Save for later

ADVANCED JAVASCRIPT CONCEPTS FOR INTERVIEW

@subhajit-adhikary

001

CALLBAKCS

A JavaScript callback is a function which is to be executed after another function has finished execution.

```
//  callbacks and callback hell
let fruits = ["apple", "banana", "kiwi"];

const animateAll = (animate) => {
  // below is the callback hell
  setTimeout(() => {
    animate(fruits[0]);
    setTimeout(() => {
      animate(fruits[1]);
      setTimeout(() => {
        animate(fruits[2]);
      }, 1000);
    }, 1000);
  }, 1000);
};

const animate = (fruit) => {
  console.log("animating", fruit);
};

animateAll(animate);
```

Simply said:- Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function. This results in callback hell.

Output

```
animating apple
animating banana
animating kiwi
```

@subhajit-adhikary

JS

002

PROMISES

A promise is an object that will produce a single value sometime in the future. If the promise is successful, it will produce a resolved value, but if something goes wrong then it will produce a reason why the promise failed.

Simply said:- It behaves very much similar to real life promises.

```js
// promises
let fruits = ["apple", "banana", "kiwi"];

const animateOne = (fruit, animate) => {
  return new Promise((res, rej) => {
    setTimeout(() => {
      animate(fruit);
      res(true);
    }, 1000);
  });
};

const animateAll = (animate) => {
  animateOne(fruits[0], animate)
    .then(() => animateOne(fruits[1], animate))
    .then(() => animateOne(fruits[2], animate))
    .catch((err) => console.log("some error occurred", err));
};

const animate = (fruit) => {
  console.log("animating", fruit);
};

animateAll(animate);
```

Output

```
animating apple
animating banana
animating kiwi
```

@subhajit-adhikary

JS

003

ASYNC/AWAIT

Async/Await makes it easier to write promises. The keyword 'async' before a function makes the function return a promise, always. And the keyword await is used inside async functions, which makes the program wait until the Promise resolves.

```
// async/await
let fruits = ["apple", "banana", "kiwi"];

const animateOne = (fruit, animate) => {
  return new Promise((res, rej) => {
    setTimeout(() => {
      animate(fruit);
      res(true);
    }, 1000);
  });
};

const animateAll = async (animate) => {
  await animateOne(fruits[0], animate);
  await animateOne(fruits[1], animate);
  await animateOne(fruits[2], animate);
};

const animate = (fruit) => {
  console.log("animating", fruit);
};

animateAll(animate);
```

Output

```
animating apple
animating banana
animating kiwi
```

@subhajit-adhikary

JS

004

STRICT MODE

The "use strict" directive enables JavaScript's strict mode. This was introduced in ECMAScript 5. It enforces stricter parsing and error handling on the code at runtime. It also helps you write cleaner code and catch errors and bugs that might otherwise go unnoticed.

```
84   // strict mode
85   "use strict";
86   x=56;
87   console.log(x)
88
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Code

d:\web_dev_recap\js_prep\js_youtube_explain\js_practice.js:86
x=56;
 ^

ReferenceError: x is not defined
    at Object.<anonymous> (d:\web_dev_recap\js_prep\js_youtube_explain\js_practice.js:86:2)
    at Module._compile (node:internal/modules/cjs/loader:1376:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1435:10)
    at Module.load (node:internal/modules/cjs/loader:1207:32)
    at Module._load (node:internal/modules/cjs/loader:1023:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:135:12)
    at node:internal/main/run_main_module:28:49

Node.js v20.11.1
```

@subhajit-adhikary

JS

005

HIGHER ORDER FUNCTIONS

Functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.

Simple words:- It is a function that accepts functions as parameters and/or returns a function.

Eg:- map, filter, reduce, some, every, forEach, sort etc.

```
// Higher order function with Array.map() and Array.filter()
const ages = [23, 45, 12, 67, 18];

const doubleAges = ages.map((age) => age*2);
console.log(doubleAges);  //[ 46, 90, 24, 134, 36 ]

const ageLessThan40 = ages.filter((age) => age<40)
console.log(ageLessThan40);  // [ 23, 12, 18 ]
```

@subhajit-adhikary

JS

CALL, APPLY, BIND

Call is a function that helps you change the context of the invoking function. In layperson's terms, it helps you replace the value of this inside a function with whatever value you want.

```
// call, apply, bind
function fullName(greet) {
  console.log(greet + " " + this.firstName + " " + this.lastName);
}
const person1 = {
  firstName: "Elon",
  lastName: "Musk",
};
const person2 = {
  firstName: "Ratan",
  lastName: "Tata",
};

fullName.call(person1, "Hello"); // Hello Elon Musk
fullName.call(person2, "Hi"); // Hi Ratan Tata

fullName.apply(person1, ["Hello"]); // Hello Elon Musk
fullName.apply(person2, ["Hi"]); // Hi Ratan Tata

const person1FullName = fullName.bind(person1); // Hello Elon Musk
const person2FullName = fullName.bind(person2); // Hi Ratan Tata

console.log("execute some other code here");
person1FullName("Hello"); // Hello Elon Musk
person2FullName("Hi"); // Hi Ratan Tata
```

Apply is very similar to the call function. The only difference is that in apply you can pass an array as an argument list.

Bind is a function that helps you create another function that you can execute later with the new context of this that is provided.

Output

```
Hello Elon Musk
Hi Ratan Tata
Hello Elon Musk
Hi Ratan Tata
execute some other code here
Hello Elon Musk
Hi Ratan Tata
```

@subhajit-adhikary

JS

007

SCOPE

The scope is the current context of execution in which values and expressions are "visible".
JavaScript variables have 3 types of scope:
Block scope:- Variables declared inside a **{ }** block cannot be accessed from outside the block. let and const have block scope.

Function scope:- Variables defined inside a function are not accessible (visible) from outside the function. let, const and var have function scope.

Global scope:- Variables declared Globally (outside any function) have Global Scope. let, const and var have global scope.

let

```
// let with different scopes

// * let creates a block scope
// * Re-declaration in NOT allowed (in same scope)
// * Re-assignment is allowed

{ // block scope
    let x = 0;
    console.log(x); // 0
    let x = 1; // Error
  }

  {
    let x = 1;
    x = 2;
    console.log(x);// 2
  }

  console.log(x); // Error in Global Scope
```

@subhajit-adhikary

SCOPE

var

```
// var with different scopes

// * No block scope, and can be re-declared
// * Only had function scope
// * var are hoisted, so they can be used before the declaration

var x = 1;
var x = 2; // valid

console.log(y) // valid
var y = 3

z=4
console.log(z) // valid
var z;
```

const

```
// const with different scopes

// * const creates a block scope
// * Re-declaration in NOT allowed
// * Re-assignment is NOT allowed
// * Must be assigned at declaration time.

{ // block scope
const x; //Error
const y=0;
y=3;
}

console.log(x); // Error in global scope
```

@subhajit-adhikary

JS

CLOSURES

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).

Simple words:- A closure gives you access to an outer function's variables and functions from an inner function.

```
// closures
function createHustler(name){
    let greetHi = 'Hi ' ;
    function greet(){
        return greetHi + name + ', welcome to hustlers group';
    }
    return greet;
}

let greetFn = createHustler('Ankit');
console.log(greetFn());
```

Output

```
Hi Ankit, welcome to hustlers group
```

@subhajit-adhikary

HOISTING

A javascript mechanism where variables with var and function declarations are moved to the top of their scope before code execution.

👉 function declarations are properly hoisted (not arrow functions)

👉 var is hoisted.

```js
// hoisting
myPlace = 'Bengaluru'; // var is hoisted
console.log(myPlace);
var myPlace;

let myName = "Subhajit";
sayHi(); // valid

function sayHi() {
  let greet = "hi";
  console.log(greet, myName);
}

sayHello(); // error
let sayHello = function() {
  console.log(myName);
};
```

Output

```
Bengaluru
hi Subhajit
d:\web_dev_recap\js_prep\js_youtube_explain\js_practice.js:200
sayHello(); // error
^

ReferenceError: Cannot access 'sayHello' before initialization
```

@subhajit-adhikary

010

IIFE (Immediately Invoked Functional Expressions)

An **IIFE** is a function that is called immediately after it is defined.

Use cases of IIFE:-

👉 Avoid polluting the global namespace.

👉 Execute async/await functions.

👉 Provides encapsulation, allowing you to create private scopes for variables and functions.

```
// IIFE (Immediately Invoked Functional Expression)
const pr = () => {
    return new Promise((res, rej) => {
      setTimeout(() => {
        res("resolved");
      }, 1500);
    });
};

(async () => {
  const a = await pr();
  console.log(a);

  const b = await pr();
  console.log(b);

  const c = await pr();
  console.log(c);
})();
```

@subhajit-adhikary

011

CURRYING

It involves breaking down a function that takes multiple arguments into a series of functions that take one argument each.

Simple words:- This creates a chain of functions, where each function returns another function until the final result is achieved.

```
// currying
// basic use case
function sum(a){
    return function(b){
        return function(c){
            return a+b+c;
        }
    }
}
console.log(sum(3)(4)(5));

// real life use case of event logger with time, type, message
const logger = (time) => (type) => (message) => `At time: ${time}, an event
of type: ${type} occurred with full details as: ${message}`;

const eventsNow = logger('5am');
const errorEvent = eventsNow('error');
console.log(errorEvent('cannot set properties of null'))
```

Output

```
12
At time: 5am, an event of type: error occurred with full details as: cannot set properties
of null
```

@subhajit-adhikary

DEBOUNCING

Debouncing is a strategy used to improve the performance of a feature by controlling the time at which a function should be executed.

Simple words:- It delays the execution of your code until the user stops performing a certain action for a specified amount of time. It is a practice used to improve browser performance.

Js code

```js
// debouncing
// js file
const inputElement = document.getElementById('fruits');

function printInputText(text) {
    console.log(text);
}

function debounce(fx, delay){
    let timeoutId = null;
    return function(text){
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => {
            fx(text);
        }, delay);
    }
}

const debounceFn = debounce(printInputText, 2000);

inputElement.addEventListener('input', (event) => {
    debounceFn(event.target.value);
})
```

HTML code

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <label for="fruits">Enter your favourite fruits</label>
    <input type="text" id="fruits" name="fruits">
    <script src="./index.js"></script>
</body>
</html>
```

@subhajit-adhikary

JS

013

THROTTLING

Throttling is a mechanism that allows a function execution for a limited number of times after that it will block its execution.

Simple words:- It limits the execution of your code to once in every specified time interval. It is a practice used to improve browser performance.

Js code

```js
// throttling (js file)
let count = 0;
function printScroll() {
    count+=1;
    console.log("scroll called", count);
}

function throttle(fx, delay){
    let timeoutId = null;
    return function(){
        if(!timeoutId){
            timeoutId = setTimeout(() => {
                fx();
                clearTimeout(timeoutId);
                timeoutId = null;
            }, delay);
        }
    }
}

const throttleFn = throttle(printScroll, 2000);

document.addEventListener('scroll', (e) => {
    throttleFn();
});
```

HTML code

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./styles.css">
    <title>Document</title>
</head>
<body>
    <div class="div-element-aqua">
    </div>
    <div class="div-element-lime">
    </div>
    <script src="./index.js"></script>
</body>
</html>
```

@subhajit-adhikary

JS

POLYFILLS

- 👉 Polyfill is a way of providing futuristic API not available in browser.
- 👉 We might need to do the native prototype modifications, so that we can get a feature/API.
- 👉 There might be situations when we have a method not supported for specific browsers, in such cases we can use polyfills.

```
// polyfills
if(!Array.prototype.contains){
    Array.prototype.contains = function(searchElement) {
        return this.indexOf(searchElement)≥0 ? true : false
    }
}
```

@subhajit-adhikary

JS

Save for later

KEEP LEARNING

PS:- Remember, these tips are just the start of your advanced journey with **Javascript**. There's always more to learn and explore, so keep coding and keep growing!

Save this post for future use

Was this helpful ??

@subhajit-adhikary