p id="demo"> </p>**

**<p>The innerText property and earlier.</p>**

**</div>**

**var x = Document.getElementById(“main”).innerText;**

**Output**:

**The text content of the button element is.**

**The innerText property and earlier**

1. **innerHTML:** it return all text and tag inside the tag.

**Code**:

**<div>**

**<p>The text content of the button element is:</p>**

**<p id="demo"> </p>**

**<p>The innerText property and earlier.</p>**

**</div>**

**var x = Document.getElementById(“main”).innerHTML;**

**Output**:

**<div>**

**<p>The text content of the button element is:</p>**

**<p id="demo"> </p>**

**<p>The innerText property and earlier.</p>**

**</div>**

1. **getAttribute**: with this we can get attribute value of tag.

**code**:

**<div id=”head” class=”container”>**

**var x = Document.getElementById(“head”).getAttribute(“class”);**

**output**: container.

1. **Attribute:** it return all attribute of a tag in array.

**code**:

**<div id=”head” class=”container” style=” color: red”>**

**var x = Document.getElementById(“head”).Attribute[1];**

**var x = Document.getElementById(“head”).Attribute[2];**

**output**:

class=”container “

style=”color:red”

**Attribute also have properties –**

**name**: with name properties we can find name of attribute.

**var x = Document.getElementById(“head”).Attribute[2].name ;**

**output:** style

**value** : it return value of attribute .

**var x = Document.getElementById(“head”).Attribute[2].value ;**

**output:** color : red

### SET

1. **innerText:**  we can change text inside tag.

**Document.getElementById(“main”).innerText=”hello world”;**

1. **innerHTML:** we can change tag inside the tag.

**Document.getElementById(“main”).innerHTML=”<h1> doremon </h1>**

1. **setAttribute**: we can set attribute and value .

**Document.getElementById(“head”).setAttribute(“class”,”syz”);**

## CSS style

**style :**

1. with this we can get element style.
2. We can change CSS of element.
3. To set value for CSS property we use camel case.

**<p id=”main” style=” border:1px solid red; color:black; ”> hello world </p>**

**document.querySelector(“#main”).style.backgroundColor=”blue”;**

**className:**

1. We can get class attribute value of an element.
2. we can set class in an element .
3. it return string.

**<p style=” border:1px solid red; color: black; ”> hello world </p>**

**document.querySelector(“#main”).className =”blue**”

**classList:**

1. We can get class attribute value of an element.
2. we can set class in an element .
3. it return array of class.

**<p style=” border:1px solid red; color: black; ”> hello world </p>**

**document.querySelector(“#main”).classList =”blue**”

**method of classList:**

**add():** we add class in element by it.

**<p style=” border:1px solid red; color: black; ”> hello world </p>**

**document.querySelector(“#main”).classList.add(“xyz”)** ;

**remove():** we can remove class attribute .

**<p style=” border:1px solid red; color: black; ”> hello world </p>**

**document.querySelector(“#main”).classList.remove(“xyz”)** ;

**Length**: we can count how many class in element.

**toggle** : we toggle class attribute by this method.

## EVENT --66

1. Click (onclick)
2. Double click ( ondbclick)
3. Right click (onContextMenu)
4. Mouse hover (onmouseenter)
5. Mouse Out (onmouseout)
6. Mouse Down (onmousedown)
7. Mouse up (onmouseup)
8. Key press (onkeypress)
9. Key Up (onkeyup)
10. Load (onload)
11. Unload (onunload)
12. Resize (onresize)
13. Scroll (onscroll)

## addEventListener:

with this we can add a event handler to an element.

**Syntax**: **element.addEventListener(event, function, useCapture);**

**event**: The name of the event.

**function:** The function to run when the event occurs.

**useCapture** :

1. Optional (default = false).
2. false – it called inner div first .
3. true - it called outer div first.

**Code :**

**<button id=”main”> click me </button>**

**Document.getElementById(“main”).addEventListener(“click”, function(){**

**console.log(“hello world”);**

**})**

### removeEventListener :

we can remove a event from a element with this.

**<button id=”main” onclick=”abc()”> click me </button>**

**Document.getElementById(“main”).addEventListener(“click”, function(){**

**document.getElementById(“main”).removeEventListener(“click”,”abc”)**

**})**

## Traversal Method

1. parentNode
2. parentElement
3. Children
4. childNOdes
5. firstChild
6. firstElementChild
7. lastChild
8. lastElementChild
9. nextElementSibling
10. nextSibling
11. previousElementSibling
12. previousSibling