**JAVASCRIPT**

* **JavaScript** to program the behavior of web pages.
* Web pages are not the only place where JavaScript is used. Many desktop and server programs use JavaScript. Node.js is the best known. Some databases, like MongoDB and CouchDB, also use JavaScript as their programming language.
* JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else.
* JavaScript Can Change HTML Content with innerHTML.
* features of JavaScript:
  + JavaScript is open and cross-platform.
  + integrated with HTML perfectly
  + make web page dynamic
* **Advantages of JavaScript**
  + **Speed.** Being client-side, JavaScript is very fast because any code functions can be run immediately instead of having to contact the server and wait for an answer.
  + **Simplicity.** JavaScript is relatively simple to learn and implement.
  + **Versatility.** JavaScript plays nicely with other languages and can be used in a huge variety of applications. Unlike PHP or SSI scripts, JavaScript can be inserted into any web page regardless of the file extension. JavaScript can also be used inside scripts written in other languages such as Perl and PHP.
  + **Server Load.** Being client-side reduces the demand on the website server.

**Disadvantages of JavaScript**

* **Security.** Because the code executes on the users' computer, in some cases it can be exploited for malicious purposes. This is one reason some people choose to disable JavaScript.
* **Reliance on End User.** JavaScript is sometimes interpreted differently by different browsers. Whereas server-side scripts will always produce the same output, client-side scripts can be a little unpredictable. Don't be overly concerned by this though - as long as you test your script in all the major browsers you should be safe.
* Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.
* Placing scripts at the bottom of the <body> element improves the display speed, because script compilation slows down the display.
* To use an external script,

<script src="myScript.js"></script>

**Placing scripts in external files has some advantages:**

* It separates HTML and code
* It makes HTML and JavaScript easier to read and maintain
* Cached JavaScript files can speed up page loads
* **JavaScript can "display" data in different ways:**
  + Writing into an HTML element, using innerHTML.
  + Writing into the HTML output using document.write().
  + Writing into an alert box, using window.alert().
  + Writing into the browser console, using console.log().
* **document.write():**
  + For testing purposes, it is convenient to use document.write()
  + Using document.write() after an HTML document is fully loaded, will delete all existing HTML
* For debugging purposes, you can use the console.log() method to display data.
* JavaScript keywords are reserved words. Reserved words cannot be used as names for variables.
* Break: Terminates a switch or a loop
* continue: Jumps out of a loop and starts at the top
* debugger: Stops the execution of JavaScript, and calls (if available) the debugging function
* do ... while: Executes a block of statements, and repeats the block, while a condition is true
* for Marks a block of statements to be executed, as long as a condition is true
* return: Exits a function
* try ... catch: Implements error handling to a block of statements
* var: Declares a variable
* Fixed values are called literals. Variable values are called variables.
* In JavaScript, the first character of identifiers or names must be a letter, or an underscore (\_), or a dollar sign ($).
* All JavaScript **variables** must be **identified** with **unique names**. These unique names are called **identifiers**.
* **Instanceof**: Returns true if an object is an instance of an object type.
* You can consider it a bug in JavaScript that typeof null is an object. It should be null.
* Undefined and null are equal in value but different in type:
  + typeof undefined           // undefined  
    typeof null                // object  
      
    null === undefined         // false  
    null == undefined          // true
* **Primitive data type: The typeof operator can return one of these primitive types:**
  + string
  + number
  + boolean
  + undefined
* **Complex Data Type:** The typeof operator can return one of two complex types:
  + function
  + object
* The typeof operator returns object for both objects, arrays, and null. The typeof operator returns "object" for arrays because in JavaScript arrays are objects.
* A JavaScript function is a block of code designed to perform a particular task.
  + Advantages 0f function:
    - You can reuse code
* JavaScript objects are variables to contain multiple values in the form of key value pair.
* **backslash escape character**
* Objects cannot be compared:

var x = new String("John");               
var y = new String("John");  
// (x == y) is false because x and y are different objects

* How can you get the reference of a caller function inside a function?

The arguments object has a callee property, which refers to the function you're inside of. For example −

function func() {

return arguments.callee;

}

func();

* A local variable takes precedence over a global variable with the same name.
* **What is callback?**

A callback is a plain JavaScript function passed to some method as an argument or option. Some callbacks are just events, called to give the user a chance to react when a certain state is triggered.

* Function overriding with prototype:

String.prototype.split = function(param){

console.log('over-ride');

}

console.log('abcd'.split(''));

* **Javascript Inheritance**: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance>
* **Write a polyfill for Object.create() method if it is not present in the browser?**

ECMAScript 5.1 has this method in its specification. If the browser is old then Object.create () method is not present.To resolve this we need to write a polyfill. Below code shows the polyfill for Object. Create ()method.

//check if create method is present inside Object  
if (tvpeof Object.create ! = ‘function’) {  
I/define the create method  
Object.create = (function() {  
var Object = function() {};  
return function (prototype) {  
If (argurnents.length> 1) {  
throw Error (’Second argument not supported’);  
}  
if (arguments. length> 1){  
throw Error (‘Second argument not supported’);  
}  
if (typeof prototype ! = ‘object’) {  
throw TypeError(’Argument must be an object’);  
}  
Object. prototype = prototype;  
var result = new Object();  
Object. prototype = null;  
return result;  
}  
})();  
}

* Javascript objects are data type in the form of key-value pair.

**THE THIS KEYWORD:**

In a function definition, this refers to the "owner" of the function.

var person = {  
    firstName: "John",  
    lastName : "Doe",  
    id       : 5566,  
    fullName : function() {  
        return this.firstName + " " + this.lastName;  
    }  
};

**When a JavaScript variable is declared with the keyword "new", the variable is created as an object:**

var x = new String();        // Declares x as a String object  
var y = new Number();        // Declares y as a Number object  
var z = new Boolean();       // Declares z as a Boolean object

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

**JAVASCRIPT SCOPE**

Scope determines the accessibility (visibility) of variables. Each function creates a new scope.

In JavaScript there are two types of scope:

* Local scope: Variables declared within a function, become LOCAL to the function.
* Global scope: A variable declared outside a function, becomes GLOBAL.

If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable but Global variables are not created automatically in "Strict Mode".

Do NOT create global variables unless you intend to. Your global variables (or functions) can overwrite window variables (or functions). Any function, including the window object, can overwrite your global variables and functions.

**HTML EVENTS:**

An HTML event can be something the browser does, or something a user does. Here are some examples of HTML events:

* An HTML web page has finished loading
* An HTML input field was changed
* An HTML button was clicked

**COMMON HTML EVENTS:**

Here is a list of some common HTML events:

* onchange: An HTML element has been changed
* onclick: The user clicks an HTML element
* onmouseover: The user moves the mouse over an HTML element
* onmouseout: The user moves the mouse away from an HTML element
* onkeydown: The user pushes a keyboard key
* onload: The browser has finished loading the page

**JS STRING:**

* **Finding a String in a String :**

The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string and lastIndexOf() returns the index of the last occurrence. The indexOf() method accepts a second parameter as the starting position for the search

The search() method searches a string for a specified value and returns the position of the match

The two methods are NOT equal. These are the differences:

* + The search() method cannot take a second start position argument.
  + The indexOf() method cannot take powerful search values (regular expressions).
* There are 3 methods for extracting a part of a string:
  + slice(start, end)
  + substring(start, end)
  + substr(start, length)
* **slice()** extracts a part of a string and returns the extracted part in a new string.

var str = "Apple, Banana, Kiwi";  
var res = str.slice(7, 13);

res= Banana

If a parameter is negative, the position is counted from the end of the string.

var str = "Apple, Banana, Kiwi";  
var res = str.slice(-12, -6);

res= Banana

If you omit the second parameter, the method will slice out the rest of the string:

var res = str.slice(7);

* **SUBSTRING() METHOD:**

substring() is similar to slice(). The difference is that substring() cannot accept negative indexes.

* **SUBSTR () METHOD:**

substring() is similar to slice(). The difference is that substring() cannot accept negative indexes.

* 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. It returns a new string.

str = "Please visit Microsoft!";  
var n = str.replace("Microsoft", "W3Schools");

* **The concat() Method:**

concat() joins two or more strings:

var text1 = "Hello";  
var text2 = "World";  
var text3 = text1.concat(" ", text2);

* **String.trim()** removes whitespace from both sides of a string.
* The **charAt()** method returns the character at a specified index (position) in a string.
* The **charCodeAt()** method returns the unicode of the character at a specified index in a string.
* Accessing a String as an Array is Unsafe:

var str = "HELLO WORLD";  
str[0];

* A string can be converted to an array with the split() method.
* Number reference: (<https://www.w3schools.com/js/js_numbers.asp>)

**JAVASCRIPT FORM VALIDATION API:**

checkValidity() : Returns true if an input element contains valid data.

setCustomValidity(): Sets the validationMessage property of an input element.

 if (!inpObj.checkValidity()) {  
        document.getElementById("demo").innerHTML = inpObj.validationMessage;  
    }

Validity Properties has several properties:

valueMissing: Set to true, if an element (with a required attribute) has no value.

Valid: Set to true, if an element's value is valid.

customError: Set to true, if a custom validity message is set.

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

txt = "Value too large";

console.log(document.getElementById("id1").validity);

}

**JAVASCRIPT HOISTING**:( <https://www.w3schools.com/jS/js_hoisting.asp>)

**OBJECTS:**

**There are different ways to create new objects:**

* Define and create a single object, using an object literal.
* Define and create a single object, with the keyword new.
* Define an object constructor, and then create objects of the constructed type.

objectName["property"] // person["age"]

or

objectName[expression] // x = "age"; person[x]

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

Don’t create object using new keyword for simplicity, readability and execution speed, use the first one (the object literal method).

The **delete** keyword deletes a property from an object

**JavaScript Object Constructors:** A function designed to create new objects, is called an object constructor.

// Constructor function for Person objects

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

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// Create a Person object

var myFather = new Person("John", "Doe", 50, "blue");

The way to create an "object type", is to use an object constructor function. In the example above, function Person() is an object constructor function. Objects of the same type are created by calling the constructor function with the new keyword

**Prototype Properties:**

JavaScript objects inherit the properties of their prototype.

Date objects inherit from Date.prototype. Array objects inherit from Array.prototype. he Object.prototype is on the top of the prototype inheritance chain. Date objects, Array objects, and Person objects inherit from Object.prototype.

Sometimes you want to add new properties (or methods) to all existing objects of a given type or (or methods) to an object constructor. This can be done by prototype Property.

<script>

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

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

Person.prototype.nationality = "English";

Person.prototype.name = function() {  
    return this.firstName + " " + this.lastName;  
};

var myFather = new Person("John", "Doe", 50, "blue");

document.getElementById("demo").innerHTML =

"My father is " + myFather.nationality;

</script>

**FUNCTIONS:**

Self-Invoking Functions:

(function () {  
    var x = "Hello!!";      // I will invoke myself  
})();

The JavaScript call() Method:

The call() method is a predefined JavaScript method. It can be used to invoke (call) a method with an owner object as an argument (parameter).

var person = {  
    **fullName**: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
var person1 = {  
    firstName:"John",  
    lastName: "Doe",  
}  
var person2 = {  
    firstName:"Mary",  
    lastName: "Doe",  
}  
person.fullName.call(**person1**);  // Will return "John Doe"

**JavaScript Function Apply:**

With the **apply**() method, you can write a method that can be used on different objects.

var person = {  
    fullName: function(city, country) {  
        return this.firstName + " " + this.lastName + "," + city + "," + country;  
    }  
}  
var person1 = {  
    firstName:"John",  
    lastName: "Doe",  
}  
person.fullName.apply(person1, ["Oslo", "Norway"]);

The Difference Between call() and apply():

* The call() method takes arguments separately.
* The apply() method takes arguments as an array.

Since JavaScript arrays do not have a max() method, you can apply the Math.max() method instead.

Math.max.apply(null, [1,2,3]); // Will also return 3

**JavaScript Closures:** <https://www.w3schools.com/jS/js_function_closures.asp>

**JavaScript HTML DOM:**

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

The HTML DOM model is constructed as a tree of Objects. The HTML DOM is a standard object model and programming interface for HTML.



The W3C Document Object Model (DOM) is a platform that allows programs and scripts to dynamically access and update the content, structure, and style of a document.

**JAVASCRIPT EVENTS:**

In this example, the content of the <h1> element is changed when a user clicks on it:

<h1 onclick="this.innerHTML = 'Ooops!'">Click on this text!</h1>

**The onload and onunload Events:**

The onload and onunload events are triggered when the user enters or leaves the page. The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

<body onload="checkCookies()">cl

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

<script>

function checkCookies() {

var text = "";

if (navigator.cookieEnabled == true) {

text = "Cookies are enabled.";

} else {

text = "Cookies are not enabled.";

}

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

}

</script>

**Other Events:**

* onchange
* onmouseover
* onmouseout
* onmousedown
* onmouseup
* mousemove

**The addEventListener() method:**

The addEventListener() method attaches an event handler to the specified element. The addEventListener() method attaches an event handler to an element without overwriting existing event handlers. You can add event listeners to any DOM object not only HTML elements. i.e the window object. The addEventListener() method makes it easier to control how the event reacts to bubbling. You can easily remove an event listener by using the removeEventListener() method.

document.getElementById("myBtn").addEventListener("click", displayDate);

**Syntax**:

element.addEventListener(event, function, useCapture);

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

EVENTLISTENERS can be added to **window**:

<script>

window.addEventListener("resize", function(){

document.getElementById("demo").innerHTML = Math.random();

});

</script>

Passing Parameters:

element.addEventListener("click", function(){ myFunction(p1, p2); });

**Event Bubbling or Event Capturing?:**

Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In bubbling the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the <div> element's click event.

In capturing the outer most element's event is handled first and then the inner: the <div> element's click event will be handled first, then the <p> element's click event.

Ex. addEventListener(event, function, useCapture);

**The removeEventListener() method:**

element.removeEventListener("mousemove", myFunction);

**Note: to stop event propagation use** event.stopPropagation();

**Note: The addEventListener() and removeEventListener() methods are not supported in IE 8 and earlier versions and Opera 6.0 and earlier versions. However, for these specific browser versions, you can use the attachEvent() method to attach an event handlers to the element, and the detachEvent() method to remove it:**

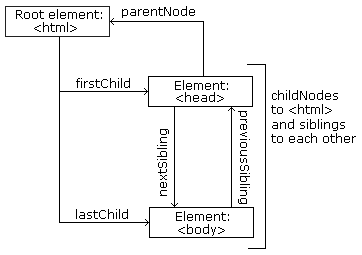
var x = document.getElementById("myBtn");  
if (x.addEventListener) {                    // For all major browsers, except IE 8 and earlier  
    x.addEventListener("click", myFunction);  
} else if (x.attachEvent) {                  // For IE 8 and earlier versions  
    x.attachEvent("onclick", myFunction);  
}

**DOM Document Object:**

The document object represents your web page. It is the owner of all other objects in your web page.

* <https://www.w3schools.com/jS/js_htmldom_document.asp> (FOR DOM SYNTAX)
* <https://www.w3schools.com/jS/js_htmldom_elements.asp>

**DOM NODE:**



**Navigating Between Nodes:**

* parentNode
* childNodes[nodenumber]
* firstChild
* lastChild
* nextSibling
* previousSibling

All three methods will give the same results:

var myTitle = document.getElementById("demo").innerHTML;

var myTitle = document.getElementById("demo").firstChild.nodeValue;

var myTitle = document.getElementById("demo").childNodes[0].nodeValue;

There are two special properties that allow access to the full document:

* document.body - The body of the document
* document.documentElement - The full document

Both will give the same result

The nodeName property specifies the name of a node:

* nodeName is read-only
* nodeName of an element node is the same as the tag name
* nodeName of an attribute node is the attribute name
* nodeName of a text node is always #text
* nodeName of the document node is always #document

**NOTE: Attribute node has been deprecated.**

**Creating New HTML Elements (Nodes):**

var para = document.createElement("p");  
var node = document.createTextNode("This is new.");  
para.appendChild(node);  
  
var element = document.getElementById("div1");  
element.appendChild(para);  
</script>

The appendChild() method in the previous example, appended the new element as the last child of the parent. If you don't want that you can use the insertBefore() method

<div id="div1">  
<p id="p1">This is a paragraph.</p>  
<p id="p2">This is another paragraph.</p>  
</div>  
  
<script>  
var para = document.createElement("p");  
var node = document.createTextNode("This is new.");  
para.appendChild(node);  
  
var element = document.getElementById("div1");  
var child = document.getElementById("p1");  
element.insertBefore(para, child);  
</script>

Removing a child:

parent.removeChild(child);

Replacing a child:   
parent.replaceChild(para, child);

**THE HTMLCOLLECTION OBJECT:**

The getElementsByTagName() method returns an HTMLCollection object. An HTMLCollection object is an array-like list (collection) of HTML elements.

var x = document.getElementsByTagName("p");

**THE HTML DOM NODELIST OBJECT:**

A NodeList object is a list (collection) of nodes extracted from a document. A NodeList object is almost the same as an HTMLCollection object.

var myNodelist = document.querySelectorAll("p");

A node list is not an array. A node list may look like an array, but it is not. You can loop through the node list and refer to its nodes like an array. However, you cannot use Array Methods, like valueOf(), push(), pop(), or join() on a node list.

**OBJECT**

**What is OOJS? (**<https://www.guru99.com/learn-object-oriented-javascript.html> )

The basic idea of OOP is that we use objects to model real world things that we want to represent inside our programs. JavaScript allows you to create objects that act like real life objects and Object can have many unique characteristics. You can create properties and methods to your objects to make programming easier. If your object is a student, it will have properties like first name, last name, id etc and methods like calculateRank, changeAddress etc. If your object is a home, it will have properties like a number of rooms, paint color, location etc

**Prototypes:**

Prototypes are the mechanism by which JavaScript objects inherit features from one another. An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is often referred to as a prototype chain.

Get prototypeproperty:

[Object.getPrototypeOf(obj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf), or via the deprecated [\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto) property

**Object.create():**

The Object.create() method creates a new object, using an existing object to provide the newly created object's \_\_proto\_\_ .

**const person = {**

**isHuman: false,**

**printIntroduction: function () {**

**console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);**

**}**

**};**

**const me = Object.create(person);**

**me.name = "Matthew"; // "name" is a property set on "me", but not on "person"**

**me.isHuman = true; // inherited properties can be overwritten**

**me.printIntroduction();**

**// expected output: "My name is Matthew. Am I human? true"**

**Object.assign():**

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object

Object.assign(*target*, ...*sources*)

**const object1 = {**

**a: 1,**

**b: 2,**

**c: 3**

**};**

**const object2 = Object.assign({c: 4, d: 5}, object1);**

**console.log(object2.c, object2.d);**

**// expected output: 3 5**

**Object.keys():**

The Object.keys() method returns an array of a given object's property names, in the same order as we get with a normal loop.

const object1 = {

a: 'somestring',

b: 42,

c: false

};

console.log(Object.keys(object1));

// expected output: Array ["a", "b", "c"]

**// Returns all properties as an array  
Object.getOwnPropertyNames(object)  
  
// Returns enumerable properties as an array  
Object.keys(object)**

**Object.keys('foo');**

**// ["0", "1", "2"] (ES2015 code)**

**How to create custom html elements ?**

<https://blog.teamtreehouse.com/create-custom-html-elements-2>

**What is ShadowDom, VirtualDOM?**

Shadow DOM: <https://www.youtube.com/watch?v=SDs4xmMcVS4>

-------------------------------------------------------------------------------------------------------------------------------

**Closure Example: (**<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>**)**

A closure is a function having access to the parent scope, even after the parent function has closed. variables can be made local (private) with closures.

For every closure we have three scopes:-

* Local Scope (Own scope)
* Outer Functions Scope
* Global Scope

function add() {

var counter = 0;

return function () {counter += 1; return counter;}

};

var f = add();

function myFunction(){

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

}

**This** example will work fine. Whenever u call that func counter value will increase.

function makeAdder(x) {

return function(y) {

return x + y;

};

}

var add5 = makeAdder(5);

var add10 = makeAdder(10);

console.log(add5(2)); // 7

console.log(add10(2)); // 12

**Emulating private methods with closures: (same url)**

var makeCounter = function() {

var privateCounter = 0;

function changeBy(val) {

privateCounter += val;

}

return {

increment: function() {

changeBy(1);

},

decrement: function() {

changeBy(-1);

},

value: function() {

return privateCounter;

}

}

};

var counter1 = makeCounter();

var counter2 = makeCounter();

alert(counter1.value()); /\* Alerts 0 \*/

counter1.increment();

counter1.increment();

alert(counter1.value()); /\* Alerts 2 \*/

counter1.decrement();

alert(counter1.value()); /\* Alerts 1 \*/

alert(counter2.value()); /\* Alerts 0 \*/

**Notice how each of the two counters, counter1 and counter2, maintains its independence from the other. Each closure references a different version of the privateCounter variable through its own closure.**

**Why let is better than var?**

let and const provide Block Scope variables (and constants) in JavaScript.

The let statement declares a block scope local variable, optionally initializing it to a value.

Redeclaring a JavaScript variable with var is allowed anywhere in a program but let doesn’t allow that.

**Scoping rules:**

if(true){

var aa= 12;

}

console.log(aa); // 12

if(true){

let aa= 12;

}

console.log(aa); // aa is not defined

Variable capturing:

var arr =[];

for(var i = 0; i< 5; i++){

arr.push(function () {console.log(i);});

}

console.log(arr);

arr[2](); //5

var arr =[];

for(let i = 0; i< 5; i++){

arr.push(function () {console.log(i);});

}

console.log(arr);

arr[2](); // 2

**Redeclare not possible with let:**

var x = 2;  
  
// Now x is 2  
   
var x = 3;  
  
// Now x is 3

**this keyword:** this is a keyword in JavaScript. Generally, 'this' keyword is a reference to the current object but inside a function or method it refers to the owner of the object.

**Difference between const and let?**

Variables defined with const behave like let variables, except they cannot be reassigned.

JavaScript const variables must be assigned a value when they are declared.

**Const:** [**https://www.w3schools.com/js/js\_const.asp**](https://www.w3schools.com/js/js_const.asp)

Not Real Constants:

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, we cannot change constant primitive values, but we can change the properties of constant objects.

**Difference between regular and arrow function**

<https://www.w3schools.com/js/js_arrow_function.asp>

The value of “this” keyword inside a regular function depends on how the function was called (the object that made the call)

The value of “this” keyword inside a arrow function depends on where the function was defined (the scope that defined the function)

The handling of this is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of this.

In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the this keyword always represents the object that defined the arrow function.

const obj = {

func1: function(){

console.log("func1", this);

}

,

func2: () => {

console.log("func2", this);

}

}

const obj2 = {

func3: obj.func2()

}

obj.func1();

obj.func2();

**Code Question:**

* let x;

if (x=2) {

let x = 2;

console.log(x);

// output: 2

}

console.log(x); // output: 2

* const a =12;  
  const obj1 = {  
   b: 4,  
   [a]: 20  
  }  
  console.log(obj1.a); // output: 12  
    
  because **a** is defined as dynamic property, so it evaluates first then obj1 value will be displayed.

**Destructuring:**

// const PI = Math.PI;

// const E = Math.E;

// const SQRT2 = Math.SQRT2;

const {PI, E, SQRT2} = Math;

// Somewhere in a React App

// const {Component, Fragment, useState} = require('react');

// useState();

// const circle = {

// label: 'circleX',

// radius: 2,

// };

// const circleArea = ({radius}, {precision = 2} = {}) =>

// (PI \* radius \* radius).toFixed(precision);

// console.log(

// circleArea(circle, { precision: 5 })

// );

// const [first, second,, forth] = [10, 20, 30, 40];

// const [value, setValue] = useState(initialValue);

**ASYNC/AWAIT:**

// const fetchData = () => {

// fetch('https://api.github.com').then(resp => {

// resp.json().then(data => {

// console.log(data);

// });

// });

// };

const fetchData = async () => {

const resp = await fetch('https://api.github.com');

const data = await resp.json();

console.log(data);

};

fetchData();

**JavaScript array reduce method:**

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce>

**Array.from():**

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from>

console.log(Array.from('foo'));

// expected output: Array ["f", "o", "o"]

console.log(Array.from([1, 2, 3], x => x + x));

// expected output: Array [2, 4, 6]

* **yml file we are using in azure pipeline instead of classic editor because it is easy to migrate**
* **Learn more about new features of Ecmascript.**

TypeScript 3.8 supports es2020 as an option for module and target. This will preserve newer ECMAScript 2020 features like optional chaining, nullish coalescing, export \* as ns, and dynamic import(...) syntax. It also means bigint literals now have a stable target below esnext.

* <https://www.geeksforgeeks.org/what-is-export-default-in-javascript/>
* Modules:

<https://javascript.info/modules-intro>

As our application grows bigger, we want to split it into multiple files, so called “modules”. A module may contain a class or a library of functions for a specific purpose.

For a long time, JavaScript existed without a language-level module syntax. That wasn’t a problem, because initially scripts were small and simple, so there was no need.

But eventually scripts became more and more complex, so the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.

To name some (for historical reasons):

AMD – one of the most ancient module systems, initially implemented by the library require.js.

CommonJS – the module system created for Node.js server.

UMD – one more module system, suggested as a universal one, compatible with AMD and CommonJS.

* Sorting an Array in Random Order

var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return 0.5 - Math.random()});

<https://www.w3schools.com/js/js_array_sort.asp>

* + See the Fisher Yates Method
  + Sorting complex arrays
* **Date Input - Parsing Dates**

<https://www.w3schools.com/js/js_date_formats.asp>

If you have a valid date string, you can use the Date.parse() method to convert it to milliseconds.

var msec = Date.parse("March 21, 2012");  
var d = new Date(msec);  
document.getElementById("demo").innerHTML = d;

* **Promise:**

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

let myPromise = new Promise(function(myResolve, myReject) {  
  let req = new XMLHttpRequest();  
  req.open('GET', "mycar.htm");  
  req.onload = function() {  
    if (req.status == 200) {  
      myResolve(req.response);  
    } else {  
      myReject("File not Found");  
    }  
  };  
  req.send();  
});  
  
myPromise.then(  
  function(value) {myDisplayer(value);},  
  function(error) {myDisplayer(error);}  
);

* Async/await:

It is introduced in Ecma 2017, These features basically act as syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards. It makes them behave like synchronous code.

* **Cookies:**

<https://www.w3schools.com/js/js_cookies.asp>

**IMPORTANT INTERVIEW QUESTIONS**

* <https://www.codementor.io/nihantanu/21-essential-javascript-tech-interview-practice-questions-answers-du107p62z>
* <https://www.toptal.com/javascript/interview-questions>
* <https://www.c-sharpcorner.com/article/top-most-50-javascript-interview-questions-and-answers/> (Qno. 16)
* <https://www.code-sample.com/2015/04/javascript-interview-questions-answers.html>