**JAVASCRIPT**

**Agenda:**

**- Process map**

**- Global/Local dictionaries**

**- Adding removing variables dynamically**

**- 5 data types (+1)**

**- Object internals**

**- 5 ways of creating objects**

**- Function internals**

**- Difference between \_\_proto\_\_ and prototype**

**- Prototype chaining**

**- Internal differences between function and lambda**

**- Inheritance using functions and classes**

**- Internals of array and array methods**

**- Closures**

**- Lack of async in JavaScript**

**- setTimeout, setInterval**

**- Promise internals**

**- Understanding async and await**

**Interpreted language : JavaScript was an interpreted language in the 1980’s because its interprets or executes each line of code and gives the output for each lines. At first, suppose there is a function lke “console.log” or “return” and to execute this, so what does the interpreter do?**

**We first tell the interpreter to run the functions, so it first takes the code, and some machine code is predefined for that In the interpreter, and the interpreter then returns it. After it return, the next function “return” is send…so it is slow because, the interpreter takes one function at a time and checks, in the mean time the program is in hold, until the interpreter returns some result, it won’t go to the next function. But when it comes to debugging, it’s awesome because it checks line by line, if error found, then the program stops. So memory saved as well as execution time saved and code crash occurred.**

**Whereas compiled language turns every code into byte code, and goes on compiling, in the middle if suppose there is an error in the code, the compiler won’t stops compiling, if the system gets crashed or the wrong codes comsumes memory, it doesn’t matter to the compiler. So compiler is actually very fast in compiling but when comes to debugging, it is a problem.**

**But now, JIT Compilation has given us both advantage of compiler and interpreter.JS has Ignition interpreter and Turbo fan compiler. So suppose there is a code like:**

**function myFunction(){**

**console.log(“helo”);**

**return 1+1;**

**}**

**}**

**function myAnotherFunction(){**

**console.log(“hiiii”);**

**return 2;**

**}**

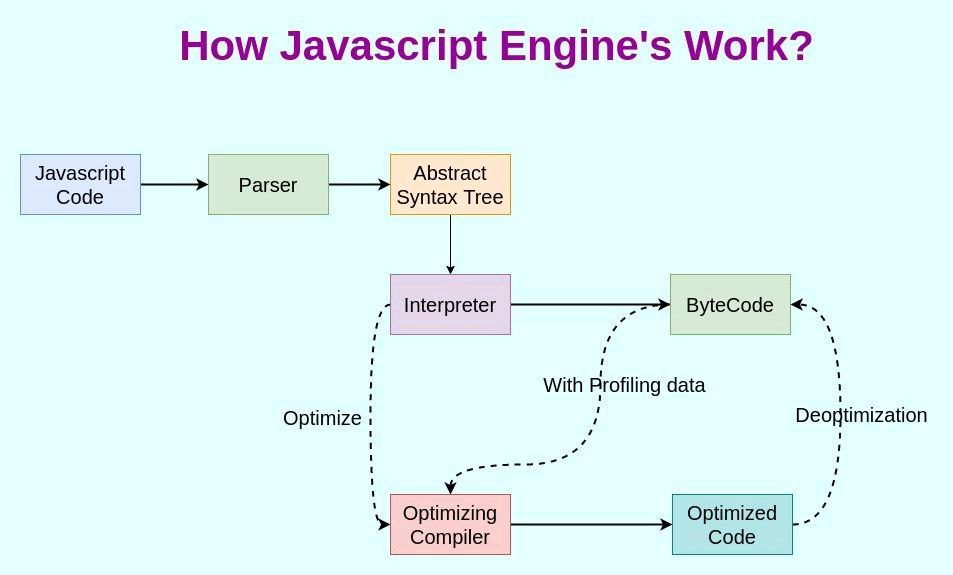
**}**

**Now,** the interpreter first starts executing the code and comes to the first function and its contents, but before that, that Turbofan compiler of JIT will turn the whole content into byte code, and compile it,no error, then interpreter moves to next function;

But now suppose there is an error in the second function which compiler found, then our interpreter tells it to stop and returns the error, if the interpreter was not there then the compiler would have gone on executing the code instead of the error and thereby leading to crash of systems.

So, JIT saves everything, it is true that the compilation is not as first as JAVA compilers, but the speed is still much fast atleast from the 1980’s JS interpreter. So, now with Java In Time Compilation we can have easy debugging as well as very fast compilation.

**Compiled Language: JAVA is a compiled language because it compiles the whole code first and then gives the output if it doesn’t finds any error. And its converts first to byte code, and then to machine generated code, so it’s bit slower then JS but more efficient then JS just because it only executes when the code is totally error free. In java the compiler after compiling the code optimizes the code again before executing for first execution.**

**JS ARCHITECTURE: CODE => Syntax parser while parse the code into JSON tree format called AST(Abstract Syntax Tree) => then interpretation and compilation occurs simultaneously with JIT Compilation => then execution is carried on.**

**ALSO READ ABOUT: A)Mark & Sweep B) In Lining C) Copy Elision D) Inline Cashing**

## [Comparison with ||](https://javascript.info/nullish-coalescing-operator" \l "comparison-with)

The OR || operator can be used in the same way as ??, as it was described in the [previous chapter](https://javascript.info/logical-operators#or-finds-the-first-truthy-value).

For example, in the code above we could replace ?? with || and still get the same result:

let firstName = null;

let lastName = null;

let nickName = "Supercoder";

// shows the first truthy value:

alert(firstName || lastName || nickName || "Anonymous"); // Supercoder

Historically, the OR || operator was there first. It exists since the beginning of JavaScript, so developers were using it for such purposes for a long time.

On the other hand, the nullish coalescing operator ?? was added to JavaScript only recently, and the reason for that was that people weren’t quite happy with ||.

The important difference between them is that:

* || returns the first truthy value.
* ?? returns the first defined value.

In other words, || doesn’t distinguish between false, 0, an empty string "" and null/undefined. They are all the same – falsy values. If any of these is the first argument of ||, then we’ll get the second argument as the result.

In practice though, we may want to use default value only when the variable is null/undefined. That is, when the value is really unknown/not set.

For example, consider this:

let height = 0;

alert(height || 100); // 100

alert(height ?? 100); // 0

* The height || 100 checks height for being a falsy value, and it’s 0, falsy indeed.
  + so the result of || is the second argument, 100.
* The height ?? 100 checks height for being null/undefined, and it’s not,
  + so the result is height “as is”, that is 0.

### [Exponentiation \*\*](https://javascript.info/operators" \l "exponentiation)

The exponentiation operator a \*\* b raises a to the power of b.

In school maths, we write that as ab.

For instance:

alert( 2 \*\* 2 ); // 2² = 4

alert( 2 \*\* 3 ); // 2³ = 8

alert( 2 \*\* 4 ); // 2⁴ = 16

Just like in maths, the exponentiation operator is defined for non-integer numbers as well.

For example, a square root is an exponentiation by ½:

alert( 4 \*\* (1/2) ); // 2 (power of 1/2 is the same as a square root)

alert( 8 \*\* (1/3) ); // 2 (power of 1/3 is the same as a cubic root)

## [String concatenation with binary +](https://javascript.info/operators" \l "string-concatenation-with-binary)

Let’s meet features of JavaScript operators that are beyond school arithmetics.

Usually, the plus operator + sums numbers.

But, if the binary + is applied to strings, it merges (concatenates) them:

let s = "my" + "string";

alert(s); // mystring

Note that if any of the operands is a string, then the other one is converted to a string too.

For example:

alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"

See, it doesn’t matter whether the first operand is a string or the second one.

Here’s a more complex example:

alert(2 + 2 + '1' ); // "41" and not "221"

Here, operators work one after another. The first + sums two numbers, so it returns 4, then the next + adds the string 1 to it, so it’s like 4 + '1' = '41'.

alert('1' + 2 + 2); // "122" and not "14"

Here, the first operand is a string, the compiler treats the other two operands as strings too. The 2 gets concatenated to '1', so it’s like '1' + 2 = "12" and "12" + 2 = "122".

The binary + is the only operator that supports strings in such a way. Other arithmetic operators work only with numbers and always convert their operands to numbers.

Here’s the demo for subtraction and division:

alert( 6 - '2' ); // 4, converts '2' to a number

alert( '6' / '2' ); // 3, converts both operands to numbers

## [Numeric conversion, unary +](https://javascript.info/operators" \l "numeric-conversion-unary)

The plus + exists in two forms: the binary form that we used above and the unary form.

The unary plus or, in other words, the plus operator + applied to a single value, doesn’t do anything to numbers. But if the operand is not a number, the unary plus converts it into a number.

For example:

// No effect on numbers

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Converts non-numbers

alert( +true ); // 1

alert( +"" ); // 0

It actually does the same thing as Number(...), but is shorter.

The need to convert strings to numbers arises very often. For example, if we are getting values from HTML form fields, they are usually strings. What if we want to sum them?

The binary plus would add them as strings:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23", the binary plus concatenates strings

If we want to treat them as numbers, we need to convert and then sum them:

let apples = "2";

let oranges = "3";

// both values converted to numbers before the binary plus

alert( +apples + +oranges ); // 5

// the longer variant

// alert( Number(apples) + Number(oranges) ); // 5

From a mathematician’s standpoint, the abundance of pluses may seem strange. But from a programmer’s standpoint, there’s nothing special: unary pluses are applied first, they convert strings to numbers, and then the binary plus sums them up.

Why are unary pluses applied to values before the binary ones? As we’re going to see, that’s because of their higher precedence.

**OBJECTS IN-DEPTH**

# Object references and copying

One of the fundamental differences of objects versus primitives is that objects are stored and copied “by reference”, whereas primitive values: strings, numbers, booleans, etc – are always copied “as a whole value”.

That’s easy to understand if we look a bit under the hood of what happens when we copy a value.

Let’s start with a primitive, such as a string.

Here we put a copy of message into phrase:

let message = "Hello!";

let phrase = message;

As a result we have two independent variables, each one storing the string "Hello!".

Quite an obvious result, right?

Objects are not like that.

**A variable assigned to an object stores not the object itself, but its “address in memory” – in other words “a reference” to it.**

Let’s look at an example of such a variable:

let user = {

name: "John"

};

And here’s how it’s actually stored in memory:

The object is stored somewhere in memory (at the right of the picture), while the user variable (at the left) has a “reference” to it.

We may think of an object variable, such as user, as like a sheet of paper with the address of the object on it.

When we perform actions with the object, e.g. take a property user.name, the JavaScript engine looks at what’s at that address and performs the operation on the actual object.

Now here’s why it’s important.

**When an object variable is copied, the reference is copied, but the object itself is not duplicated.**

For instance:

let user = { name: "John" };

let admin = user; // copy the reference

Now we have two variables, each storing a reference to the same object:

As you can see, there’s still one object, but now with two variables that reference it.

We can use either variable to access the object and modify its contents:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // changed by the "admin" reference

alert(user.name); // 'Pete', changes are seen from the "user" reference

It’s as if we had a cabinet with two keys and used one of them (admin) to get into it and make changes. Then, if we later use another key (user), we are still opening the same cabinet and can access the changed contents.

**TO KNOW MORE DETAILS ABOUT OBJECTS:**

**FOLLOW :** [**https://javascript.info/object-copy**](https://javascript.info/object-copy)

### **[Method shorthand](https://javascript.info/object-methods" \l "method-shorthand)(two objects, same meaning)**

There exists a shorter syntax for methods in an object literal:

// these objects do the same

user = {

sayHi: function() {

alert("Hello");

}

};

// method shorthand looks better, right?

user = {

sayHi() { // same as "sayHi: function(){...}"

alert("Hello");

}

};

**STRINGS**:

The square brackets are a modern way of getting a character, while charAt exists mostly for historical reasons.

The only difference between them is that if no character is found, [] returns undefined, and charAt returns an empty string:

let str = `Hello`;

alert( str[1000] ); // undefined

alert( str.charAt(1000) ); // '' (an empty string)

# Printing each character in the string with For of

for (let char of "Hello") {

alert(char); // H,e,l,l,o (char becomes "H", then "e", then "l" etc)

}

# INCLUDES METHOD IN JS

The more modern method [str.includes(substr, pos)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes) returns true/false depending on whether str contains substr within.

It’s the right choice if we need to test for the match, but don’t need its position:

alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false

**str.substr(start [, length])**

Returns the part of the string from start, with the given length.

In contrast with the previous methods, this one allows us to specify the length instead of the ending position:

let str = "stringify";

alert( str.substr(2, 4) ); // 'ring', from the 2nd position get 4 characters

The first argument may be negative, to count from the end:

## **[Performance](https://javascript.info/array" \l "performance)**

Methods push/pop run fast, while shift/unshift are slow.

Why is it faster to work with the end of an array than with its beginning? Let’s see what happens during the execution:

fruits.shift(); // take 1 element from the start

It’s not enough to take and remove the element with the number 0. Other elements need to be renumbered as well.

The shift operation must do 3 things:

1. Remove the element with the index 0.
2. Move all elements to the left, renumber them from the index 1 to 0, from 2 to 1 and so on.
3. Update the length property.

**The more elements in the array, the more time to move them, more in-memory operations.**

The similar thing happens with unshift: to add an element to the beginning of the array, we need first to move existing elements to the right, increasing their indexes.

And what’s with push/pop? They do not need to move anything. To extract an element from the end, the pop method cleans the index and shortens length.

The actions for the pop operation:

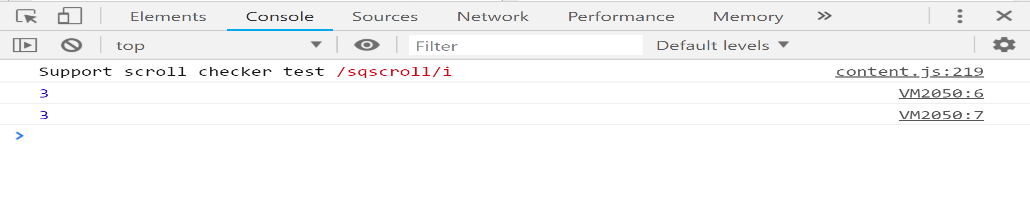
fruits.pop(); // take 1 element from the end

**The pop method does not need to move anything, because other elements keep their indexes. That’s why it’s blazingly fast.**

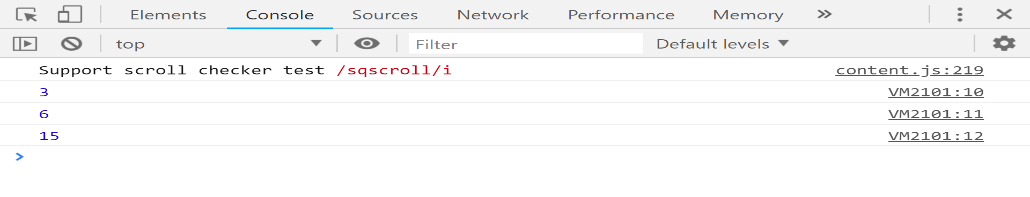
The similar thing with the push method.

**Rest parameter** is an improved way to handle function parameter, allowing us to more easily handle various input as parameters in a function. The rest parameter syntax allows us to represent an indefinite number of arguments as an array. With the help of a rest parameter a function can be called with any number of arguments, no matter how it was defined. Rest parameter is added in ES2015 or ES6 which improved the ability to handle parameter.

|  |
| --- |
| // Without rest parameter  function fun(a, b){      return a + b;  }  console.log(fun(1, 2)); // 3  console.log(fun(1, 2, 3, 4, 5)); // 3 |

**Output:**  
  
In the above code, no error will be thrown even when we are passing arguments more than the parameters, but only the first two arguments will be evaluated. It’s different in the case with rest parameter. With the use of the rest parameter, we can gather any number of arguments into an array and do what we want with them.  
Javascript code demonstrating addition of numbers using rest parameter.

|  |
| --- |
| // es6 rest parameter  function fun(...input){      let sum = 0;      for(let i of input){          sum+=i;      }      return sum;  }  console.log(fun(1,2)); //3  console.log(fun(1,2,3)); //6  console.log(fun(1,2,3,4,5)); //15 |

**Output:**  
  
We were able to get the sum of all the elements that we enter in the argument when we call the fun() function. We get the sum of all the elements in the array by making use of the for..of loop which is used to traverse the iterable elements inside an array.

**Note:** The rest parameter have to be the last argument, as its job is to collect all the remaining arguments into an array. So having a function definition like the code below doesn’t make any sense and will throw an error.

**NOTE1:**

The main difference is that **document**. **write**() inserts text into the **document** (the webpage in the browser window); **console**. ... **log**() is a method of the **JavaScript Console**, a special object incuded in browsers for developers to use to debug and test their code in an interactive terminal. **log**() method to print to **console JavaScript**. The **JavaScript console log** function is mainly used for code debugging as it makes the **JavaScript** print the output to the **console**. To open the browser **console**, right-click on the page and select Inspect, and then click **Console.**

**CLOSURE:**

*Closure means that an inner function always has access to the vars and parameters of its outer function, even after the outer function has returned.*

***QUESTIONS:***

1. what is a closure?  
   2. callbacks in JS  
   3. settimeout  
   4. how can I write a set timeout for 5 minutes?

5. What would be the output of the following code snippet?  
function foo(){  
var a = 100  
console.log(this.a)  
bar()  
}  
  
function bar(){  
var a = 200  
console.log(this.a)  
}  
  
function doFoo(fn){  
fn()  
}  
var obj = {  
a: 10,  
foo: foo  
}  
var a = 'I am global'  
doFoo(obj.foo)  
6. this operator  
7. variable hoisting :this concept tells about the “var” which is hoisted always above, and then its initialization or function are placed when compiled  
8. function declaration  
9. function expression  
10. function hoisting

12. Find the output    
function fn1(){   
var age = 12;   
function fn2(){   
var age = 10;   
var age1 = 2;   
}  
console.log(age);   
console.log(age1);   
} fn1();

13. Anonymous Function: this is a function where no function name is there, it is stored in a variable, and the variable is used to call he function. **Anonymous** functions can be used as an argument to other functions or as an immediately invoked **function** execution.

Example: var obj = function(){

Document.write(“hii deba”);

obj(); //calling a anonymous function

//passing Anonymous function as arguments

Function func1(myfunc){// so myfunc gets the anonymous function as an argument

return myfunc();//here we call the anonymous function and return the value in it to parent function

}

Document.write(func1(function(){ return “rockingdeba”;}));

//Returning anonymous function as arguments //

Function func2(){

Return function(){ return a+b;}//here the whole anonymous function is returned to the calling function, which is later called through another variable.

}

Var obj2 = Func2(20);

Document.write(obj2());

14.IIFE : Immediately Invoking function expression, which is a self executing function and contains two parts, 1st one is the anonymous function enclosed with the lexical scope enclosed within the grouping operator().

15.difference between null and undefined

16.Factory Function: it is a function which returns objects without use of new keyword.

Function mobile(){

Return {

Color:”red”,

Height: function(){ return “price is 3000”;}

};

Var obj = Mobile();

Document.write(obj.color+” “+obj.height());//to call the anonymous function in height, we //se braces with obj.height.

Please find below the links which may help you to improve your understanding by practising snippet based questions  
https://www.interviewcake.com/javascript-interview-questions  
https://dmitripavlutin.com/simple-but-tricky-javascript-interview-questions/  
https://www.codementor.io/@nihantanu/21-essential-javascript-tech-interview-practice-questions-answers-du107p62z  
https://www.toptal.com/react/interview-questions  
https://www.codementor.io/blog/5-essential-reactjs-interview-questions-du1084ym1  
<https://hackr.io/blog/react-interview-questions>

Note:

Normal function:

Function show(a,b)

{

return a+b;

}

Show();

Arrow function is the shorter version of anonymous function.

Var myfund = (a,b) => {return a+b;};//one way to declare

Or can be written as

var myfund = (a,b) => a+b;// 2nd way to declare

myfund(10,30);

# Immediately Invoked Function Expression - IIFE

Immediately Invoked Function Expression (IIFE) is one of the most popular design patterns in JavaScript. It pronounces like iify. IIFE has been used since long by JavaScript community but it had misleading term "self-executing anonymous function". [Ben Alman](http://benalman.com/) gave it appropriate name "Immediately Invoked Function Expression". As you know that a function in JavaScript creates the local scope. So, you can define variables and function inside a function which cannot be access outside of that function. However, sometime you accidently pollute the global variables or functions by unknowingly giving same name to variables & functions as global variable & function names. For example, there are multiple .js files in your application written by multiple developers over a period of time. Single JavaScript file includes many functions and so these multiple .js files will result in large number of functions. There is a good chance of having same name of function exists in different .js files written by multiple developer and if these files included in a single web page then it will pollute the global scope by having two or more function or variables with the same name. Consider following example of two different JavaScript file included in single page.

**Advantages of IIFE:**

1. Do not create unnecessary global variables and functions
2. Functions and variables defined in IIFE do not conflict with other functions & variables even if they have same name.
3. Suppose there are two functions in javascript with the same name “display”.Now if we call with ‘display’ function, compiler will get confused.

Then the solution is “IIFE”. IEFE solves this problem by having its own scope and restricting functions and variables to become global. The functions and variables declare inside IIFE will not pollute global scope even they have same name as global variables & functions. So let's see what is an IIFE is.

QUESTION:Difference between “undefined” and “undefined error”?ans. var a; document.write(a); document.write(b);

Declaring objects is of two types:

1. OBJECT CONSTRUCTOR=> Var obj = new object();

OR //Both are same

1. OBJECT LITERAL=> var obj = { };

**FACTORY FUNCTION:**

When a function returns a object, we call it a factory function. It can use object instance without using new keyword or classes.

Ex. Function mobile()

{

Return {

Model : “Galaxy”,

Price : function() { return (“Price: Rs. 3000”); }

};

}

Var Samsung = mobile(); //so here we call the previous function to get the same properties of the previous object in this “Samsung” variable

**JS => javascript:void(0) =>** This is used for href=”#”, because when we declare an <a> tag, and has href=#, then if we click that tag, we just refresh or reload the whole page, which is actually unnecessary because no link is provided in href. So, to remove the unnecessary effect of href we declare the above code.

<a *href*="javascript:void(0);" *onclick*="addMore(<?=date('Y')?>)" *class*="btn btn-primary btn-sm pull-right"><i *class*="fa fa-plus"></i> Add More </a>

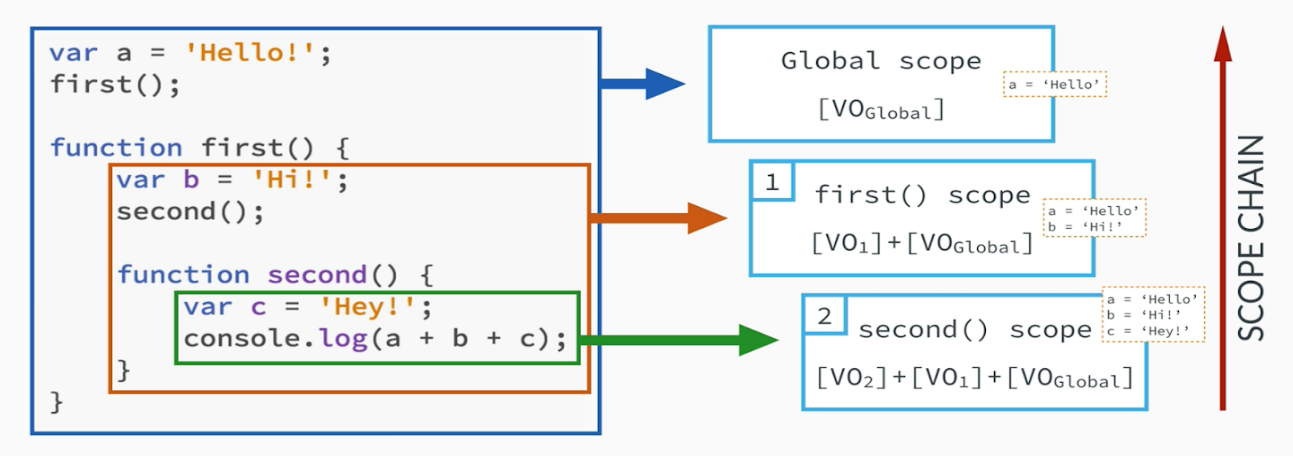
So here we have a onclick event which will perform the task, but when we click the button, we don’t need the whole page to refresh because of the present of href. So to stop the effect we use javascript:void(0)

Sometimes, you will decide to call some JavaScript from inside a link. Normally, when you click a link, the browser loads a brand new page or refreshes the same page (depending on the URL specified).  
But you most likely don’t desire this to happen if you have hooked up some JavaScript thereto link.

To prevent the page from refreshing, you could use void(0).

***LEXICAL SCOPE:***

A **lexical scope in JavaScript** means that a variable defined outside a function can be accessible inside another function defined after the variable declaration. But the opposite is not true; the variables defined inside a function will not be accessible outside that function.



**window** is just the global object that houses several properties, one of them is location. location also has properties, one of them is href. location.href is just window.location.href.

**location.href** property returns the entire URL of the current page.

**window.location** property represents the currect location of the window object, if you change this you will get redirected.

**Create a new number method that returns a number's half value:**

Number.prototype.myMethod = function() {  
  return this.valueOf() / 2;  
};

**Use the new method on a number:**

var n = 55;  
var x = n.myMethod();

## **Definition and Usage**

For normal objects: var Mobile = function(model\_no){

This.model = model\_no;

};

Var samsung = new Mobile(‘Galaxy’);

Var nokia = new Mobile(‘3310’);

Samsung.color= ‘white’;//here we create a new property of the samsung object

Document.write(samsung.color);//now printing the property

//lets check if the property is also available for nokia object even, so what did we do,we called the property below

Document.write(nokia.color);//it doesnot show any value, so for every object if we need a same property like colour, then we have to define every time, which Is a problem and time issue

//to solve it, we used prototype objects, which are a property of the original object, and there we assign the prototype object with the common properties and make other objects to inherit it, thus making it a global property and every objects can access it now. HaPPy EndinG.

The prototype constructor allows you to add new properties and methods to JavaScript numbers.

When constructing a property, ALL numbers will be given the property, and its value, as default.

When constructing a method, ALL numbers will have this method available.

**Note:** Number.prototype does not refer to a single number object, but to the Number() object itself.

**Note:** Prototype is a global object constructor which is available for all JavaScript objects.

More Brief Description about prototypes:

First of all when we first create any functions also often called function objects(ex. Function foo() {} ) then two objects are created when we call that function by {::foo() },

1st object is “ function foo() “,

2nd object created for that function ‘foo’ is termed as prototype object, which is accessed by writing foo.prototype

So this foo function object can contain many properties, when function Is called a property is created (ex. Prototype only, which is pointing to prototype objects.so to acces the protoptype object we write “function\_name.property =>foo.prototype”, here name of the property is only ‘prototype’ for pointing to prototype objects.

Function foo(){} =>a function is created which will be called by the constructor

foo => returns function foo() in console

foo() =>the function is called here in console

var newfooobj = new foo() =>here we create a new object containing the function new foo() and is a constructor for the ‘foo’ function object

newfooobj =>this contains the \_proto\_ property which will point to the prototype object here (‘foo.prototype’).

foo.prototype.test = “hii deba”; or newfooobj.\_\_proto\_\_.test=”hii deba” => this is how we created our own property named ‘test’ and set the values in it

we can take a new object and use the new keyword with the name of a previous function like foo and call its constructor,thus the new object will inherit all the properties of the foo constructor. Prototype makes inheritance possible

There are two interrelated concepts with **prototype** in JavaScript:

1. First, every JavaScript function has a **prototype property** (this property is empty by default), and you attach properties and methods on this prototype property when you want to implement inheritance. This prototype property is not enumerable; that is, it isn’t accessible in a for/in loop. But Firefox and most versions of Safari and Chrome have a \_\_proto\_\_ “pseudo” property (an alternative syntax) that allows you to access an object’s prototype property. You will likely never use this \_\_proto\_\_ pseudo property, but you should know that it exists and it is simply a way to access an object’s prototype property in some browsers.

The prototype property is used primarily for inheritance; you add methods and properties on a function’s prototype property to make those methods and properties available to instances of that function.

function PrintStuff (myDocuments) {

this.documents = myDocuments;

}

// We add the print () method to PrintStuff prototype property so that other instances (objects) can inherit it:

PrintStuff.prototype.print = function () {

console.log(this.documents);

}

// Create a new object with the PrintStuff () constructor, thus allowing this new object to inherit PrintStuff's properties and methods.

var newObj = new PrintStuff ("I am a new Object and I can print.");

// newObj inherited all the properties and methods, including the print method, from the PrintStuff function. Now newObj can call print directly, even though we never created a print () method on it.

newObj.print (); //I am a new Object and I can print.

The second concept with prototype in JavaScript is the **prototype attribute**. Think of the prototype attribute as a characteristic of the object; this characteristic tells us the object’s “parent”. In simple terms: An object’s prototype attribute points to the object’s “parent”—the object it inherited its properties from. The prototype attribute is normally referred to as the *prototype object*, and it is set automatically when you create a new object.To expound on this: Every object inherits properties from some other object, and it is this other object that is the object’s prototype attribute or “parent.” (You can think of the *prototype attribute* as the lineage or the parent). In the example code above, *newObj*‘s prototype is PrintStuff.prototype.

Note: All objects have attributes just like object properties have attributes. And the object attributes are *prototype, class,* and *extensible* attributes. It is this prototype attribute that we are discussing in this second example.

1. **Prototype Property: Prototype-based Inheritance**  
   Prototype is important in JavaScript because JavaScript does not have classical inheritance based on Classes (as most object oriented languages do), and therefore all inheritance in JavaScript is made possible through the prototype property. JavaScript has a prototype-based inheritance mechanism.Inheritance is a programming paradigm where objects (or Classes in some languages) can inherit properties and methods from other objects (or Classes). In JavaScript, you implement inheritance with the prototype property. For example, you can create a Fruit function (an object, since all functions in JavaScript are objects) and add properties and methods on the Fruit prototype property, and all instances of the Fruit function will inherit all the Fruit’s properties and methods.

Demonstration of Inheritance in JavaScript:

function Plant () {

this.country = "Mexico";

this.isOrganic = true;

}

// Add the showNameAndColor method to the Plant prototype property

Plant.prototype.showNameAndColor = function () {

console.log("I am a " + this.name + " and my color is " + this.color);

}

// Add the amIOrganic method to the Plant prototype property

Plant.prototype.amIOrganic = function () {

if (this.isOrganic)

console.log("I am organic, Baby!");

}

function Fruit (fruitName, fruitColor) {

this.name = fruitName;

this.color = fruitColor;

}

// Set the Fruit's prototype to Plant's constructor, thus inheriting all of Plant.prototype methods and properties.

Fruit.prototype = new Plant ();

// Creates a new object, aBanana, with the Fruit constructor

var aBanana = new Fruit ("Banana", "Yellow");

// Here, aBanana uses the name property from the aBanana object prototype, which is Fruit.prototype:

console.log(aBanana.name); // Banana

// Uses the showNameAndColor method from the Fruit object prototype, which is Plant.prototype. The aBanana object inherits all the properties and methods from both the Plant and Fruit functions.

console.log(aBanana.showNameAndColor()); // I am a Banana and my color is yellow.

Note that the showNameAndColor method was inherited by the aBanana object even though it was defined all the way up the prototype chain on the Plant.prototype object.

Indeed, any object that uses the Fruit () constructor will inherit all the Fruit.prototype properties and methods and all the properties and methods from the Fruit’s prototype, which is Plant.prototype. This is **the principal manner in which inheritance is implemented in JavaScript** and the integral role the prototype chain has in the process.

For more in-depth coverage on Objective Oriented Programming in JavaScript, get Nicholas Zakas’s [Principles of Object-Oriented Programming in JavaScript](https://leanpub.com/oopinjavascript) (it is only $14.99).

1. **Prototype Attribute: Accessing Properties on Objects**  
   Prototype is also important for accessing properties and methods of objects. The **prototype attribute** (or prototype object) of any object is the “parent” object where the inherited properties were originally defined.This is loosely analogous to the way you might inherit your surname from your father—he is your “prototype parent.” If we wanted to find out where your surname came from, we would first check to see if you created it yourself; if not, the search will move to your prototype parent to see if you inherited it from him. If it was not created by him, the search continues to his father (your father’s prototype parent).

Similarly, if you want to access a property of an object, the search for the property begins directly on the object. If the JS runtime can’t find the property there, it then looks for the property on the object’s prototype—the object it inherited its properties from.  
If the property is not found on the object’s prototype, the search for the property then moves to prototype of the object’s prototype (the father of the object’s father—the grandfather). And this continues until there is no more prototype (no more great-grand father; no more lineage to follow). **This in essence is the prototype chain:** the chain from an object’s prototype to its prototype’s prototype and onwards. And JavaScript uses this prototype chain to look for properties and methods of an object.  
If the property does not exist on any of the object’s prototype in its prototype chain, then the property does not exist and undefined is returned.

More about PROTOTYPES

1. <https://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/#:~:text=Prototype%20is%20important%20in%20JavaScript,a%20prototype%2Dbased%20inheritance%20mechanism>.
2. <https://timkadlec.com/2008/01/using-prototypes-in-javascript/#:~:text=Prototypes%20allow%20you%20to%20easily,created%20in%20the%20previous%20post>.
3. <https://www.youtube.com/watch?v=qUBcYdWlkz0&list=PLqq-6Pq4lTTaflXUL0v3TSm86nodn0c_u&index=14>

A **constructor** is a function used for initializing new objects with its various properties or methods, and you use the new keyword to call the constructor.

**PROBLEM OF prototype:**

*// need to understand*

function Parent() { */\* ... \*/* }

function CreatedConstructor() {

   Parent.call(this)

}

CreatedConstructor.prototype = Object.create(Parent.prototype)

*// CreatedConstructor.prototype.constructor = CreatedConstructor // sets the*

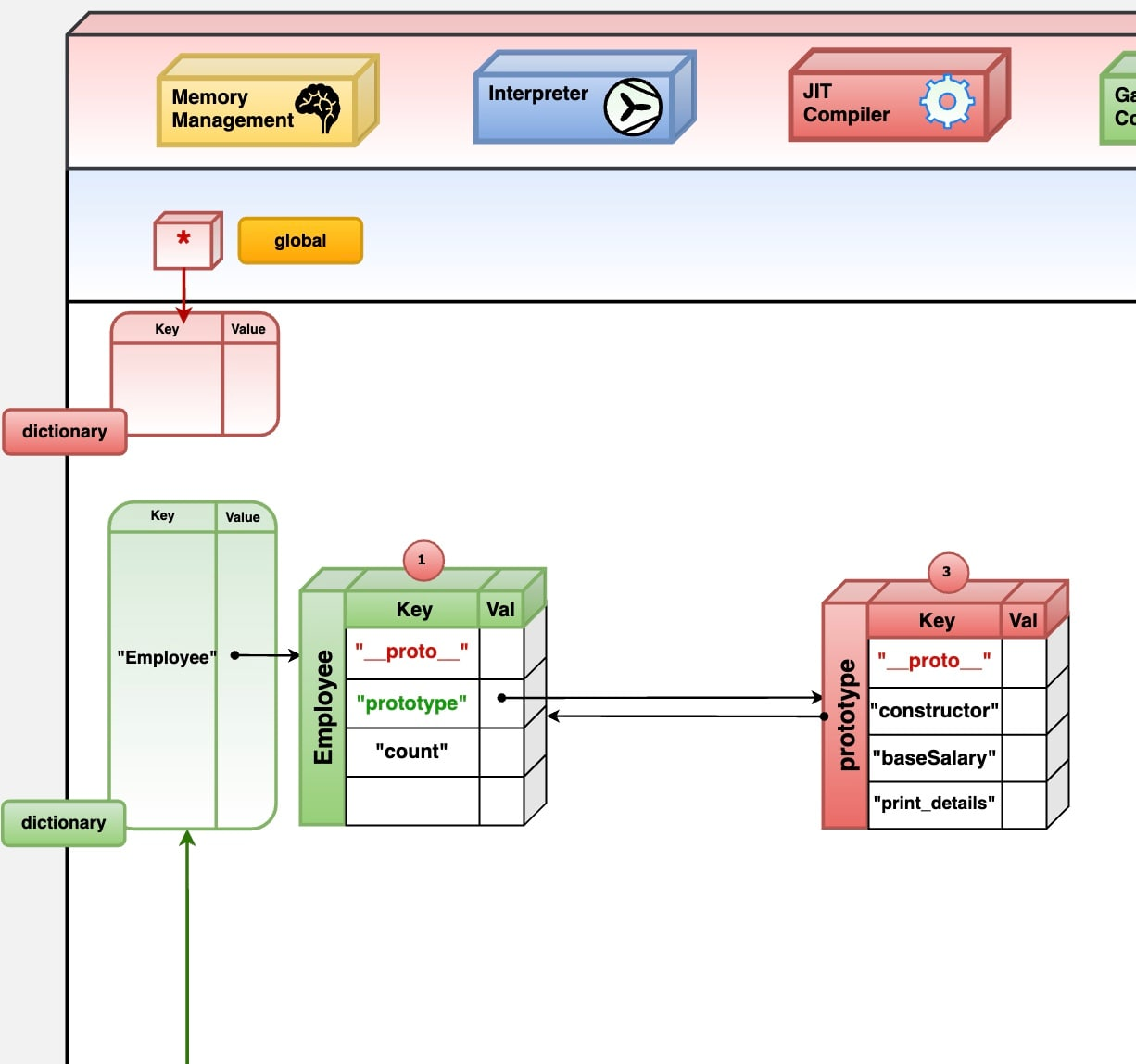
*correct constructor for future use so uncomment it for correct input*

CreatedConstructor.prototype.create = function create() {

  return new this.constructor()

}

new CreatedConstructor().create().create() *// TypeError undefined is not a function since constructor === Parent*



Here is a Concept Visual which shows how function prototype and constructor are related.

In the first case 'constructor' was not set hence .create() was undefined

In second case 'constructor' was set manually, hence it could go back to the function object and find .create()

**DOUBT Question:**

<body>

<p id="val">

<span>rhythmic beatz</span>

<b>cewwec</b>

<h1>Hello</h1>

</p>

<script>

var getp = document.getElementById("val");

var find = getdiv.getElementsByTagName("\*");

console.log(find)

</script>

</body>

//this code gives an output in the console like..

Why its not showing the h1 tag

HTMLCollection(2) [span, b]

0: span

1: b

length: 2

proto: HTMLCollection

only showing "span" tag and "b" tag..

**SOLUTION:**

It is a block level element. What that DTD is saying is that <p> tags can only contain inline elements. The p element should not contain any other block element, although it is a block element itself.

Reason: <p> auto-closes when another block element starts. It means that HTML assumes </p> before it sees another block tags.

Thtswhy we don’t see the <h1> tag inside <p> tag as it closes its tag before the <h1> tag when we try to access the parent <p>.

**BLOCK & INLINE:**

Every HTML element has a default display value, depending on what type of element it is.

There are two display values: block and inline.

Block-level Elements

A block-level element always starts on a new line.

A block-level element always takes up the full width available (stretches out to the left and right as far as it can).

A block level element has a top and a bottom margin, whereas an inline element does not.

The <**div**> element is a block-level element.

**Inline Elements**

An inline element does not start on a new line.

An inline element only takes up as much width as necessary.

This is a <**span**> element inside a paragraph.

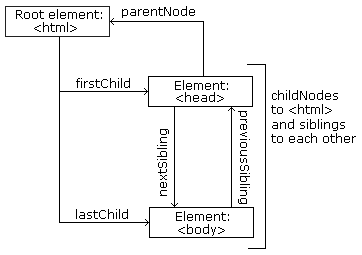
## **Node Relationships**

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

* In a node tree, the top node is called the root (or root node)
* Every node has exactly one parent, except the root (which has no parent)
* A node can have a number of children
* Siblings (brothers or sisters) are nodes with the same parent

<html>  
  
  <head>  
    <title>DOM Tutorial</title>  
  </head>  
  
  <body>  
    <h1>DOM Lesson one</h1>  
    <p>Hello world!</p>  
  </body>  
  
</html>



From the HTML above you can read:

* <html> is the root node
* <html> has no parents
* <html> is the parent of <head> and <body>
* <head> is the first child of <html>
* <body> is the last child of <html>

and:

* <head> has one child: <title>
* <title> has one child (a text node): "DOM Tutorial"
* <body> has two children: <h1> and <p>
* <h1> has one child: "DOM Lesson one"
* <p> has one child: "Hello world!"
* <h1> and <p> are siblings

## **DOM Root Nodes**

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

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

## [DOM properties](https://javascript.info/dom-attributes-and-properties#dom-properties)

We’ve already seen built-in DOM properties. But if we need our own self-made document properties , we can add our own.

DOM nodes are regular JavaScript objects. We can alter them.

For instance, let’s create a new property in document.body:

document.body.myData = {

name: 'Caesar',

title: 'Imperator'

};

alert(document.body.myData.title); // Imperator

|  |  |  |
| --- | --- | --- |
| HTML tags are used to hold the HTML element. | HTML element holds the content. | HTML attributes are used to describe the characteristic of an HTML element in detail. |

**ATTRIBUTE CONCEPTS AND USED WHEN , WHY , HOW, WHERE ?**

**Ans.**First understand difference between standard and non-standard attributes,

Standard properties are those which is created for that particular element only like, ex. In “ <input type=”text”> , type is same as something=”standard” attribute whose existence is there for input type and its pre-defined.

Non-Standard properties are those which are created for a element by the coders for any line of benefits to be used later like, ex <input about=”elephant”>, about is same as something=”non-standard” attribute which is created by user for any specific purpose only.

So, the HTML elements have attributes. When the browser parses or reads the HTML to create DOM objects for elements, it recognizes standard attributes and creates DOM properties from them.

So when an element has id or another *standard* attribute, the corresponding property gets created. But that doesn’t happen if the attribute is non-standard.

<body id="test" something="non-standard">

<script>

alert(document.body.id); // test

// non-standard attribute does not yield a property

alert(document.body.something); // undefined

</script>

</body>

Please note that a standard attribute for one element can be unknown for another one. For instance, "type" is standard for <input> ([HTMLInputElement](https://html.spec.whatwg.org/" \l "htmlinputelement)), but not for <body> ([HTMLBodyElement](https://html.spec.whatwg.org/" \l "htmlbodyelement)). Standard attributes are described in the specification for the corresponding element class.

Here we can see it:

<body id="body" type="...">

<input id="input" type="text">

<script>

alert(input.type); // text

alert(body.type); // undefined: DOM property not created, because it's non-standard, body doesnot have any “type” attribute

</script>

</body>

So, if an attribute is non-standard, there won’t be a DOM-property for it. Is there a way to access such attributes?

Sure. All attributes are accessible by using the following methods:

* elem.hasAttribute(name) – checks for existence.
* elem.getAttribute(name) – gets the value.
* elem.setAttribute(name, value) – sets the value.
* elem.removeAttribute(name) – removes the attribute.

These methods operate exactly with what’s written in HTML.

Also one can read all attributes using elem.attributes: a collection of objects that belong to a built-in [Attr](https://dom.spec.whatwg.org/" \l "attr) class, with name and value properties.

Here’s a demo of reading a non-standard property:

<body something="non-standard">

<script>

alert(document.body.getAttribute('something')); // non-standard

</script>

</body><body>

<div id="elem" about="Elephant"></div>

<script>

alert( elem.getAttribute('About') ); // (1) 'Elephant', reading

elem.setAttribute('Test', 123); // (2), writing

alert( elem.outerHTML ); // (3), see if the attribute is in HTML (yes)

for (let attr of elem.attributes) { // (4) list all

alert( `${attr.name} = ${attr.value}` );

}

</script>

</body>

The *Element* object provides various utility functions for manipulating elements in the DOM.

Here is the list of all the utility functions with examples. All the methods defined here are automatically added to any element accessed using the $() function.

So, writing Element.show('firstDiv'); is the same as writing $('firstDiv').show();

## **Prototype Element Method**

**NOTE** − Make sure you have at least version 1.6 of prototype.js.

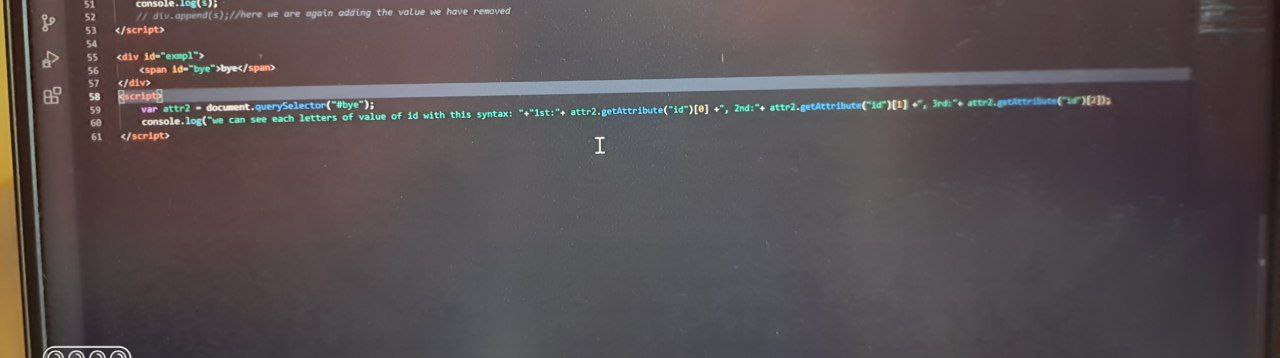
|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1. | [absolutize()](https://www.tutorialspoint.com/prototype/prototype_element_absolutize.htm)  Turns element into an absolutely-positioned element without changing its position in the page layout. |
| 2. | [addClassName()](https://www.tutorialspoint.com/prototype/prototype_element_addclassname.htm)  Adds the given CSS class name to the element's class names. |
| 3. | [addMethods()](https://www.tutorialspoint.com/prototype/prototype_element_addmethods.htm)  Makes it possible to mix in your own methods to the Element object, which you can later use as methods of extended elements. |
| 4. | [adjacent()](https://www.tutorialspoint.com/prototype/prototype_element_adjacent.htm)  Finds all siblings of the current element that match the given selector(s). |
| 5. | [ancestors()](https://www.tutorialspoint.com/prototype/prototype_element_ancestors.htm)  Collects all of element's ancestors and returns them as an array of extended elements. |
| 6. | [childElements()](https://www.tutorialspoint.com/prototype/prototype_element_childelements.htm)  Collects all of the element's |

**DocumentFragment in vanilla JS vs REACT VIRTUAL DOM**

Can anyone tell me that documentFragment in js is same as react components, first all the changes or addition or modification are done in documentFragment, and then it is apended to the HTML Body?

Ans. Yes, documentFragment and virtual DOM are similar... both create an in-memory representation of DOM tree, which can then be applied in one go

Below image with code actually says that we can print individual letters of a value in class attribute:



var divNode = document.createElement("div");

    var pnode = document.createElement("p");

    var pnode2 = document.createElement("p");

*// difference between append & appendChild*

    var textNode = pnode.append("hii bro");

*// var textNode2 = pnode.appendChild("hii bro2");// this will throw an*

*error because  appendChild doesnot except text or comment nodes for adding to HTML DOM*

    console.log(divNode.append(pnode));*//append doesnot return anything*

    console.log(divNode.appendChild(pnode));*//append child is a return type*

*and returns the child element*

obj.method(params) -> is almost equivalent to -> obj.method.call(obj, params), OR

let data = [].map.call(elements, elem => elem.textContent);

example :

### Spread Operator (...)

Lets take the array as a bottle. The spread operator allows you to grab all the content of the bottle without grabbing the bottle itself and putting that content wherever we want. Let's take a look at the following code.

const animals = [🦁, 🐘, 🐍, 🦍, 🐯];

const someOtherAnimals = [...animals];

// someOtherAnimals now contains 🦁, 🐘, 🐍, 🦍, 🐯 and

// animals remains unchanged

Here, the spread operator is used to copy the content of the animals array into the someOtherAnimals array. The spread operator can also be used to copy the contents of multiple arrays into another array.

const wild = [🦁, 🐘, 🐍, 🦍, 🐯];

const domestic = [🐐, 🐔, 🐱, 🐶];

const animals = [...wild, ...domestic];

// animals now contains 🦁, 🐘, 🐍, 🦍, 🐯, 🐐, 🐔, 🐱, 🐶 and

// both wild and domestic remains unchanged.

### Rest Operator (...)

The rest operator allows to us to represent an indefinite number of arguments as an array. So unlike the spread operator that spreads out the elements in an array, the rest operator (or the gather operator as some people call it) groups multiple elements into an array. Here's how that would work

const addAll = (...numbers) => {

return numbers.reduce((acc, num) => acc + num);

};

const sum = addAll(1, 2, 3, 4); // sum is 10

const sum1 = a

this == can be manually passed

var Car = {

  color: 'red',

  owner: 'foo',

  price: 1000

}

function Product(title, price) {

  this.title = title;

  this.price = price;

  console.log(this); // Car object with passed title and price properties

  return 9;

}

let p2 = Product.apply(Car, ['Pen', 1000]); // Indirectly called using apply

let p1 = Product.call(Car, 'Pencil', 700); // Indirectly called using call

console.log(p1); // 9

console.log(Car); // only one Car object

JavaScript calculates value for **this** for you

For example - when you call function as function which is also known s as function invocation pattern , “this” is global object of window in browser

If you call function as constructor means using new , value of “this “ is newly created object.

However if you want to override JavaScript provided this in a function and pass your object as value of this inside a function , you need to call the function either using apply or call or bind. This is also called indirect invocation pattern .So in above code Element object is passed as value of this

## A)Children vs childNodes : childNodes include the text node and comment node also along with other nodes like div, h1, etc,

Whereas Children are used ,excluding the text and comment nodes, for the nodes like div,h1 and the rest except commnet and text.

## B) .append()

This method is used to add an element in form of a Node object or a DOMString (basically means text). Here's how that would work.

// Inserting a Node object

const parent = document.createElement('div');

const child = document.createElement('p');

parent.append(child);

// This appends the child element to the div element

// The div would then look like this <div><p></p></div>

// Inserting a DOMString

const parent = document.createElement('div');

parent.append('Appending Text');

// The div would then look like this <div>Appending Text</div>

## .appendChild()

Similar to the **.append** method, this method is used to elements in the DOM, but in this case, only accepts a Node object.

// Inserting a Node object

const parent = document.createElement('div');

const child = document.createElement('p');

parent.appendChild(child);

// This appends the child element to the div element

// The div would then look like this <div><p></p></div>

// Inserting a DOMString

const parent = document.createElement('div');

parent.appendChild('Appending Text');

// Uncaught TypeError: Failed to execute 'appendChild' on 'Node': parameter 1 is not of type 'Node'

console.log("1st difference");

  var textNode = pnode.append("hii bro");*//here we can use append to insert DOM Strings to elemnts*

    var textNode2 = pnode.appendChild("hii bro2");*// this will throw an error because  appendChild  doesnot except text or comment nodes for adding to HTML DOM*

But instead of inserting DOM strings to apend child, we can create text or comment  nodes first, and then we can append it to element ,lets see in below code:

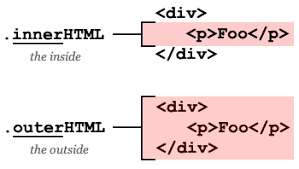
 var pNode = document.createElement("div");

    var nNode = document.createComment("hii brother");

    var cnode = document.createTextNode("good boy");

    pNode.appendChild(nNode);

    console.log(pNode);



a)innerHTML will change the content of the element whereas,

b)outerHTML will change the whole element with the new element or text, it’s a kinda replace.

# CSS Object Model (CSSOM)

The **CSS Object Model** is a set of APIs allowing the manipulation of CSS from JavaScript. It is much like the DOM, but for the CSS rather than the HTML. It allows users to read and modify CSS style dynamically.

<p id=”myp” style=”background-color: darkgrey”>Paragraph</p>

Here background-color is the CSS property but “p.style.backgroundColor = ‘red’ “ it’s a DOM property to manipulate or modify the css.

**JAVSCRIPT FUNCTION “SORT” IN-Depth DETAIL**

To study the behavior of [Array#sort](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) we are using naive [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort) which actually written behind the sort function in WEB API.

const sort = arr => {

for (let i = 1; i < arr.length; i++) {

for (let j = i; j && arr[j-1] > arr[j]; j--) {

[arr[j], arr[j-1]] = [arr[j-1], arr[j]];

}

}

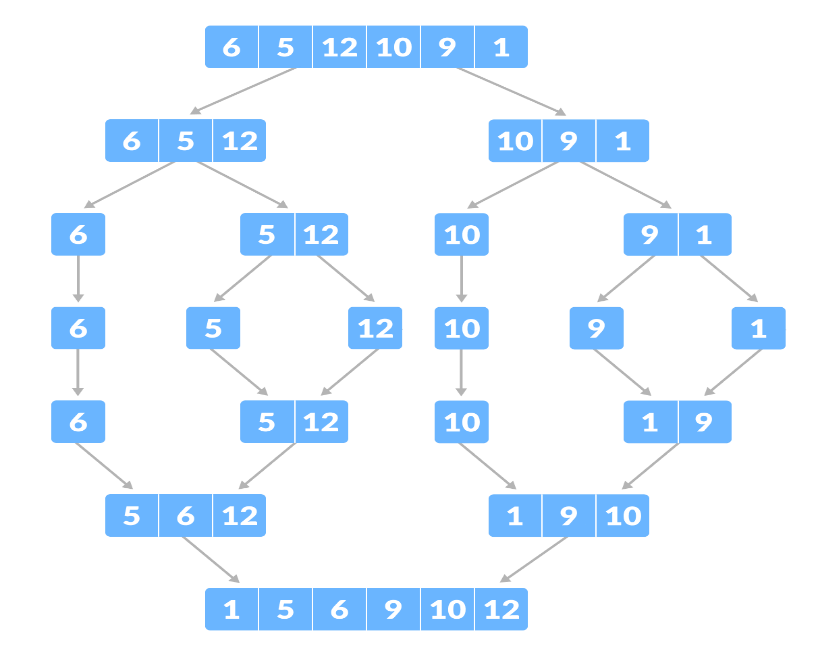
};

const array = [3, 0, 4, 5, 2, 2, 2, 1, 2, 2, 0];

sort(array);

console.log("" + array);

***INSERTION SORT IN DEPTH***



<script>

    const \_mergeArrays = (a, b) => {

  const c = []

  while (a.length && b.length) {

*//   for a=[5,6,7] and for b=[1,9,10]*

*// a[0]=5 > b[0], then b.shift will remove the first elemnt from arr2, and now array becomes b=[9,10], and the shifted value (here 1) is stored in the  “c” array, again,*

*// now a=[5,6,7] and b=[9,10],*

*//  a[0]=5 > b[0],condition false => a.shift occurs => and (a[0]=5) stored in c array =>now array becomes =>*

*//  a=[6,7] and b=[9,10], and the process continues, and c array upto here will be like c=[1,5]*

    c.push(a[0] > b[0] ? b.shift() : a.shift());

    console.log(c);

  }

*//if we still have values, let's add them at the end of `c`*

*//   for example when two arrays with value like (5) & (6) are cheked above,  then one is pushed in the c array, but pusing the other value in c array, we  need this line because,*

*// in above code, loop will run until either of a.length or b.length becomes  zero, if any one of them has length zero, then the loop will not run*

*// b.shift or a.shift will remove the value from the array*

  while (a.length) {

*// for array like a=[5] b=[12] , first it's checked which is greater in  above code, so (5) which is the smaller is shifted to c array*

*// now b= [] (empty) ,so b.length =0 =>now above code doesnot satiesfies  as oneof the length is zero, so then it comes to these block, and the array  which is left to insert in c array is pushed there, here (12)*

    c.push(a.shift())

  }

  while (b.length) {

    c.push(b.shift())

  }

  return c

}

const mergeSort = (a) => {

  if (a.length < 2) return a

  const middle = Math.floor(a.length / 2)

  const a\_l = a.slice(0, middle)

  const a\_r = a.slice(middle, a.length)

  const sorted\_l = mergeSort(a\_l)

  const sorted\_r = mergeSort(a\_r)

  return \_mergeArrays(sorted\_l, sorted\_r)

}

var a=[6,5,12,10,9,1];

console.log("merge sort with another technic");

console.log(mergeSort(a));

</script>

JAVASCRIPT EVENTS

A bubbling event goes from the target element straight up. Normally it goes upwards till <html>, and then to document object, and some events even reach window, calling all handlers on the path.

1) EVENT.TARGET

<p onclick="alert('pp')">Click on a paragraph. An alert box will alert the element that triggered the event.</p>

<p><strong>Note:</strong> The target property returns the element that triggered the event, and not necessarily the eventlistener's element.</p>

<h1 onclick="alert('h1')">asdsad

<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

<p onclick="alert('p')">P</p>

</div>

</form>

</h1>

<script>

function myFunction(event) {

alert(event.target.nodeName);

}</script>

</body>

EXPLANATION: The target event property returns the element that triggered the event and goes on bubbling like bubbles to the tope, and in the way if it finds any other events, It triggers them also.

Suppose you clicked on the element “FORM” , then we will the event in form is triggred, then it follows a bottom to top approach. Now, as we clicked on “FORM” our bottom is now the element “FORM” , now it moves from bottom to top, and triggers all other elemnts in the way until it reaches its parent. Here parent is “H1” Node. And then stops. Here “H1” is the parent and “form” , “div” , “p” are the childs.

The target property gets the element on which the event originally occurred, opposed to the [currentTarget](https://www.w3schools.com/jsref/event_currenttarget.asp) property, which always refers to the element whose event listener triggered the event.

2)EVENT.currentTarget

<body onclick="myFunction(event)">

<p>Click on a paragraph. An alert box will alert the element whose eventlistener triggered the event.</p>

<p><strong>Note:</strong> The currentTarget property does not necessarily return the element that was clicked on, but the element whose eventlistener triggered the event.</p>

<script>

function myFunction(event) {

alert(event.currentTarget.nodeName);

}

</script>

</body>

EXPLANATION: The currentTarget event property returns the element whose event listeners triggered the event.This is particularly useful during capturing and bubbling.

The currentTarget property always refers to the element whose event listener triggered the event, opposed to the [target](https://www.w3schools.com/jsref/event_target.asp) property, which returns the element that triggered the event.

## [Bubbling](https://javascript.info/bubbling-and-capturing" \l "bubbling)

The bubbling principle is simple.

**When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.**

Let’s say we have 3 nested elements FORM > DIV > P with a handler on each of them:

<style>

body \* {

margin: 10px;

border: 1px solid blue;

}

</style>

<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

<p onclick="alert('p')">P</p>

</div>

</form>

A click on the inner <p> first runs onclick:

1. On that <p>.
2. Then on the outer <div>.
3. Then on the outer <form>.
4. And so on upwards till the document object.

So if we click on <p>, then we’ll see 3 alerts: p → div → form.

The process is called “bubbling”, because events “bubble” from the inner element up through parents like a bubble in the water.

**Almost all events bubble.**

The key word in this phrase is “almost”.

For instance, a focus event does not bubble. There are other examples too, we’ll meet them. But still it’s an exception, rather than a rule, most events do bubble.

## [event.target](https://javascript.info/bubbling-and-capturing" \l "event-target)

A handler on a parent element can always get the details about where it actually happened.

**The most deeply nested element that caused the event is called a target element, accessible as event.target.**

Note the differences from this (=event.currentTarget):

* event.target – is the “target” element that initiated the event, it doesn’t change through the bubbling process.
* this – is the “current” element, the one that has a currently running handler on it.

For instance, if we have a single handler form.onclick, then it can “catch” all clicks inside the form. No matter where the click happened, it bubbles up to <form> and runs the handler.

In form.onclick handler:

* this (=event.currentTarget) is the <form> element, because the handler runs on it.
* event.target is the actual element inside the form that was clicked.

Check it out:

script.js

form.onclick = function(event) {

event.target.style.backgroundColor = 'yellow';

// chrome needs some time to paint yellow

setTimeout(() => {

alert("target = " + event.target.tagName + ", this=" + this.tagName);

event.target.style.backgroundColor = ''

}, 0);

};

It’s possible that event.target could equal this – it happens when the click is made directly on the <form> element.

## [Stopping bubbling](https://javascript.info/bubbling-and-capturing" \l "stopping-bubbling)

A bubbling event goes from the target element straight up. Normally it goes upwards till <html>, and then to document object, and some events even reach window, calling all handlers on the path.

But any handler may decide that the event has been fully processed and stop the bubbling.

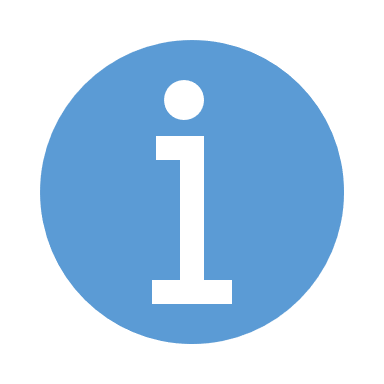
The method for it is event.stopPropagation().

For instance, here body.onclick doesn’t work if you click on <button>:

<body onclick="alert(`the bubbling doesn't reach here`)">

<button onclick="event.stopPropagation()">Click me</button>

</body>

**Event.stopImmediatePropagation()**

* If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.
* In other words, event.stopPropagation() stops the move upwards, but on the current element all other handlers will run.
* To stop the bubbling and prevent handlers on the current element from running, there’s a method event.stopImmediatePropagation(). After it no other handlers execute.

** Don’t stop bubbling without a need!**

Bubbling is convenient. Don’t stop it without a real need: obvious and architecturally well thought out.

Sometimes event.stopPropagation() creates hidden pitfalls that later may become problems.

For instance:

1. We create a nested menu. Each submenu handles clicks on its elements and calls stopPropagation so that the outer menu won’t trigger.
2. Later we decide to catch clicks on the whole window, to track users’ behavior (where people click). Some analytic systems do that. Usually the code uses document.addEventListener('click'…) to catch all clicks.
3. Our analytic won’t work over the area where clicks are stopped by stopPropagation. Sadly, we’ve got a “dead zone”.

There’s usually no real need to prevent the bubbling. A task that seemingly requires that may be solved by other means. One of them is to use custom events, we’ll cover them later. Also we can write our data into the event object in one handler and read it in another one, so we can pass to handlers on parents information about the processing below.

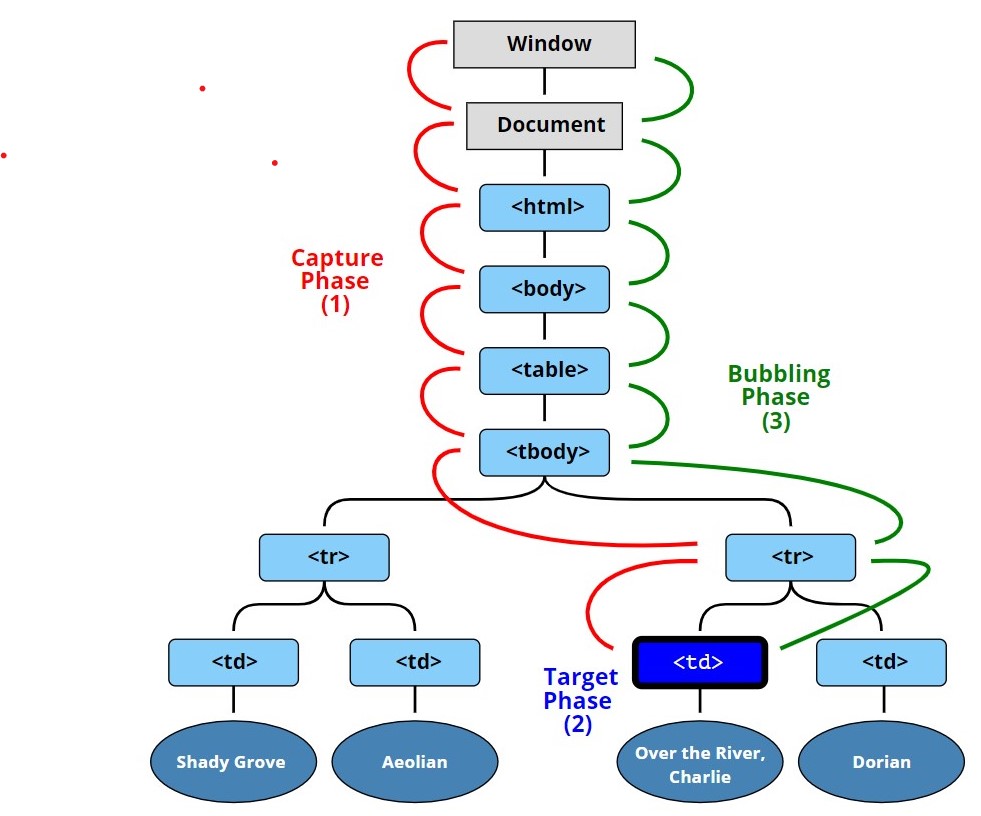
## [Capturing](https://javascript.info/bubbling-and-capturing" \l "capturing)

There’s another phase of event processing called “capturing”. It is rarely used in real code, but sometimes can be useful.

The standard [DOM Events](http://www.w3.org/TR/DOM-Level-3-Events/) describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.

Here’s the picture of a click on <td> inside a table, taken from the specification:

That is: for a click on <td> the event first goes through the ancestors chain down to the element (capturing phase), then it reaches the target and triggers there (target phase), and then it goes up (bubbling phase), calling handlers on its way.

## **The addEventListener() method**

The **addEventListener**() method accepts three arguments: an event name, an event handler function, and a **Boolean** value that instructs the method to call the event handler during the capture phase (**true**) or during the bubble phase (**false**).

**Before we only talked about bubbling, because the capturing phase is rarely used. Normally it is invisible to us.**

Handlers added using on<event>-property or using HTML attributes or using two-argument addEventListener(event, handler) don’t know anything about capturing, they only run on the 2nd and 3rd phases.

To catch an event on the capturing phase, we need to set the handler capture option to true:

elem.addEventListener(..., {capture: true})

// or, just "true" is an alias to {capture: true}

elem.addEventListener(..., true)

There are two possible values of the capture option:

* If it’s false (default), then the handler is set on the bubbling phase.
* If it’s true, then the handler is set on the capturing phase.

<form>FORM

<div>DIV

<p>P</p>

</div>

</form>

<script>

for(let elem of document.querySelectorAll('\*')) {

elem.addEventListener("click", e => alert(`Capturing: ${elem.tagName}`), true);

elem.addEventListener("click", e => alert(`Bubbling: ${elem.tagName}`)); }

</script>

The code sets click handlers on *every* element in the document to see which ones are working.

If you click on <p>, then the sequence is:

1. HTML → BODY → FORM → DIV (capturing phase, the first listener):
2. P (target phase, triggers two times, as we’ve set two listeners: capturing and bubbling)
3. DIV → FORM → BODY → HTML (bubbling phase, the second listener).

**Listeners on same element and same phase run in their set order**

* If we have multiple event handlers on the same phase, assigned to the same element with addEventListener, they run in the same order as they are created:
* elem.addEventListener("click", e => alert(1)); // guaranteed to trigger first
* elem.addEventListener("click", e => alert(2));

## [Summary](https://javascript.info/bubbling-and-capturing" \l "summary)

When an event happens – the most nested element where it happens gets labeled as the “target element” (event.target).

* Then the event moves down from the document root to event.target, calling handlers assigned with addEventListener(..., true) on the way (true is a shorthand for {capture: true}).
* Then handlers are called on the target element itself.
* Then the event bubbles up from event.target to the root, calling handlers assigned using on<event>, HTML attributes and addEventListener without the 3rd argument or with the 3rd argument false/{capture:false}.

Each handler can access event object properties:

* event.target – the deepest element that originated the event and it’s the child of the parent.
* event.currentTarget (=this) – the current element or parent that handles the event (the one that has the handler on it)
* event.eventPhase – the current phase (capturing=1, target=2, bubbling=3).

<form id="forms" data-name="1">FORMs

    <div id="divs" data-name="2">DIVs

      <p id="para" data-name="3">Ps</p>

    </div>

  </form>

<script>

Var Formbubbling = document.getElementById(“forms”);

Formbubbling.addEventListener("click",function(event){

        console.log("target = "+ event.target.tagName + ",this = "+event.currentTarget.tagName);

    });

</script>

**OUTPUT:**

**target = P,this = FORM, When clicked on p element**

**target = DIV,this = FORM, When clicked on div element**

**target = FORM,this = FORM, When clicked on form element**

Different MOUSE EVENTS

* + mousedown – It fires just when the mouse button is pressed down but not released
  + mouseup – It fires when the mouse button is released from being pressed
  + click – It fires as soon as something is clicked, in this scenario , mousedown, mouseup & click all event are together fired in this sequence.
* dblclick – It fires when something is clicked twice in rapidly, mousedown, mouseup, click, mousedown, mouseup, click and dblclick events fires in sequence.
* mouseenter - It fires when a mouse starts to hover over same element. NO BUBBLE
* mouseleave – It fires when a mouse leaves from hovering over some element
* mousemove – It fires when the mouse moves
* contextmenu – It fires when mouse right button is clicked

**CALLBACKS:**

No, it doesn't. "Execution context" doesn't mean "thread", and until recently Javascript had no support for anything resembling threads. What actually happens is that an event is pushed onto the event queue that's set to execute in the number of milliseconds specified by the second argument to SetTimeout/SetInterval. The consequence of this is that if you request a 1000ms delay, then 1000ms is the MINIMUM delay you'll get. If the execution engine is busy doing something else when the 1000ms delay is over then it will have to wait until it's done with what it's doing.

A Javascript engine simply processes a queue of events sequentially on a single thread. When the event queue is empty, that thread idles. In a browser, events are added to the queue by user input, page loading, etc. In node.js events can also be HTTP requests or hardware events. And setTimeout() simply adds another kind of event with the additional condition that it should only be added to the queue after a certain time has passed.

When calling **setTimeout** or setInterval , a timer thread in the browser starts counting down and when time up puts the callback function in **javascript** thread's execution stack. The callback function is not executed before other functions above it in the stack finishes.

Firstoff, the term callback refers to a closure that is being used for something.

For example, suppose you create a closure function and just store it in a variable. It isn't a callback, because it isn't being used for anything.

But, suppose you create a closure and store it somewhere that it will get called when something happens. It is now termed a callback.

Usually, callbacks are created by different parts of the program than the parts that call them. So it's easy to imagine that some parts of the program are "calling back" the other parts.

Simply put, **callbacks allow one part of the program to tell another part to do something (anything) when something happens.**

* setTimeout(callback, time)
* This is the signature
* ()=> {} this is a function
* An anonymous function is present in setTimeout parameter, which is a callback func

let calbckFunc = (items) => {

  setTimeout(() => {

console.log("hii"); }, 3000); }

let order = (items, calbck) => {

    setTimeout(() => {

calbck(items);

}, 0000); }

order('Cake', calbckFunc);

* The setTimeout has its one of the parameter only which is holding a callback function itself, I didn't saw the signature of setTimeout, now I understand....my eyes didn't caught the callback function setTimeout was holding as a parameter, I was only seeing the content inside the callback.

but inside the calback function of setTimeout, we can either choose anyone of the function calling...with callback or without it.

callbacks are mainly used in addEventListeners and setTimeout, because these are predefined with one of their parameter as callback, but for other cases we can do normal functional calling also.

CONCLUSION: Callback are just some functions which is to be called back later to serve some purpose of the same code, and we can directly call a outside function also without callBack. They are mainly in used in WEB API like addEventListeners, setTimeout etc., for delaying some task and making the rest of the code to eecute.

## What Is Asynchronous Programming?

**An**[**asynchronous**](https://www.baeldung.com/java-asynchronous-programming)**model allows multiple things to happen at the same time**. When your program calls a long-running function, **it doesn’t block the execution flow**, and your program continues to run. When the function finishes, the program knows and gets access to the result (if there’s a need for that).

ASYNCHRONOUS NATURE:

JS is asynchronous because, it may happen that in the middle of the code there is some function which is to be delayed or can take time which will make the overall code to run slow, so thatswhy we introduced async functions like callback, and WebAPI’s for executing the asynchronous nature.

So, now we enabled async nature, So what is it doing.

* It is helping the code to run and execute smoothly, and if there is any code which is to be delayed, then we are sending it using a “callback” parameter which is present inside the setTimeout Function like (“ setTimeout(callback, time) ”), and inside this callback we are writing the codes that are to be delayed and by what “time”…But due to this delay, the rest of the code doesn’t stop, they are continuously executed until the end, and when nothing is left to execute, then this seTimeout values are executed and returned.
* Eventloop, whenever WebAPI’s like addEventListener and setTimeout are seen in code, they are first sent to the event loop, which is further sent to the event Queue, and there “FIFO” is followed, so the event which comes first in code is served first…and these contents of event Queue is sent to the stack only after the entire JS code is executed upto the end.

# EVERYTHING NEED TO KNOW ABOUT FUNCTION

# What is Functional Programming

In most simple term, Functional Programming is a form of programming in which you can pass functions as parameters to other functions and also return them as values. In functional programming, we think and code in terms of functions.

JavaScript, Haskell, Clojure, Scala, and Erlang are some of the languages that implement functional programming.

# First-Class Functions

If you have been learning JavaScript, you may have heard that JavaScript treats functions as first-class citizens. That’s because in JavaScript or any other functional programming languages functions are objects.

In JavaScript functions are a special type of objects. They are Function objects. For example:

To prove functions are objects in JavaScript, we could do something like this:

function greeting() {

  console.log('Hello World');

}

// Invoking the function

// We can add properties to functions like we do with objects

greeting();  // prints 'Hello World'

greeting.lang = 'English';

// Prints 'English '

console.log(greeting.lang); // Prints 'English'

**Note —** While this is perfectly valid in JavaScript, this is considered a harmful practice. You should not add random properties to the function objects, use an object if you have to.

In JavaScript, everything you can do with other types like object, string, or number, you can do with functions. You can pass them as parameters to other functions (callbacks), assign them to variables and pass them around etc. This is why functions in JavaScript are known as First-Class Functions.

## **Passing Functions as Parameters**

We can pass functions as parameters to other functions. For example:

     const arr = [1,3,5,7];

    var output3 = [];

    const mapArea = function(arr){

        return Math.PI \* arr \* arr;

    }

    const mapCircumference = function(radius){

        return 2 \* Math.PI \* radius;

    }

    // this below code is a whole replacement of map function and is termed as higher order function

    Array.prototype.protofunc = (logic) => {

        console.log(this);

        for(var k=0; k<this.length ; k++){

            output3.push(logic(this[k]));

        }

        return output3;

    }

    console.log('Use of map function just to calculate area instead of passing callbacks and all')

    console.log(arr.map(area));

    console.log(arr.protofunc(area));

Now that we know what first-class functions are, let’s dive into Higher-Order functions in JavaScript.

# Higher-Order Functions

Higher order functions are functions that operate on other functions, either by taking them as arguments or by returning them. In simple words, A Higher-Order function is a function that receives a function as an argument or returns the function as output.

For example, Array.prototype.map, Array.prototype.filter and Array.prototype.reduce are some of the Higher-Order functions built into the language.

## **Higher-Order Functions in Action**

Let’s take a look at some examples of built-in higher-order functions and see how it compares to solutions where we aren’t using Higher-Order Functions.

## **Array.prototype.map**

The map() method creates a new array by calling the callback function provided as an argument on every element in the input array. The map() method will take every returned value from the callback function and creates a new array using those values.

The callback function passed to the map() method accepts 3 arguments: element, index, and array.

Let’s look at some examples:

## **Example 1#**

Let’s say we have an array of numbers and we want to create a new array which contains double of each value of the first array. Let’s see how we can solve the problem with and without Higher-Order Function.

**Without Higher-order function**

const arr1 = [1, 2, 3];  
const arr2 = [];for(let i = 0; i < arr1.length; i++) {  
 arr2.push(arr1[i] \* 2);  
}// prints [ 2, 4, 6 ]  
console.log(arr2);

**With Higher-order function map**

const arr1 = [1, 2, 3];const arr2 = arr1.map(function(item) {  
 return item \* 2;  
});console.log(arr2);

We can make this even shorter using the arrow function syntax:

const arr1 = [1, 2, 3];const arr2 = arr1.map(item => item \* 2);console.log(arr2);

## **Example 2#**

Let’s say we have an array containing the birth year of different persons and we want to create an array that contains their ages. For example:

**Without Higher-order function**

const birthYear = [1975, 1997, 2002, 1995, 1985];  
const ages = [];for(let i = 0; i < birthYear.length; i++) {  
 let age = 2018 - birthYear[i];  
 ages.push(age);  
}// prints [ 43, 21, 16, 23, 33 ]  
console.log(ages);

**With Higher-order function map**

const birthYear = [1975, 1997, 2002, 1995, 1985];const ages = birthYear.map(year => 2018 - year);// prints [ 43, 21, 16, 23, 33 ]  
console.log(ages);

## **Array.prototype.filter**

The filter() method creates a new array with all elements that pass the test provided by the callback function. The callback function passed to the filter() method accepts 3 arguments: element, index, and array.

Let’s look at some examples:

## **Example 1#**

Let’s say we have an array which contains objects with name and age properties. We want to create an array that contains only the persons with full age (age greater than or equal to 18).

**Without Higher-order function**

const persons = [  
 { name: 'Peter', age: 16 },  
 { name: 'Mark', age: 18 },  
 { name: 'John', age: 27 },  
 { name: 'Jane', age: 14 },  
 { name: 'Tony', age: 24},  
];const fullAge = [];for(let i = 0; i < persons.length; i++) {  
 if(persons[i].age >= 18) {  
 fullAge.push(persons[i]);  
 }  
}console.log(fullAge);

**With Higher-order function filter**

const persons = [  
 { name: 'Peter', age: 16 },  
 { name: 'Mark', age: 18 },  
 { name: 'John', age: 27 },  
 { name: 'Jane', age: 14 },  
 { name: 'Tony', age: 24},  
];const fullAge = persons.filter(person => person.age >= 18);console.log(fullAge);

## **Array.prototype.reduce**

The reduce method executes the callback function on each member of the calling array which results in a single output value. The reduce method accepts two parameters: 1) The reducer function (callback), 2) and an optional initialValue.

The reducer function (callback) accepts four parameters: accumulator, currentValue, currentIndex, sourceArray.

If an initialValue is provided, then the accumulator will be equal to the initialValue and the currentValue will be equal to the first element in the array.

If no initialValue is provided, then the accumulator will be equal to the first element in the array and the currentValue will be equal to the second element in the array.

## **Example 1#**

Let’s say we have to sum an array of numbers:

**With Higher-order function reduce**

const arr = [5, 7, 1, 8, 4];const sum = arr.reduce(function(accumulator, currentValue) {  
 return accumulator + currentValue;  
});// prints 25  
console.log(sum);

Every time the reducer function is called on each value in the array, the accumulator keeps the result of previous operation returned from the reducer function, and the currentValue is set to the current value of the array. In the end the result is stored in the sum variable.

We can also provide an initial value to this function:

const arr = [5, 7, 1, 8, 4];const sum = arr.reduce(function(accumulator, currentValue) {  
 return accumulator + currentValue;  
}, 10);// prints 35  
console.log(sum);

**Without Higher-order function**

const arr = [5, 7, 1, 8, 4];let sum = 0;for(let i = 0; i < arr.length; i++) {  
 sum = sum + arr[i];  
}// prints 25  
console.log(sum);

You can see that using High-order function made our code cleaner, more concise and less verbose.

# Creating Our own Higher-Order Function

Up until this point, we saw various Higher-order functions built into the language. Now let’s create our own Higher-order function.

Let’s imagine JavaScript didn’t have the native map method. We could build it ourselves thus creating our own Higher-Order Function.

Let’s say, we have an array of strings and we want to convert this array to an array of integers, where each element represent the length of the string in the original array.

const strArray = ['JavaScript', 'Python', 'PHP', 'Java', 'C'];  
function mapForEach(arr, fn) { const newArray = [];  
 for(let i = 0; i < arr.length; i++) {  
 newArray.push(  
 fn(arr[i])  
 );  
 } return newArray;  
}const lenArray = mapForEach(strArray, function(item) {  
 return item.length;  
});// prints [ 10, 6, 3, 4, 1 ]  
console.log(lenArray);

In the above example, we created an Higher-order function mapForEach which accepts an array and a callback function fn. This function loops over the provided array and calls the callback function fn inside the newArray.push function call on each iteration.

The callback function fn receives the current element of array and returns the length of that element, which is stored inside the newArray. After the for loop has finished, the newArray is returned and assigned to the lenArray.

**ARROW FUNCTIONS Vs NORMAL FUNCTION & “THIS “**

#### **Arrow function and this context**

[this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) represents an object that executes the current function. In short, this is defined by the function execution context. Such as how a function is called, it refers to a global object window. For example, when a function is being executed from a global object.

You might have used this keyword in a real-life situation without realizing it. Suppose you are walking along with your mother and meet a friend along the way. This is how you would introduce your mom to your friend. ***This is my mother.***

Take a close look at this is that sentence. this shows a reference to your mother. this represents the mother in the current sentence. It is the same way JavaScript uses this keyword.

Let see how JavaScript will refer to a mother using this.

// an object parent with the property mom\_name

const parent = {

// add property

mom\_name: "Samantha Quinn",

// create a method to return parent

mother: function () {

return "This is my mother.";

},

};

console.log(parent.mother());

Log ***This is my mother*** to the console. But what if we replace “this” in the sentence with JavaScript this keyword.

// an object parent with the property mom\_name

const parent = {

mom\_name: "Samantha Quinn",

mother: function () {

return `${this} is my mother.`;

},

};

console.log(parent.mother());

This will print an object because we didn’t reveal the mother’s name. We didn’t refer to the mother. It executes the object parent, but we didn’t refer this to the property of parent.

If we specify the mother’s name.

const parent = {

mom\_name: "Samantha Quinn",

mother: function () {

return `${this.mom\_name} is my mother`;

},

};

console.log(parent.mother());

***Samantha Quinn is my mother***, will be printed in the console.

In this example, we used the this keyword to refer to the parent. Meaning this refers to its parent object. It refers to the context where the anonymous function is called. And this will bind to the parent object to return the name of the mother.

What if we use this globally. Let’s see that with examples.

function test() {

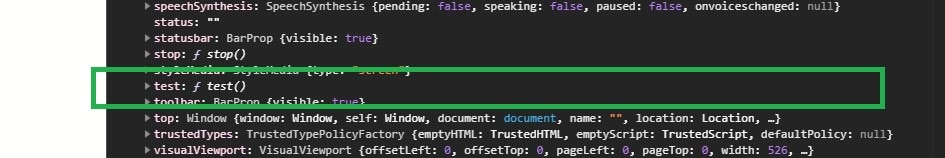
console.log(this);

}

test();

Run the above call in a browse console. You will get something like:

Window {window: Window, . . . .}



This is because the test() is called from a global context, and this will refer to a global object. In this case, a global object window is called from the browser. this is not defined by the caller.

Thus, it will turn to the default window object. The Javascript engine will check if test() is available in the window object. If not, the engine will add it to the many available Javascript window object methods.

Let’s get a little complex. What if we use the arrow function.

const parent = {

mom\_name: "Samantha Quinn",

mother: () => {

return `${this.mom\_name} is my mother.`;

},

};

console.log(parent.mother());

We get undefined......

It actually makes sense. In the regular function, a function always defines its this value. Arrow functions treat this keyword differently. They don’t define their own context since it doesn’t have its own this context. They inherit that from the parent scope whenever you call this.

this in regular function always refers to the context of the function being called. However, in the arrow function, this has nothing to do with the caller of the function. It refers to the scope where the function (the enclosing context) is present. That’s why we get undefined.

To understand this, let’s go through another example.

function User() {

(this.name = "John Doe"),

(this.score = 20),

(this.sayUser = function () {

// when `this` is accessible

console.log(this.name);

function innerFunction() {

// when `this` refers to the global scope/object

console.log(this.name);

}

innerFunction();

});

}

let name = new User();

name.sayUser();

Output:

John Doe

undefined

In the first case, we get the user name because this.name is inside this.sayUser, which is accessible. The reason is because this.sayUser is a method of the object User.

On the other hand, the this.name inside innerFunction function is not accessible. It refers to the global object window where sayUser is not defined—thus returning undefined.

To solve that, you would typically assign this to a variable that doesn’t shadow innerFunction.

For example:

function User() {

(this.name = "John Doe"),

(this.score = 20),

(this.sayUserName = function () {

// when `this` is accessible

console.log(this.name);

// when `this` refers to the global scope/object

// a variable that doesn't shadow `innerFunction`

let self = this;

function innerFunction() {

console.log(self.name);

}

innerFunction();

});

}

let name = new User();

name.sayUserName();

However, when the innerFunction is inside an arrow function, this will refer to the parent scope by creating this of its own context.

For example:

function User() {

(this.name = "John Doe"),

(this.score = 20),

(this.sayUser = function () {

// when `this` is accessible

console.log(this.name);

let innerFunction = () => {

// when `this` refers to the global scope/object

console.log(this.name);

};

innerFunction();

});

}

let name = new User();

name.sayUser();

Let’s look at the broader scope of how the arrow function binds to this keyword.

Here is an example that uses the regular function.

Execute the examples below with the console. You can use the Google chrome console.

let animals = {

// add property

domesticAnimals: ["cat", "dog", "cow", "goat", "sheep", "donkey", "pig", "horse"],

// add method

printdomesticAnimals: function () {

// print after 3 seconds

setTimeout(function () {

console.log(this.domesticAnimals.join(" - "));

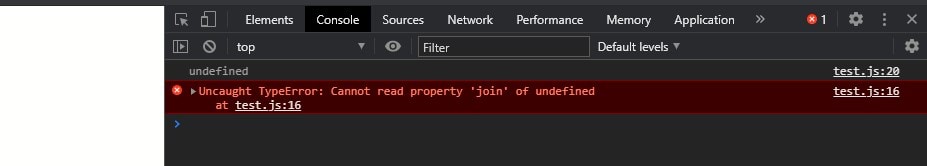
}, 3000);

},

};

animals.printdomesticAnimals();

The above example accesses the object’s property domesticAnimals and prints the domestic animals in 3 seconds. Unfortunately, we ran into an error.



setTimeout() can’t find the domesticAnimals, which means the JavaScript engine interprets domesticAnimals as undefined. Meaning this doesn’t point to the property domesticAnimals. this seems to be pointing to somewhere else.

Does it refer to the:

* inner function context
* the outer (enclosing) function context
* the object context or
* the window context?

The example below will help us to understand where this keyword points. We will log this inside the context of outer (printdomesticAnimals()) and inner (setTimeout()) function.

let animals = {

// add property

domesticAnimals: ["cat", "dog", "cow", "goat", "sheep", "donkey", "pig", "horse"],

// add method

printdomesticAnimals: function () {

console.log("inside printdomesticAnimals", this);

// print after 3 seconds

setTimeout(function () {

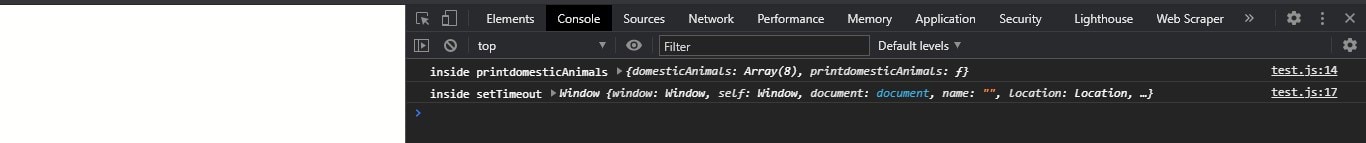
console.log("inside setTimeout", this);

}, 3000);

},

};

animals.printdomesticAnimals();



this inside printdomesticAnimals() points to the object animals with the domesticAnimals property. this inside setTimeout() points to the window object where property domesticAnimals is undefined.

This is where the arrow function comes into play. They don’t have their own this context. When used inside the outer (enclosing) function, this keyword will point to where the function is present.

In this case, this will be attached to the outer context printdomesticAnimals() where setTimeout() is called. printdomesticAnimals() will be the enclosing context where this will be attached.

let animals = {

// add property

domesticAnimals: ["cat", "dog", "cow", "goat", "sheep", "donkey", "pig", "horse"],

// add method

printdomesticAnimals: function () {

// print after 3 seconds

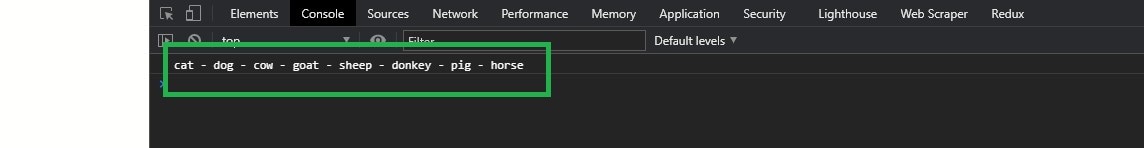
setTimeout(() => console.log(this.domesticAnimals.join(" - ")), 3000)

},

};

animals.printdomesticAnimals();

When we use the arrow function, we get the results as we expected.



#### **Arrow functions with object literal**

Let’s have an example that represents a [JavaScript object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects).

const sayName = function(){

return {

name: "Jonh Doe",

age: 26,

};

}

console.log(sayName().name);

sayName() is a function expression that returns an object that has properties name and age set to “John Doe” and “26”, respectively.

Convert to an arrow function.

const sayName = () => {

name: "Jonh Doe",

age: 26,

};

console.log(sayName().name);

You should note that when we return the literal object using the arrow function it causes an error. This is because JavaScript can’t distinguish if the curly braces represent a block of code or an object.

To solve this, wrap the literal object with parenthesis.

For example:

const sayName = () => ({

name: "Jonh Doe",

age: 26,

});

console.log(sayName().name);

### **When not to use arrow functions**

The arrow functions concept is great, however, they are not ideal across all functional instances. You should be keen on where to apply the arrow function.

For example, there are some instances that you should avoid using.

They include:

#### **Arrow function can never be a method**

For example, this applies to the mom example we explained earlier

For example:

// an object parent with the property mom\_name

const parent = {

// add property

mom\_name: "Samantha Quinn",

// create a method to return parent

mother: () => {

return `${this.mom\_name} is my mother.`;

},

};

console.log(parent.mother());

Object parent has one property, mom\_name, and one method, mother.

In this case, this.mom\_name return undefined because this value is equal to the [method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions) upon where we call the object property.

As we said earlier, this inside an arrow function is equivalent to the global object.

Whenever an arrow function is inside an object, it will derive this value from the enclosing lexical scope. Lexical scope, in this case, is the global scope.

this.mom\_name in the mother method is equal to the window.mom\_name in the web browser. The window.mom\_name is undefined by default.

Window object doesn’t have the property mom\_name. Thus console.log(mom. mother()) will return undefined as this inherits its eclosing context mother where mom\_name is undefined.

To prevent the this value from binding to the global scope, use the regular function inside the object method as follows:

// an object mom with the property mom\_name

const parent = {

// add property

mom\_name: "Samantha Quinn",

// create a method to return parent

mother: function () {

return `${this.mom\_name} is my mother.`;

},

};

console.log(parent.mother());

Avoid arrow functions when using a code block with methods. They can be confusing at times due to their lexical scoping. This occurs mostly on object methods, prototype methods, and class methods. this is scoped to the parent (window) context.

#### **An arrow function can never be a constructor**

The value this points to its parent does not have a [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor).

const User = () => {

(this.name = "John Doe"), (this.age = 20);

};

const user = new User();

console.log(user);

When you execute, this will throw an error Uncaught TypeError: User is not a constructor because arrow functions are not constructable.

However, a regular function can be constructors,

For example:

function User() {

(this.name = "John Doe"), (this.age = 20);

}

const user = new User();

console.log(user);

Using the new keyword to create an object in an arrow function will output an error.

#### **Click handlers**

Suppose I have a button CLICK ME.

Here is the code to implement a click me button.

<!DOCTYPE html>

<html lang="en">

<head>

<style>

button {

font-size: 100px;

}

.on {

background: #ffc600;

}

</style>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<button id="clicky">CLICK ME</button>

<script>

// Call the following arrow function when the button is clicked.

const button = document.querySelector('#clicky'); button.addEventListener('click', () => { this.classList.toggle('on'); });

</script>

</button>

</body>

</html>

When you click the button, you get an error. But why? This means that this in the click event handle is undefined. And will always return undefined.

Remember what we said about the arrow functions using this?. this, inside the arrow function, implies that the function does not have a this value of its own.

They don’t define their own context since it doesn’t have its own this context. They inherit that from the parent scope whenever you call this. The parent scope is the window scope. Thus this in the above example will reference the window object.

this is not attached to the element we refer to. The window object doesn’t have .classList.toggle property. Thus Javascript engine will add the .classList.toggle to the window object and set it to undefined. To fix these issues, you would use the regular function where this is bound to the element that triggers the click event.

<!DOCTYPE html>

<html lang="en">

<head>

<style>

button {

font-size: 100px;

}

.on {

background: #ffc600;

}

</style>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<button id="clicky">CLICK ME</button>

<script>

const button = document.querySelector('#clicky'); button.addEventListener('click', function() { this.classList.toggle('on'); });

</script>

</button>

</body>

</html>

#### **Debugging may not be easy**

Arrow functions cannot be named. They are anonymous. Anonymous functions are labeled as anonymous during a debugging session.

This doesn’t give you any idea what the code block is running. When you run into issues, it may be harder to debug the root cause. When the functions have names, it’s simpler to trace back to the problem. With anonymous functions, it adds a level of complexity to debugging.

#### **Readability takes a hit**

Although arrow functions help with writing short and concise code, it is not necessarily readable. Most programmers are used to the traditional way of writing functions, and arrow functions change this completely. This makes code harder to read and might take a while for someone newer to grasp the code.

Therefore in such circumstances, developers may choose to use regular functions rather than arrow functions. The primary objective when you compose a function is to create the purest function practicable. Meaning that the function would still return its same value. If you’re using regular functions or arrow functions, it doesn’t matter. It should be about writing readable and cleaner code always.

### **Final Notes**

Arrow functions save you some keystrokes when working with the functions. They are especially useful for inline functions, as they pass along the outer this context.

With an arrow function:

* No more function keywords are need, and parenthesis are optional as well as curly braces.
* Arrow functions make code shorter, more concise, and less verbose.
* The keyword return would indeed be optional. The Arrow function has an implicit return. Thus no curly braces. If you are using the curly braces, you have to use the return statement.
* Arrow functions are not attached to an identifier (the function name). To call or reuse them, your need to assign them to a variable.
* They are frequently used in callback chaining, promise chaining, array methods, and situations where anonymous (the function has no name) functions would be useful.
* They handle this operator a lot more clearly. The Arrow functions shine most whenever you need this to be attached to the context and not its own function.

***PROMISES:***

* A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is an object representing the eventual completion or failure of an asynchronous operation. Since most people are consumers of already-created promises, this guide will explain consumption of returned promises before explaining how to create them.
* Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.
* Imagine a function, createAudioFileAsync(), which asynchronously generates a sound file given a configuration record and two callback functions, one called if the audio file is successfully created, and the other called if an error occurs.
* Here's some code that uses createAudioFileAsync():
* function successCallback(result) {
* console.log("Audio file ready at URL: " + result);
* }
* function failureCallback(error) {
* console.error("Error generating audio file: " + error);
* }
* createAudioFileAsync(audioSettings, successCallback, failureCallback);
* Copy to Clipboard
* If createAudioFileAsync() were rewritten to return a promise, you would attach your callbacks to it instead:
* createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
* Copy to Clipboard
* This convention has several advantages. We will explore each one.

****

Parallel .then, sharing same results of the promise function

.then



.then



If(resolve)



If(resolve)



Promise1



If(reject)



If(resolve)



If(true)



.then

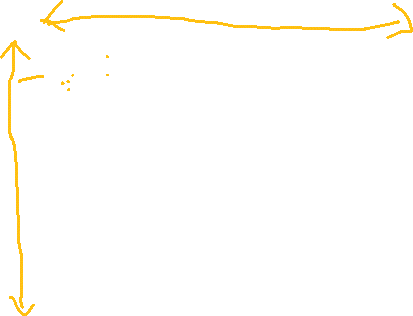
.then



.then



Multiple .then chaining



If(true)

If any error occurred,then reject is called, and then the error is later handled in catch block



.then

.Catch(alert);



Reject(new Error(“Some technical fault”);

The**fetch()**methodin JavaScript is used to request to the server and load the information in the webpages. The request can be of any APIs that returns the data of the format JSON or XML. This method returns a promise.

**Syntax:**

fetch( url, optio**ns )**

**Parameters:**This method accept two parameters as mentioned above and described below:

* **URL:**It is the URL to which the request is to be made.
* **Options:**It is an array of properties. It is an**optional**parameter.

**Return Value:**It returns a promises whether it is resolved or not. The return data can be of the format JSON or XML.  
It can be the array of objects or simply a single object.

**NOTE:**Without options Fetch will always act as a get request.

**Making Post Request using Fetch:**Post requests can be made using fetch by giving options as given below:

let options = {

method: 'POST',

headers: {

'Content-Type': 'application/json;charset=utf-8'

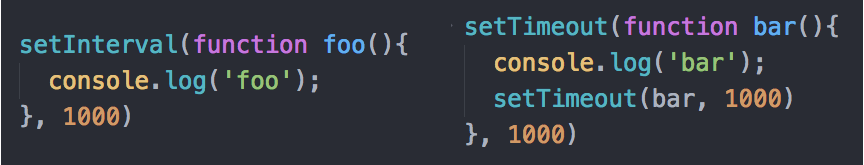
},

body: JSON.stringify(data)

}

You may be familiar with both the setTimeout and setInterval methods, which allow function calls to be scheduled after a specified delay, or repeatedly at a specified interval. While setInterval fills a useful role, it is somewhat lacking in it’s inability to modify its interval timing after its initial invocation. Which is to say, what if 1000ms is a good amount of time at first, but maybe not later? Or sometimes it is and other times it isn’t, who can anticipate the timing requirements of some future function call? setInterval will tick on forever at the same rate, heedless of my program’s current needs.

A recursive setTimeout provides another way.



Beware though, that since the subsequent setTimeout calls take place within the call back function itself, the execution context may not be able to close, and this can cause an unlimited number of contexts to be added to your call stack.

