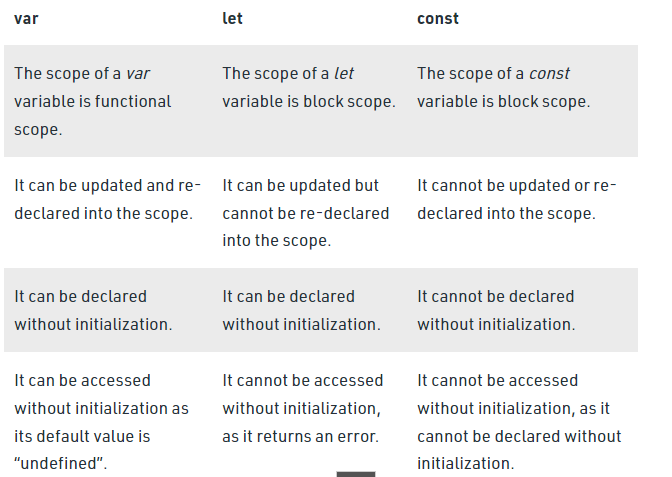
* **What is hoisting?**

JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables or classes to the top of their scope, prior to execution of the code.

* **What is scoping?**

Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are local and global: Global variables are those declared outside of a block. Local variables are those declared inside of a block.

* **How are var, let const different?**



* **What are the two main differences in arrow functions?**

this in arrow functions is defined by the enclosing lexical context. A regular object doesn't define a this local to the object. So the lookup continues outward and you get the global object. On the other hand when you use the new operator with a function it creates an object and explicitly sets this to point to that object. That is the value of this the arrow function will see because that is the value of this in the immediate lexical context.

const person = {

name: 'bbb',

// non-arrow function

getName() { console.log(this.name)}

}

person.getName();

Output:-bbb

You can see the way an arrow function will define this by looking outward to enclosing contexts by combing your examples:

const Person = function() {

this.fname = 'Bob';

this.obj = {

getName: () => { console.log(this.fname)}

}

}

const test = new Person();

test.obj.getName();

Output:- Bob

* Does Call apply bind work for arrow functions?

In case of arrow functions our methods: Call/Apply & Bind doesn’t work as expected.

As the documentation of MDN states:

"Since arrow functions do not have their own this, the methods call() or apply() can only pass in parameters. thisArg is ignored."

* **What does call apply bind do?**

The call() method is a predefined JavaScript method.

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

const person = {  
  **fullName**: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}

const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}

person.fullName.call(**person1**);

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

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

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

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  display: function () {  
    let x = document.getElementById("demo");  
    x.innerHTML = this.firstName + " " + this.lastName;  
  }  
}  
  
let display = person.display.bind(person);  
setTimeout(display, 3000);

This example will display the person name after 3 seconds.

* **What are closures?**

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.

function OuterFunction() {

var outerVariable = 100;

function InnerFunction() {

alert(outerVariable);

}

return InnerFunction;

}

var innerFunc = OuterFunction();

innerFunc(); // 100

* **Write a program to debounce a search bar?**

The **debounce() function** forces a function to wait a certain amount of time before running again. The function is built to limit the number of times a function is called.

const getSuggestions = () => {

//Calls API to get Data

console.log("Fetching Data...", counter++);

}

const debounce = function (fn, d) {

let timer;

return function () {

let context = this, args = arguments;

clearTimeout(timer);

timer = setTimeout(() => {

fn.apply(context, args);

}, d)

}

}

const debounceForData = debounce(getSuggestions, 300);

<input type="text" onkeyup="debounceForData()"/>

* **Write a program to throttle a search bar?**
* The major difference between debouncing and throttling is that debounce calls a function when a user hasn’t carried out an event in a specific amount of time, while throttle calls a function at intervals of a specified amount of time while the user is carrying out an event.
* For example, if we debounce a scroll function with a timer of 250ms (milliseconds), the function is only called if the user hasn’t scrolled in 250ms. If we throttle a scroll function with a timer of 250ms, the function is called every 250ms while the user is scrolling.

The major difference between debouncing and throttling is that debounce calls a function when a user hasn’t carried out an event in a specific amount of time, while throttle calls a function at intervals of a specified amount of time while the user is carrying out an event.

For example, if we debounce a scroll function with a timer of 250ms (milliseconds), the function is only called if the user hasn’t scrolled in 250ms. If we throttle a scroll function with a timer of 250ms, the function is called every 250ms while the user is scrolling.

let throttlePause;

const throttle = (callback, time) => {

  //don't run the function if throttlePause is true

  if (throttlePause) return;

  //set throttlePause to true after the if condition. This allows the function to be run once

  throttlePause = true;

  //setTimeout runs the callback within the specified time

  setTimeout(() => {

    callback();

    //throttlePause is set to false once the function has been called, allowing the throttle function to loop

    throttlePause = false;

  }, time);

};

* **Create a custom method for an array called myMap, use prototype chain to achieve this**

Array.prototype.myMap = function(callback) {

let newArray = [];

let x = this.length;

//console.log("x",callback)

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

let counter = callback(this[i]);

//console.log(this[i],counter)

newArray.push(counter);

}

return newArray;

};

let arr = [1, 2, 3];

arr = arr.myMap(e => e \* 5);

console.log(arr);

* **What is event bubbling?**

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:

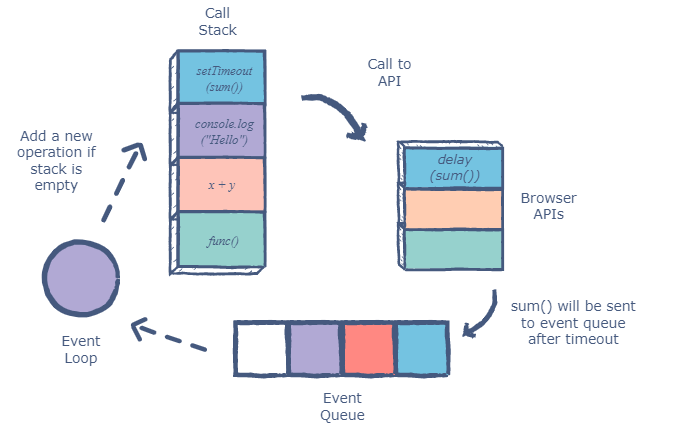
1. On that <p>.
2. Then on the outer <div>.
3. Then on the outer <form>.
4. And so on upwards till the document object.

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

* **What is event loop?**

The **event loop** is the secret behind JavaScript’s asynchronous programming. JS executes all operations on a single thread, but using a few smart data structures, it gives us the illusion of multi-threading. Let’s take a look at what happens on the back-end.

* The **call stack** is responsible for keeping track of all the operations in line to be executed. Whenever a function is finished, it is popped from the stack.
* The **event queue** is responsible for sending new functions to the stack for processing. It follows the queue data structure to maintain the correct sequence in which all operations should be sent for execution.
* Whenever an async function is called, it is sent to a browser API. These are APIs built into the browser. Based on the command received from the call stack, the API starts its own single-threaded operation.
* An example of this is the setTimeout method. When a setTimeout operation is processed in the stack, it is sent to the corresponding API which waits till the specified time to send this operation back in for processing.
* Where does it send the operation? The event queue. Hence, we have a cyclic system for running async operations in JavaScript. The language itself is single-threaded, but the browser APIs act as separate threads.
* The event loop facilitates this process; it constantly checks whether or not the call stack is empty. If it is empty, new functions are added from the event queue. If it is not, then the current function call is processed.



* **Explain promises to a 5 year old, with simple examples**

Sometimes you need to wait for Axios (a popular HTTP request library)to return a value before proceeding. A promise allows you to do that.

It lets you write asynchronously executed but predictable code

Let’s see a basic promise.

new Promise ((resolve, reject) => {  
let success = true;  
// Code  
if (success) resolve()  
else reject()  
})

The idea is that a promise will resolve if everything goes right and reject if anything goes wrong. A function usually wraps the promise.

* Write a function called sleep that will return a promise, if you do not provide a number to the function, then it will return an error and goto the catch block.

sleep(500).then(res=> {

console.log('slept for ${res} milli seconds})

})

.then(errr=>{

console.log(err)

})

* **what does async await mean?**

**Async:**

It simply allows us to write promises based code as if it was synchronous and it checks that we are not breaking the execution thread. It operates asynchronously via the event-loop. Async functions will always return a value. It makes sure that a promise is returned and if it is not returned then javascript automatically wraps it in a promise which is resolved with its value.

const getData = async() => {

    var data = "Hello World";

    return data;

}

getData().then(data => console.log(data)); //Hello World

**Await:**  
Await function is used to wait for the promise. It could be used within the async block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

const getData = async() => {

    var y = await "Hello World";

    console.log(y);

}

console.log(1);

getData();

console.log(2);

Output:-

1

2

Hello World

* What does the this keyword mean?

“This” keyword **refers to an object that is executing the current piece of code**. It references the object that is executing the current function.

* **What are classes? what are getters and setters?**

Classes are a template for creating objects. They encapsulate data with code to work on that data.

Getter-This example uses a lang property to get the value of the language property.

const person = {  
  firstName: "John",  
  lastName: "Doe",  
  language: "en",  
  get lang() {  
    return this.language;  
  }  
};  
  
// Display data from the object using a getter:  
document.getElementById("demo").innerHTML = person.lang;

Setter-This example uses a lang property to set the value of the language property.

const person = {  
  firstName: "John",  
  lastName: "Doe",  
  language: "",  
  set lang(lang) {  
    this.language = lang;  
  }  
};  
  
// Set an object property using a setter:  
person.lang = "en";  
  
// Display data from the object:  
document.getElementById("demo").innerHTML = person.language;

* **How do you declare private and static variables in classes**

A static variable is a class property that is used in a class and not on the instance of the class. The variable is stored on the data segment area of the memory, and the same value is shared among every instance created in a class. To use a static variable, we use the static keyword.

A **private variable is only visible to the current class**. It is not accessible in the global scope or to any of its subclasses.

* Create a Calculator class, it should be able to add, reduce multiply and divide. it should have a value getter, and that should return final output. keep the history of changes made as well, and keep that private, and a user should be able to see previous changes made to the value
* What is currying?

Currying is a function that takes one argument at a time and returns a new function expecting the next argument. It is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

const addCurry =(a) => {

return (b)=>{

return (c)=>{

return a+b+c

}

}

}

console.log(addCurry(2)(3)(5)) // 10

* **Write a program to flatten an array**

// input: [ 1, [ 2, 3 ], [ 3 ], [ [ [ 5]], 6] ]

// output => [ 1, 2, 3, 3, 5, 6 ]

var a=arr.join().split(",").map(Number)

console.log(a)