**JavaScript Interview Questions**

**#1 Facts About JavaScript on Synchronous and Asynchronous?**

JavaScript is a single-threaded synchronous language which executes all the code line by line in the order in which it is written.

In asynchronous programming, you can perform tasks without waiting for a task to complete its execution. Even when a long-running task starts, the remaining program code continues to run, and when the task is complete, the program is informed and gets access to the result. In short, you can run some blocks of code in the background while the main code continues getting executed. This helps you utilize time and resources efficiently.

With the help of Web APIs we can achieve asynchronous programming in JavaScript.

**1. What is the difference between the ‘==’ and ‘===’ operator?**

Ans: Both are comparison operators in JavaScript but they differ in the scope of comparison. While ‘==’ only compares the values, ‘===’ compares both the value and the type.

For example:

var a = 5;

var b = "5";

(a == b) // Returns true since the values of both x and y are the same

(a === b) // Returns false since the typeof x is ‘number’ and typeof y is ‘string’

**2. What is implicit type coercion in JavaScript?**

Ans: Implicit type coercion is an automatic conversion from one data type to another.

There are different types of implicit type coercions of which string coercion is the most common one.

When using the ‘+’ operator to add a number to a string, the number type is implicitly changed to a string.

For example:

Var a = 5;

Var b = “5”;

a + b // Returns ‘55’

Var c = ”strings”;

a+c // Returns ‘5strings’

**3. What are static and dynamic typed languages? What type of language is JavaScript?**

**Ans:** In statically typed languages, the type of the variable is checked during compile time. The variables and values have types. Also, the variables cannot change their type.

In contrast, in dynamically typed languages, the type of the variable is checked during runtime. The variables do not have any type and therefore can change type dynamically. Values, however, have types.

The main difference between the two is that in dynamically typed languages, the type of variables can be changed even after defining them, without any errors.

For javascript:

Var a = 5;

console.log(a) // Returns number 5

a = “hello”

console.log(a) //Returns string ‘hello’

Since this can run without any errors, JavaScript is a dynamically typed language.

**4. Briefly explain the difference between Java and JavaScript.**

Ans: Java is a complete object-oriented structured programming language like C++. On the other hand, JavaScript is a client-side and server-side scripting language that can be inserted into HTML pages and is understood by browsers. Both the languages are independent of each other and designed for different purposes.

**5. What are undeclared and undefined variables?**

Ans: Undeclared variables are not declared anywhere; therefore, the program has no knowledge of their existence. They are variables that do not exist in the program at any place. If the program runs into any such variable, a runtime error throws up.

On the other hand, undefined variables are actually defined in the program but not given any value. These variables do actually exist in the program. An undefined value is returned if the program reads the value of such variables.

**6. What is the meaning of null in JavaScript? How is it different from undefined and 0?**

Ans: Null means non-existent or empty. It has to be assigned. While undefined means that the variable is declared but no value has been assigned to it, null means value has been assigned to it but is empty. Also, undefined is of type ‘undefined’ while null is an object.

Null is also different from 0 as though it means empty, 0 is a numeric value while null is not.

**7. What are case-sensitive languages? Is JavaScript case sensitive?**

Ans: Case-sensitive languages are those that treat the variables and function names differently if upper or lower case is used.

For example, var A and var a are treated differently.

JavaScript is a case-sensitive language.

**8. Explain anonymous functions and how are they different from regular functions?**

Ans: Anonymous functions are functions without names. They are often inaccessible after the initial creation.

For example:

Var print = function(){

console.log(“print is called”);

}

print(); // displays “print is called”

In this example, the function has been given no name. Since we need to call it later, we simply assigned its value to a variable.

**9. What are JavaScript objects? How do objects store data?**

Ans: Objects are just a collection of variables lumped together in key-value pairs. They store their elements as key-value pairs. Each key-value pair is termed as a property. Objects are used to store information about almost every single entity. Objects are custom data types in JavaScript as they depend upon the user who decides what value to enclose within them.

For example:

var hotel = {

numberOfRooms : 12,

location : "Pune",

stars : 3

}

**10. Briefly explain object methods.**

Ans: In JavaScript, objects generally store information in the form of key-value pairs. However, it is possible to assign a function instead of a value to a key.

A method is a function that is assigned to a key in an object.

For example:

var author = {

name: "Alan Moore",

books: function() {

console.log("Alan Moore is famous for his illustrated novels.");

}

}

author.books(); //Returns ‘Alan Moore is famous for his illustrated novels.’

**11. Pass by Value && Pass by Reference?**

Ans: In JavaScript any primitive (String, Integer, Float, Boolean etc.) value we pass to function argument by default it passed to the function as a pass by value type. But when we pass any non-primitive (Object, Array, etc.) to function argument by default it passed as reference to the function.

Let’s understand with example:

**1. Pass by Value**

//Declare a function

function foo(x) {

console.log(x)

}

//Declare a variable

var name = ‘Jyotiraditya’

foo(name) -> o/p: Jyotiraditya

function foo(x) {

console.log (‘before’ + x)

x = ‘JDM’

console.log (‘after’ + x)

}

foo(name) ->

o/p: before Jyotiraditya

after JDM

now if we try to print name variable even after changing, it will not change

name -> Jyotiraditya

2. Pass by Reference

//Declare a function

function bar(x) {

console.log ('before' + ” “ + x.name)

x.name = 'JDM'

console.log ('after' + “ “ + x.name)

}

var person = {

name: 'Jyotiraditya'

}

bar (person) ->

o/p: before Jyotiraditya

after JDM

now if we try to print the person object we will see change,

person ->

{name: ‘JDM’}

Very Important: Let’s say that there are two variables 'a' and 'b' are of non-primitive type, i.e., object. The value contained in the variable 'a' is assigned to the variable 'b'. Next, the object 'a' is entirely changed. Thus, the variable 'a' now points to a completely different value. This does not result in any change in the value contained in the variable 'b'. Had the change been partial, wherein the value of the object held by the variable 'a' was changed, the same change would have been reflected in the variable 'b'. Remember that changing the value of a variable never changes the underlying primitive or object or array, it just points the variable to a new primitive or object or array.\

Example:

**var** a = {key1: **1**, key2: **2**};

**var** b = a;

a = {newKey1: **10**, newKey2: **20**};

console.log(a, b);

O/P

{newKey1: **10**, newKey2: **20**}

{key1: **1**, key2: **2**}

Pass by Value

The illustration given below depicts what happens behind the scenes when you assign the value contained in the variable 'a' to the variable 'b'. Note that both the variables are of primitive type.

Diagram

Description automatically generated

Pass by Reference

The illustration given below depicts what happens behind the scenes when you assign the value contained in the variable 'a' to the variable 'b'. Note that both the variables are of custom type (object / array).

Diagram

Description automatically generated

Remember:

1. All the **primitive data types** in JavaScript are **passed by value** and the **custom types such as objects and arrays are passed by reference**.
2. When a variable refers to an object or an array, the 'value' contained in the variable acts as a **reference** to the object/array.
3. Changing the **value of a variable** never changes the **underlying primitive or object or array**; it just points the variable to a new primitive or object or array.
4. Changing a **property of an object or array** referenced by a variable changes the property in the referenced object or array.

**12. What is Scope in JavaScript?**

Scope provides code security and helps to debug the code faster and more efficiently. There are two types of scope in JavaScript.

1. Local Scope – Anything inside a function is in its local scope, its lifetime is within the function.
2. Global Scope – Anything outside a function is in global scope, its lifetime is till the browser window or tab is closed

**Local Scope:**

var x = 10;

function foo(){

var x = 20

console.log(x)

}

foo()

O/P- 20. -> Here x is redefined as local variable to that function, so the scope is local

console.log(x)

O/P- 10 -> Here it is printed 10 because initially we declare a variable called x as a global variable, so the scope is global.

**Global Scope:**

var x = 10

function foo () {

console.log(x)

}

foo ()

O/P- 10 ->Here x is defined in global scope, thus it is printing 10.

**Advance Concept:**

function f1() {

var p = 20

f2()

}

function f2() {

console.log(p)

}

f2()

O/P - Uncaught Reference Error: p is not defined.

It is because we are trying to access the variable p, which is defined inside the function f1(), and we are

trying to access it inside f2()

Now if we write the code like this, we will be able to get the desired o/p

function f1() {

var p = 20

f2()

function f2() {

console.log(p)

}

} O/P -> 20

Ex.2

**function** alpha() {

**var** x = **3**;

gamma();

**function** beta() {

**function** gamma() {

**var** y = **9**;

console.log(x); // statement 1

console.log(y); // statement 2

}

}

}

alpha();

Uncaught Reference Error: gamma is not defined

Uncaught Reference Error: gamma is not defined



**Comments:**

In the given code snippet, the alpha() function is called. Inside the definition of this function, gamma() function is called. Now, if you observe carefully, the gamma() function is defined inside the beta() function and is not directly inside the alpha() function. Thus, due to an upper layer of the beta() function, the gamma() function cannot be directly accessed inside the alpha() function. Now, when the gamma() function is called inside the alpha() function, a reference error is thrown saying 'gamma is not defined'. This happens for both the statements, when executed individually, as mentioned in the question. Thus, this option is the correct choice.

**13. var, let, & const keyword?**

Ans: **var** is global scope, where are **let** and **const** both are block scoped.

**var:**

console.log(d)

var d = 10

O/P- undefined

undefined

**Imp – var hoisted on top of the scope**

**let or const:**

console.log(d)

let d = 10

O/P- Reference error, d is not defined.

**Imp – let & const are not hoisted on top of the scope**

The variable declared using **const** keyword, can’t change its value, but a variable declared using **let** keyword can change its value. The variable declared using

Let’s dig into more details:

const passengerBus = {

wheels: 8,

seatingCapacity: 40

}

passengerBus //Trying to print, I go the desire output

O/P- {wheels: 8, seatingCapacity: 40}

passengerBus.seatingCapacity = 50 //Manipulate the variable of the passengerBus object

passengerBus //Now, trying to print, I got the changed o/p

{wheels: 8, seatingCapacity: 50}

So, the objects (including arrays & functions) in a summary **(non-primitive)** declared using the const keyword are **mutable**. You can **add/delete/update** a property inside an object declared using the **const** keyword.

Similar things can be done in array too.

const android = ['Marshmallow', 'Noughat', 'Oreo']

arr[3] = 'Pie' // adding new version

console.log(android);

O/P- ['Marshmallow', 'Noughat', 'Oreo', 'Pie']

**14.What is Prototype chain?**

Ans: The \_\_proto\_\_ property either refers to an object or is null.

Each object has a private property, which holds a link to another object called its prototype. This prototype object has a prototype of its own, and this prototype chain continues until an object is reached with null as its prototype. By definition, null has no prototype and acts as the final link in this prototype chain. Almost all objects in JavaScript are instances of Object, which sits on the top of a prototype chain.

**15. What is Prototypal Inheritance?**

Ans: Assigning the prototype of an object to another object, thereby making the latter access the resources of the former. This is known as prototypal inheritance.

**16. Class in JavaScript?**

Ans: Classes are in fact 'special functions'. When you check for the data type of a class in JavaScript, you will get a function in return.

A class can derive all the properties and methods of another class. This means that a child class can inherit the properties and methods of its parent class using the ‘extends’ keyword. An important point to note is that when a child class is implementing a parent class, the super() function should be called inside the constructor of the child class. This function will call the parent class constructor first to derive the properties and methods of the parent class. Also, remember that you need to call the super() function before using the ‘this’ keyword inside a class constructor. All the classes in JavaScript extend the Object class by default.

Using classes is an alternative to the prototypal inheritance achieved using the prototype chain.

**class** Animal {

constructor() {

**this**.moves = **true**;

}

}

**class** Rabbit **extends** Animal {

constructor() {

**super**();

**this**.eats = **true**;

}

}

**let** rabbit = **new** Rabbit();

console.log(rabbit.moves);

**17. What is this keyword?**

Ans: ‘this’ refers to the current instance. The ‘this’ keyword points towards the global window object in the context of browsers when written inside a regular function call. However, in the case of a method, which is defined as a function inside an object, the ‘this’ keyword points towards the object to which the method is bond.

**var** operation = {

a: **1**,

b: **2**,

add: **function**() {

**this**.a + **this**.b;

console.log("a = " + **this**.a + ", b = " + **this**.b);

**var** print = **function**() {

console.log("a = " + **this**.a + ", b = " + **this**.b);

}

print();

}

}

operation.add();

O/P

a = 1, b = 2

a = undefined, b = undefined

Another example of this:

let person = {

office:'AArete',

location:'Pune',

set: function (name){

this.name = name

},

get:function(){

*return* this.name

}

}

person.set('Jyotiraditya')

person.get() 🡪 It will print Jyotiraditya

**18. What is bind() method in JavaScript?**

Ans: bind() method actually binds the passed scope to the function.

Two important points to be noted about the bind() method are as follows:

1. You can pass any context to the bind() method. context should be your first argument of the bind() method. Whatever is passed as the context, the ‘this’ keyword inside the function to which the bind() method is applied, starts pointing to the passed context.
2. You can pass other arguments to the bind method, however note that first method must be always your context, the remaining arguments can be anything that the bind() method takes and the function, to which the bind method is applied.

**19. What is Arrow function in JavaScript?**

Ans: It is being introduced in ES6 or ES2015. It has much smaller syntax.

There are differences between arrow functions and traditional functions, as well as some limitations:

* Arrow functions don't have their own bindings to this or super, and should not be used as methods.
* Arrow functions don't have access to the ‘new.target’ keyword.
* Arrow functions aren't suitable for ‘call’, ‘apply’ and ‘bind’ methods, which generally rely on establishing a scope.
* Arrow functions cannot be used as constructors.
* Arrow functions cannot use yield, within its body.

**20. Tell me about Array.map()?**

Ans: The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.

*//\* Map*

const array1 = [1, 4, 9, 16];

*// pass a function to map*

const map1 = array1.map(x => x \* 2);

console.log(map1);

Thus, the map() method is used when you wish to apply a common function to all the elements in the given array. The function takes a parameter, which applies to each element in the array. The resultant value is an array containing all the elements after the given function has been applied to each original element.

**21. About Array.filter()?**

Ans: The filter() method creates a new array with all elements that pass the test implemented by the provided function. It is used to filter out arrays based on conditions.

*//\* Filter*

const arr = [1, 2, 3, 4, 5]

console.log(arr.filter(function (x){

*return* x > 2

}))

**22. About Array.reduce()?**

Ans: The final result of running the reducer across all elements of the array is a single value. It means it returns single value.

console.log(arr.reduce(function(x, y) {

console.log('X ' + x + ' ' +'Y '+ y)

*return* x \* y

}))

**23. Template Literals?**

Ans: Template literals are literals delimited with backtick (`) characters. It also uses in String interpolation. Using Template literals you can write multi line string without \n

*//! Template Literals*

let message = `Hey, I am template literals as I have enclosed with backticks(')`

*//?string interpolation using template string*

let fname = 'Jyotiraditya'

let lname = 'Dhalmahapatra'

console.log(`My name is ${fname} ${lname}`)

*//? multiline using template literals*

let interest = 'code'

let occupation = 'coder'

console.log(`My name is ${fname} ${lname}

I like to ${interest}

Docs dekh ke BA samjhe kya, ${occupation} hai mein :)`)

**24. What is closure?**

Essentially, a closure is an inner function that has access to the outer (enclosing) function’s resources due to the scope chain, wherein a child can access all the resources of its parent.

It is the combination of a function and the lexical environment in which the function was declared. This closure function has access to all the local variables that were declared inside this lexical scope when the closure function was created.

*//Closure*

const showMessage = (name) =>{

let greeting = 'Hi ' + name

*return* printName = () =>{

console.log(greeting)

}

}

let sayHello = showMessage('Jyotiraditya')

console.log(greeting) *//This will throw error, as greeting is not defined the inner function scope*

sayHello()

Closures are important in situations when you require the inner function to access the outer function’s variables (or resources) as long as the inner function wants, even when the outer function has finished executing.

Let’s understand this in an example:

const modifyItems = () =>{

let items = 0;

*return* {

add: () =>{

console.log('Items added to the cart')

items++;

},

remove: () =>{

console.log('Item remove from the cart')

items--;

},

getCount: () =>{

console.log(`Total count = ${items}`)

}

}

}

let item = modifyItems()

item.add()

item.add()

item.add()

item.getCount()

item.remove()

item.getCount()

**O/P**

Items added to the cart

Items added to the cart

Items added to the cart

Total count = 3

Item removed from the cart

Total count = 2

**25. What is call() and apply() method?**

Ans: It is similar to the bind() function. Let’s understand the bind function first.

A bind() method binds the passed scope to the function.

Two important points to be noted about the bind() method are as follows:

1. You can pass any context to the bind() method. context should be your first argument of the bind() method. Whatever is passed as the context, the ‘this’ keyword inside the function to which the bind() method is applied, starts pointing to the passed context.
2. You can pass other arguments to the bind method, however note that first method must be always your context, the remaining arguments can be anything that the bind() method takes and the function, to which the bind method is applied.

Example:

Let’s say we have an object like this below:

let company = {

organisation: 'AArete',

region: 'Pune',

pin: 411027,

get: function (){

console.log('The company ' + this.organisation + ' located at ' + this.region + ' has pincode ' + this.pin)

let print = function(){

console.log('The company ' + this.organisation + ' located at ' + this.region + ' has pincode ' + this.pin)

}

print()

}

}

company.get()

here the O/P will be :

The company AArete located at Pune has pincode 411027 // Outer Console

The company **undefined** located at **undefined** has pincode **undefined** //Inner Console

To achieve the same result we can use bind() method and passed the first argument as the current context.

So the code will look like below:

let company = {

organisation: 'AArete',

region: 'Pune',

pin: 411027,

get: function (){

console.log('The company ' + this.organisation + ' located at ' + this.region + ' has pincode ' + this.pin)

let print = function(){

console.log('The company ' + this.organisation + ' located at ' + this.region + ' has pincode ' + this.pin)

}.bind(this)

print()

}

}

company.get()

Although, you can fix the above problem by using Arrow function. Just need to make the inner function as arrow function.

*//? Using Arrow method*

let newCompany = {

organisation: 'AArete',

region: 'Pune',

pin: 411027,

get: function (){

console.log('Outer : '+ 'The company ' + this.organisation + ' located at ' + this.region + ' has pincode ' + this.pin)

let print = () => {

console.log('Inner: ' + 'The company ' + this.organisation + ' located at ' + this.region + ' has pincode ' + this.pin)

}

print()

}

}

newCompany.get()

**b. call() method**

It is similar to bind method, but the only difference here is in case, of bind() method you need to explicitly call the method where bind method is applied, where in case of call() method the method gets called implicitly where call() method is applied.

Example:

let person = {

firstName: 'Jyotiraditya',

lastName: 'Dhalmahapatra',

get: function(){

console.log(`Outer: ${this.firstName} ${this.lastName}`)

let print = function (type, city){

console.log(`Inner: ${this.firstName} ${this.lastName}. I am a ${type} Developer. I am from ${city}`)

}.call(this, 'Fullstack', 'Bhubaneswar')

}

}

person.get()

**c. apply() method**

It is similar to call() method. Well, the only difference between the call() method and apply() method is that the call() method accepts arguments passed individually separated by a comma, On the other hand, the apply() method is called with an array of arguments.

*//\* apply()*

let person = {

firstName: 'Jyotiraditya',

lastName: 'Dhalmahapatra',

get: function(){

console.log(`Outer: ${this.firstName} ${this.lastName}`)

let print = function (type, city){

console.log(`Inner: ${this.firstName} ${this.lastName}. I am a ${type} Developer. I am from ${city}`)

}.apply(this, ['Fullstack', 'Bhubaneswar'])

}

}

person.get()

**26. Spread Operator?**

Ans:

*//! Spread Operator*

let getSum = (a, b, c) =>{

console.log (a + b + c)

}

let arr = [1, 2, 3]

getSum.apply(null, arr) *//using apply method*

getSum(**...**arr) *//using spread operator*

*//! another use case of using spread operator*

*//! in concatenating two arrays*

let a = [1, 2, 3]

let b = [4, 5, 6]

console.log([**...**a, **...**b])

*//! another use case of using spread operator*

*//! to make copy of Arrays*

let animal = ['Tiger', 'Babbon', 'Bear']

let newAnimal = [**...**animal] *//value copy, Deep copy*

*//let newAnimal = animal //Reference copy, Shallow copy*

console.log('Before')

console.log(newAnimal)

console.log(animal)

newAnimal.push('Elephant')

newAnimal.push('Deer')

console.log('After')

console.log(newAnimal)

console.log(animal)

*//! another use case of using spread operator*

*//! to make copy of Objects*

let petAnimal = {

type:'Dog',

breed:'Labrador'

}

let newPetAnimal = {**...**petAnimal} *//value copy, Deep copy*

*//let newPetAnimal = petAnimal //Reference copy, Shallow copy*

console.log('Before')

console.log(newPetAnimal)

console.log(petAnimal)

newPetAnimal.name = 'Bruno'

console.log('After')

console.log(newPetAnimal)

console.log(petAnimal)

*//? Practice Problems*

const mergeStudents = (classA, classB) => {

*// TODO 1: Create a variable with identifier mergedClass*

*// TODO 1: Variable mergedClass must contain the classA's students, then two students "Emily" and "Frank", and then class B's students*

let mergedClass = [**...**classA, 'Emily', 'Frank']

mergedClass = [**...**mergedClass, **...**classB]

*// TODO 2: Print the variable mergedClass on the console*

console.log(mergedClass)

}

*// arrays containing names of students in two classes*

const classA = ["Alice", "Bob", "Catherine", "Dan"];

const classB = ["George", "Jane", "Katy", "Luis"];

mergeStudents(classA, classB);

**27. Rest Parameters**

Ans: Well, this is opposite what the spread operator does. It helps sending individual parameters as functions and receiving them as an array inside the function. By doing this, you need not worry about how many arguments the user passes inside a function.

let total = function (**...**arr){

let totalSum = 0;

arr.forEach(items => totalSum += items)

*return* totalSum

}

console.log(total(2, 4, 6, 8, 10))

**O/P**

30

Another Rest Parameters example

const multiplication = (multiplier, **...**mularr) => {

*// for(let i=0; i<mularr.length; i++){*

*// mularr[i] \*= multiplier*

*// }*

let newArray = mularr.map(items => multiplier \* items)

console.log(newArray)

}

multiplication(2, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

O/P:  
(10) [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

**28. De-structuring**

Ans: There are mainly 2 types of de-structuring in JavaScript

Array De-structuring:

const mobile = ['+91', '9663', '659623']

const [countrycode, operator, subscriber] = mobile

console.log(countrycode)

console.log(subscriber)

Object De-structuring:

*//!Object De-Structuring*

const customer = {

firstName: 'Jyotiraditya',

lastName: 'Dhalmahapatra',

country: 'India'

}

*//use the exact keyword what is there in the object*

const {firstName, lastName} = customer

console.log(firstName)

console.log(lastName)

*//incase, you need to use a separate key, then use alias syntax*

const {firstName: fn, lastName: ln} = customer

console.log(fn)

console.log(ln)

**29. What is Event Loop in JavaScript?**

Ans: 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(stack and queue), it gives us the illusion of multi-threading.

Diagram

Description automatically generated

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.

**30. What is Passing function as an argument in JavaScript?**

Ans: It means we can pass the functions as an argument; it helps us to write generic methods, and don’t need to change the business logic method.

Let’s understand this with an example:

*//!Passing Function as an argument*

const sum = (x, y) =>{

*return* x + y

}

const mulitply = (x, y) =>{

*return* x \* y

}

const getResult = (x, y, operate) => {

*return* operate(x, y)

}

const value = getResult(20,30, mulitply)

console.log(value)

O/P: 600

**30. What is callback function?**

Ans: A callback function is a function passed into another function as an argument.

Let’s understand with some examples:

function greeting(name) {

alert('Hello ' + name);

}

function processUserInput(callback) {

var name = prompt('Please enter your name.');

callback(name);

}

processUserInput(greeting);

Ex2

const sum = (x, y) =>{

*return* x + y

}

const mulitply = (x, y) =>{

*return* x \* y

}

const getResult = (x, y, operate) => {

*return* operate(x, y)

}

const value = getResult(20,30, mulitply)

console.log(value)

Let’s look into some more examples:

The following code, which you saw in the video above, showcases a callback written as an anonymous inline function and passed as an argument to another function:

Code snippet

let add = (callback) => {

let x = 2, y = 3;

console.log("Sum :", x + y);

callback();

}

add(function() {

console.log("Finished this operation!");

});

const subtract = (callback) => {

let x = 2, y = 3;

console.log("Difference :", x - y);

callback();

}

subtract(function() {

console.log("Finished this operation!");

});

O/P:

Sum : **5**

Finished **this** operation!

Difference : -**1**

Finished **this** operation!

You also learnt how a callback can be written as an anonymous independent function and passed as an argument to another function. You saw the following code in the video above for the same:

let add1 = (callback) => {

let x = 2, y = 3;

console.log("Sum :", x + y);

callback();

}

const subtract1 = (callback) => {

let x = 2, y = 3;

console.log("Difference :", x - y);

callback();

}

const displayCompletion = () => {

console.log("Finished this operation!");

}

add1(displayCompletion);

subtract1(displayCompletion);

Sum : **5**

Finished **this** operation!

Difference : -**1**

Finished **this** operation!

Some real-time examples:

let getName = (callback) => {

*// get name from DB*

let name;

setTimeout(() => {

name = "Jyotiraditya";

callback(name);

}, 2000);

}

let greet = (name) => {

console.log(`Hello ${name}`);

}

getName(greet);

O/P

Hello Jyotiraditya

**31. What is Promise?**

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

A promise is one of these 3 states:

**Pending**: Initial state, neither fulfilled nor rejected

**Fulfilled**: meaning the operation was completed successfully.

**Rejected**: meaning that the operation failed.

* There are two types of code in asynchronous programming, which are as follows:
* **Producer code:** This code produces a certain result.
* **Consumer code:** This code consumes the result produced by the producer code.
* A **promise** is an object which makes the result produced by the **producer** code available to the **consumer** code, thereby linking them together.
* The **producer** code is contained inside the **promise** object.
* The **producer** code, which is inside the **promise** object, contains the **'resolve'** and **'reject'** callbacks.
* The producer code is executed as soon as the promise object is created. **You need not explicitly call the producer code.**

**Promise Syntax:**

**let** promiseObj = **new** Promise((resolve, reject) => {

// producer code

});

**32. What all are the properties of Promise?**

Ans: 1. The **internal properties** of an object are the ones that cannot be directly accessed or manipulated. Also, the internal properties are enclosed in ***double square brackets [[]]***.

2.  A promise object consists of two internal properties, which are as follows:

* **PromiseStatus**
* **PromiseValue**

3.  When a promise object is created, the **[[PromiseStatus]]** property is initialised with ***‘pending’****,* and the **[[PromiseValue]]** property is initialised with ***undefined***.

**33. How Producer works in Promise?**

Ans: The result produced by the producer code may either be a success or a failure, both of which can be summarised as follows:

* **Success**  
  When the result of the producer code is a success, the **resolve()** callback is invoked to resolve the promise object.
* **Failure**  
  When the result of the producer code is a failure, the **reject()**callback is invoked to reject the promise object.

Text

Description automatically generated

Let’s look into some example:

*//! Promise Syntax*

let promiseObj = new Promise((resolve, reject) => {

*//!Producing Object*

console.log('Producing code is executed automatically as soon as the promise object is created')

console.log('Getting the name from DB....')

setTimeout(() => {

resolve('Jyotiraditya')

}, 3000)

})

console.log(promiseObj)

O/P:

Producing code is executed automatically as soon as the promise object is created

Getting the name from DB....

Promise {<pending>}[[Prototype]]: Promise

[[PromiseState]]: "fulfilled"

[[PromiseResult]]: "Jyotiraditya"

*//! Promise Syntax*

let promiseObj = new Promise((resolve, reject) => {

*//!Producing Object*

console.log('Producing code is executed automatically as soon as the promise object is created')

console.log('Getting the name from DB....')

setTimeout(() => {

reject('Error getting name form DB')

}, 3000)

})

console.log(promiseObj)

O/P:

Producing code is executed automatically as soon as the promise object is created

Getting the name from DB....

Promise {<pending>}[[Prototype]]: Promise

[[PromiseState]]: "rejected"

[[PromiseResult]]: "Error getting name form DB"

Uncaught (in promise) Error getting name form DB

**\*IMP: At a time this is not possible to call two call back one after another. If written It will only execute the first callback and come out of promise object.**

**34. How Consumer works in Promise?**

Ans: There are two types of consumer code in Promise.

1. then() method
2. catch() method

let’s understand **then()** method first

Syntax:

promiseObj.then((val) => {

*//!Code to be executed on Resolved callback*

}, (err) => {

*//!Code to be executed on Reject callback*

})

Example of success callback:

let promiseObj = new Promise((resolve, reject) => {

*//!Producing Object*

console.log('Producing code is executed automatically as soon as the promise object is created')

console.log('Getting the name from DB....')

setTimeout(() => {

resolve('Jyotiraditya')

}, 3000)

})

console.log(promiseObj)

promiseObj.then((val) => {

console.log(`Name received from the DB = ${val}`)

}, (err) => {

console.log(`error occurred: ${err}`)

})

O/P:

Producing code is executed automatically as soon as the promise object is created

Getting the name from DB....

Promise {<pending>}

[[PromiseState]]: ”Fulfilled”

[[PromiseValue]]: ”Jyotiraditya”

Name received from the DB = Jyotiraditya

Example of Reject callback:

let promiseObj = new Promise((resolve, reject) => {

*//!Producing Object*

console.log('Producing code is executed automatically as soon as the promise object is created')

console.log('Getting the name from DB....')

setTimeout(() => {

reject('Error getting name form DB')

}, 3000)

})

promiseObj.then((val) => {

console.log(`Name received from the DB = ${val}`)

}, (err) => {

console.log(`error occured: ${err}`)

})

O/P:

Producing code is executed automatically as soon as the promise object is created

Getting the name from DB....

Promise {<pending>}[[Prototype]]:

Promise[[PromiseState]]: "rejected"

[[PromiseResult]]: "Error getting name form DB

error occured: Error getting name form DB

Now let’s understand the catch() method

Syntax:

promiseObj.**catch**(errorCallback);

promiseObj.**catch**((parametersToHoldArgumentsPassedWhenPromiseIsRejected) => {

// code to execute when promise is rejected

});

Now Simplifying:

promiseObj.**catch**((err) => {

// code to be executed when promise is rejected

});

The catch() consumer has only one argument.

*//! Promise with Approach 2*

let promise = new Promise((resolve, reject) => {

console.log("Getting name from the DB....")

setTimeout(() => {

*//resolve("Jyotiraditya")*

reject(new Error("Failed to fetch the name from the DB"))

}, 2000)

})

promise.then((val) => {

console.log(`Name received from the DB = ${val}`)

})

.catch((err) => {

console.log(`error occured: ${err}`)

})

**35. async and await keyword in JavaScript**

Ans: The async and await keywords were introduced in ES8 (ES2017) and are internally based on promises but make the code even more readable than promises.

When the keyword async is prepended to a function, it can be safely assumed that a promise is returned from that function.

*//! Using Async and Await*

let promise = new Promise((resolve, reject) => {

console.log('Fetching name from the DB...')

setTimeout(() => {

*// resolve('Jyotiraditya')*

reject(new Error(`couldn't fetch name from DB!`))

}, 3000)

})

let getname = async() => {

try {

let name = *await* promise

console.log(`Name recieved: ${name}`)

} catch (err) {

console.log(`Error occurred: ${err}`)

}

}

getname()