JavaScript

# How can use JavaScript in code?

1. Javascript code use between <script> ….. </script> tag.

Example:<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. JavaScript is the default scripting language in HTML.

2. You can place any number of scripts in an HTML document.

3. Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

4. for using external script file, file name use 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.

2. Semicolons separate JavaScript statements.

3. JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.

**JavaScript Values**

The JavaScript syntax defines two types of values:

1. Fixed values (Literal)
2. Variable values (Variables)

\*\* Numbers are written with or without decimals:

\*\* Strings are text, written within double or single quotes

\*\* All JavaScript identifiers are case sensitive.

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

**4 Ways to Declare a JavaScript Variable:**

1. Using var
2. Using let
3. Using const
4. Using nothing

**Let**

1. Variables defined with let cannot be Redeclared in same block.
2. Variables defined with let must be Declared before use.
3. Variables defined with let have Block Scope.

**Const**

1. Variables defined with const cannot be Redeclared.
2. Variables defined with const cannot be Reassigned.
3. Variables defined with const have Block Sco
4. 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.

Because of this you can NOT:

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

But you CAN:

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

**Data Types**

1. Number
2. String
3. Boolean
4. Object
5. Undefined

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

# Strings

1. You can use single or double quotes
2. You can use quotes inside a string, as long as they don't match the quotes surrounding the string

Comparing two JavaScript objects **always** returns **false**.

Code Result Description

\' ' Single quote

\" " Double quote

\\ \ Backslash

Code Result

\b Backspace

\f Form Feed

\n New Line

\r Carriage Return

\t Horizontal Tabulator

\v Vertical Tabulator

\*\* Do not create Strings objects.

**String Methods**

1. slice(start, end) : slice() extracts a part of a string and returns the extracted part in a new string.
2. substring(start, end): substring() is similar to slice().
3. substr(start, length)
4. replace()The replace() method replaces a specified value with another value in a string.
5. replace() method is case sensitive
6. toUpperCase():
7. toLowerCase():
8. concat()
9. concat() joins two or more strings:
10. trim()
11. padEnd()
12. charAt()
13. split()

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

**String Search**

1. String indexOf()
2. String lastIndexOf()
3. String startsWith()
4. String endsWith()

# 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:**

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

const p = cars.length;

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

Array.isArray(cars);

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

cars instanceof Array;

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

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

console.log(fruits.toString());

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

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

fruits.join(" \* ");

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

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

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

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

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

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

1. **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];

1. **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);

1. **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

1. **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");

1. **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.
2. s**ort ():**
3. String Sort: methos sort the array alphabetically.
   * 1. const fruits = ["Banana", "Orange", "Apple", "Mango"];  
        fruits.sort();
4. 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});
5. Random Order:
   * 1. const points = [40, 100, 1, 5, 25, 10];  
        points.sort(function(){return 0.5 - Math.random()});
6. **reverse() :** reverses the elements in an array.
7. **Math.max()**
8. **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;  
}

**reduce() 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()**

# 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](https://www.w3schools.com/js/js_es6.asp))
13. Math.random()

# 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");

# 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);

# 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/ in 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

}

# 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. **new Map() Method:**

Creating a Map by passing an Array to the new Map() constructor:

const fruits = new Map ([

  ["apples", 500],

  ["bananas", 300],

  ["oranges", 200]

]);

1. **Map.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:**

The get() method 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:**

The forEach() 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:**

The entries() method returns an iterator object with the [key, values] in a Map:

let text = "";

for (const x of fruits.entries()) {

  text += x;

}

# Modules ------ES6

JavaScript modules rely on the import and export statements.

1. **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

1. Block scope
2. Function scope
3. Global scope

# **Hoisting**

# **Use Strict**

**"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**

\*\*\*\* Primitives: A primitive value is a value that has no properties or methods. Primitive values are immutable (they are hardcoded and cannot be changed). if x = 3.14, you can change the value of x, but you cannot change the value of 3.14

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

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.**

An object literal is a list of name:value pairs (like age:50) inside curly braces {}.

Example 1:

const person = {

firstName:"John",

lastName:"Doe",

age:50,

eyeColor:"blue"

};

Example 2:

const person = { };  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";

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

const person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
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().

JavaScript Objects Are Mutable

1. Objects are mutable: They are addressed by reference, not by value.
   * + - 1. const x = person;
2. The object x is not a copy of person. It is person. Both x and person are the same object.
3. Any changes to x will also change person, because x and person are the same 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.

The syntax for accessing the property of an object is:

1. objectName.property
2. objectName["property"]
3. objectName[expression]

The JavaScript for...in statement loops through the properties of an object.

for (let variable in object) {

  // code to be executed

}

**Nested Object:**

myObj = {

  name:"John",

  age:30,

  cars: {

    car1:"Ford",

    car2:"BMW",

    car3:"Fiat"

  }

}

Object called:

myObj.cars.car2;

myObj.cars["car2"];

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

**Nested Arrays and Objects:**

Values in objects can be arrays, and 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];

  }

}

**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.

**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");

\*\* It is considered good practice to name constructor functions with an upper-case first letter.

**Constructor Function Vs Object Literal**

Object Literal is generally used to create a single object. The constructor function is useful if you want to create multiple objects.

\*\*using object literal

let person = {

name: 'Sam'

}

\*\* using constructor function

function Person () {

this.name = 'Sam'

}

let person1 = new Person ();

**Adding property and method in construction function:**

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

this.firstName = first;

   this.lastName = last;

   this.age = age;

   this.eyeColor = eyecolor;

   this.nationality = "English";

}

const ob1 = new Person ("John", "Doe", 50, "blue");  
const ob2 = new Person ("Sally", "Rally", 48, "green");

**Adding property:**

1. Adding a new property to an existing object

Ob1.nationality = "English";

You cannot add a new property to an object constructor the same way you add a new property to an existing object:

Person.nationality = "English";

1. To add a new property to a constructor, you must add it to the constructor function

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

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eyecolor;

this.nationality = "English";

}

**Adding method:**

1. Adding a new method to an existing object :

myFather.name = function () {

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

};

1. Your constructor function can also define methods:

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

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eyecolor;

this.name = function() {

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

};

}

Use of construction function :

Property call:

Obj.lastname;

Method call :

Obj.name();

**Object Prototypes**

1. All JavaScript objects inherit properties and methods from a prototype.
2. We cannot add a new property and method to an existing object constructor:
3. But with prototype we can add property and method in object.

Proparty add:

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

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}  
  
Person.prototype.nationality = "English";

method add:

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;  
};

# Function

1. JavaScript functions are defined with the function keyword.

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

1. **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();

1. **Function Expressions**:

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

let z = x(4, 3);

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

1. **Function () Constructor**

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

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

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

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

1. **JavaScript functions can be used as values:**

function myFunction(a, b) {  
  return a \* b;  
}  
let x = myFunction(4, 3);

1. **JavaScript functions can be used in expressions:**

function myFunction(a, b) {  
  return a \* b;  
}  
let x = myFunction(4, 3) \* 2;

**JavaScript functions have both properties and methods.**

1. arguments.length
2. toString()
3. **Arrow Functions: ES-6**
4. Arrow functions allows a short syntax for writing function expressions.
5. You don't need the function keyword, the return keyword, and the curly brackets.
6. Arrow functions are not hoisted. They must be defined before they are used.
7. Arrow functions do not have their own this. They are not well suited for defining object methods.
8. 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,
9. 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#advanced_syntax):

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

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

1. [Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) are supported, and always require parentheses:

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

1. [Default parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters) are supported, and always require parentheses:

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

1. [Destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#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 { /\* … \*/ }

1. **Function Parameters:**

parameters are the names listed in the function definition.

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

}

1. **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

}

1. **The Arguments:**
2. JavaScript functions have a built-in object called the arguments object.
3. Arguments are the real values passed to (and received by) the function.
4. **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

# 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

JavaScript functions are executed in the sequence they are called.

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 with 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.

## JavaScript Callbacks

A callback is a function passed as an argument to another function.

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);~~

## 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**

# 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.

# AJAX

# 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 validationMessage 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