**1. What is React?**

React(aka React.js or ReactJS) is an open-source front-end JavaScript library that is used for building composable user interfaces, especially for single-page applications. It is used for handling view layer for web and mobile apps based on components in a declarative approach.

React was created by Jordan Walke, a software engineer working for Facebook. React was first deployed on Facebook's News Feed in 2011 and on Instagram in 2012.

**2. What is state in React?**

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders. It is always recommended to make our state as simple as possible and minimize the number of stateful components.

**3. What are props in React?**

Props are inputs to components. They are single values or objects containing a set of values that are passed to components on creation similar to HTML-tag attributes. Here, the data is passed down from a parent component to a child component.

The primary purpose of props in React is to provide following component functionality:

* Pass custom data to your component.
* Trigger state changes.
* Use via this.props.reactProp inside component's render() method.

**4. What is the difference between state and props?**

**Props State**

* Props are used to pass data from a parent component to a child component | # state is used to manage data within a component
* Props are immutable and cannot be changed within a component | # state is mutable and can be updated using the setState function
* Props are passed down from the parent component and are read-only within the child component | # state is only accessible within the component where it is defined
* Props can be used to customize the behavior or appearance of a component | # state is used to keep track of information that can change over time

**Questions from** [**https://www.codingninjas.com/studio/library/top-react-hooks-interview-questions**](https://www.codingninjas.com/studio/library/top-react-hooks-interview-questions)

**1. What is React?**

React is an efficient, flexible, open-source front-end javaScript library created by FaceBook in 2011. It is a tool for building complex and reusable user interfaces, especially for single-page applications. It follows the component-based approach.

**2. What are the important features of React?**

The following are React's important features:

* It supports server-side rendering.
* It follows the component-based approach.
* It follows unidirectional data flow.
* It uses Virtual DOM rather than real DOM.
* React uses JSX, a combination of basic HTML and JavaScript.

**3. What are the advantages of React?**

The advantages of React are as follows:

* **Easy to Learn and Use:** React is easy to learn and use, as it uses JSX, a combination of basic HTML and JavaScript with some addition.
* **Reusable Components:** React follows the components-based approach to building mobile and web applications, and an application consists of multiple components. You can reuse these components throughout the application.
* **Unidirectional Data Flow:** React follows the unidirectional data flow while designing a react app. It becomes easier to debug errors and identify where the problem occurs.
* **Use of Virtual DOM:** React uses the Virtual DOM instead of Real DOM. Virtual DOM compares the component's previous states and updates only the changed items in Real DOM instead of updating all the components again. Thus it makes the web application faster.
* **SEO Friendly:** React allows server-side rendering, which boosts the SEO of an application.

**4. What are the limitations of React?**

Following are the limitations of React:

* The first thing is that React is not a fully-fledged framework. It is just a JavaScript library.
* The beginner programmer might face difficulties understanding React fully.
* There are numerous components in React, which might take time to grasp all the benefits fully.
* The coding might become complex using JSX.

**5. What do you understand by React hooks?**

Hooks are the newly added features in React v16.8. They are in-built functions that allow the developers to use state and life cycle methods within the components in React. They allow you to use all React features without writing a class component. Using Hooks, we can extract the stateful logic from a component so it can be tested independently and reused. This makes it possible to share Hooks among many components.

**6. What are the benefits of using React hooks?**

Following are some of the benefits of using React Hooks:

* If you use React Hooks, you can code in React without using classes.
* You can easily test and reuse existing states in your code using hooks.

**7. Will react hooks work inside the class component? What do you think?**

No, react hooks won’t work inside the class component.

**8. What is the Virtual DOM?**

The Virtual DOM in react a concept, where a virtual representation of the real DOM is kept inside the memory and is synced with the real DOM by a library such as ReactDOM.

**9. What are the differences between class components and functional components?**

| **Points** | **Functional Components** | **Class Components** |
| --- | --- | --- |
| **Declaration** | These are nothing but JavaScript Functions. So, you declare it in the same manner as the JavaScript function. | On the other hand, class components are declared using the ES6 class. |
| **Handling Props** | Handling props is very straightforward. You can use any prop as an argument to a functional component that can be directly used inside HTML elements. | For class components, the props are handled differently. Here, we make use of the **“this”** keyword. |
| **Handling State** | Functional components use react hooks for handling state. | For the class components, we can't use the hooks, so for this case, for handling the state, we make use of a different syntax. |
| **Constructor** | For the functional components, constructors are not used. | For the class components, constructors are used for storing the state. |
| **Render Method** | In the functional component, there is no use of the render() method. | In the class component, it must have the render() method. |

**10. What do you know about React props?**

In react, the prop stands for "Properties". They are plain JavaScript objects. Every React Component accepts a single object argument, known as the props. You can pass these props from one component to another using HTML attributes, and the accepting components accept these props as an argument. One thing to note is that props are immutable, which means they can be changed or manipulated inside a component.

**11. What do you understand by React state?**

In React, the states are plain JavaScript objects. The state is used to contain data or information about the component. The state of a component can change over time. Whenever the state changes, it re-renders.

**12. What is the difference between state and props in React?**

Both the state and props are JavaScript objects. They are different in their functionality regarding the component.

| **State** | **Props** |
| --- | --- |
| **States** are managed within the component, similar to variables declared within a function. | **Props** get passed to the component similar to function parameters. |
| State is mutable (that is it can be modified). | Props are immutable (that is they cannot be modified). |
| You can read as well as modify states. | Props are read-only. |

**13. What is the use of useState() in React Hooks?**

useState is one of the built-in React hooks. It allows you to track the state in a functional component in React. The term state refers to data or properties that need to be tracked. You need to pass the initial state inside the useState(), which returns variables with the state value and another function to update the current state of the variable.

**14. What is the use of useEffect in React Hooks?**

useEffect is one of the built-in React Hooks. It allows you to manage side effects in your functional components in React. Here the term side effects refer to fetching requests, manipulating DOM, using timer functions and more.

The useEffect accepts two arguments a callback function and dependencies. The callback function contains the side effects, while the dependencies are optional. If the value of the dependencies has changed between the rendering, then only useEffect() to execute the callback function.

**15. What is the use of useRef in React Hooks?**

useRef is one of the built-in React Hooks. It allows you to persist values between render. You can also use it to store a mutable value that does not cause a re-render when updated. You must pass the initial value inside the useRef, which returns a mutable ref object. The object has a property called "current", where the value is persisted.

**16. What is the use of the useCallback hook?**

useCallback is one of the react hooks used to prevent functions declared within the body of function components from being recreated on every render.

**17. What is the use of the useMemo hook?**

The useMemo hook is used to memoise an expensive operation that we give it. The term "memoise" refers to remembering past values that have already been computed.

**18. Why were Hooks introduced in React?**

The main reason for introducing React Hooks was to make the functional components stateful. Before React v16.8, the functional components were called stateless components as we could only do the state management and the life cycle methods using only the class components. So whenever we needed to use the state management or life cycle methods, we had to change the functional components to the class components.

**19. Name the rules that must be followed while using React Hooks.**

The rules that must be followed while using React Hooks are mentioned below:

* You can only call the React Hooks from the react functional components.
* React hooks must only be called at the top level. They should not be called inside nested functions, loops or conditions.

**20.Why do React Hooks make use of refs?**

Following are some of the reasons for using refs in React Hooks:

* + Managing focus, media playback or text selection.
  + Integrating with DOM libraries by third-party.
  + Triggering the imperative animations.

**21. What do you know about the Custom Hooks?**

React custom hooks are reusable functions that a React JS developer can use to add special and unique functionality to the React applications.

Custom Hooks in react is a JavaScript function whose name starts with "use", and they may call other Hooks.

**22.What is Redux and its architecture**

Redux is a popular state management library for JavaScript applications, and it is often used in conjunction with React. Redux architecture is based on three main principles:

* **A single store:**
  + Redux applications have a single store that holds the entire state of the application. This makes it easy to keep track of the state and to make changes to it.
* **State updates are made with pure reducers:**
  + Reducers are pure functions that take the current state and an action as input and return a new state. This makes it easy to reason about how the state will change in response to an action.
* **Changes are made with actions:**
  + Actions are objects that describe what happened in the application. They are dispatched to the store, which then updates the state using the reducers.

**23.What is the context API in React?**

The React Context API is a way for a React app to effectively produce global variables that can be passed around. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. Context is also touted as an easier, lighter approach to state management using Redux.

------------------------------------------------- **JAVASCRIPT** --------------------------------------------------------

**1.What is JavaScript?**

JavaScript is a lightweight, cross-platform, single-threaded, and interpreted compiled programming language. It is also known as the scripting language for webpages. It is well-known for the development of web pages, and many non-browser environments also use it.

**2. ECMAScript 2015 or ES6 Features:**

* [**The let keyword**](https://www.w3schools.com/js/js_es6.asp#mark_let)

The let keyword allows you to declare a variable with block scope.

let x = 10;

x

10

x = 50;

50

x

50

* [**The const keyword**](https://www.w3schools.com/js/js_es6.asp#mark_const)

The const keyword allows you to declare a constant (a JavaScript variable with a constant value). Constants are similar to let variables, except that the value cannot be changed.

const a = 20;

undefined

a

20

a = 40;

VM191:1 Uncaught TypeError: Assignment to constant variable.

a

20

* [**Arrow Functions**](https://www.w3schools.com/js/js_es6.asp#mark_arrow)

ES6 provides a feature known as Arrow Functions. It provides a more concise syntax for writing function expressions by removing the "function" and "return" keywords.

Arrow functions are defined using the fat arrow (=>) notation.

var sum = function(a, b) { return a + b; }

console.log(sum(2, 3)); // 5

// Arrow function

var sum = (a, b) => a + b;

console.log(sum(2, 3)); // 5

As you can see there is no function and return keyword in arrow function declaration.

You can also skip the parentheses i.e. () in case when there is exactly one parameter, but you will always need to use it when you have zero or more than one parameter.

Additionally, if there's more than one expression in the function body, you need to wrap it braces ({}). In this case you also need to use the return statement to return a value.

* [**The ... Operator**](https://www.w3schools.com/js/js_es6.asp#mark_spread)

The spread operator, which is also denoted by (...), performs the exact opposite function of the rest operator. The spread operator spreads out (i.e. splits up) an array and passes the values into the specified function, as shown in the following example:

**function addNumbers(a, b, c) {**

**return a + b + c;**

**}**

**let numbers = [5, 12, 8];**

**console.log(addNumbers(...numbers));**

**25**

* [**For/of**](https://www.w3schools.com/js/js_es6.asp#mark_forof)
* [**Map Objects**](https://www.w3schools.com/js/js_es6.asp#mark_map)

Being able to use an Object as a key is an important Map feature

**const ab = new Map([**

**["apple",500],**

**["orange",400],**

**["idly",300],**

**]);**

**ab.get("apple");**

**500**

* [**Set Objects**](https://www.w3schools.com/js/js_es6.asp#mark_set)

**let letter = new Set();**

**letter**

**Set(0) {size: 0}**

**letter.add(1);**

**Set(1) {1}letter.add(2);**

**Set(2) {1, 2}**

**letter.add(kkasjdkaks);**

**VM1595:1 Uncaught ReferenceError: kkasjdkaks is not defined**

**at <anonymous>:1:12**

**letter.add(q);**

**VM1616:1 Uncaught ReferenceError: q is not defined**

**at <anonymous>:1:12**

**letter.add(4);**

**Set(3) {1, 2, 4}**

**letter**

**Set(3) {1, 2, 4}**

* [**Classes**](https://www.w3schools.com/js/js_es6.asp#mark_class)

In ECMAScript 5 and earlier, classes were never existed in JavaScript. Classes are introduced in ES6 which looks similar to classes in other object oriented languages, such as Java, PHP, etc., however they do not work exactly the same way. ES6 classes make it easier to create objects, implement inheritance by using the extends keyword, and reuse the code.

In ES6 you can declare a class using the new class keyword followed by a class-name. By convention class names are written in TitleCase (i.e. capitalizing the first letter of each word).

**class Rectangle {**

**// Class constructor**

**constructor(length, width) {**

**this.length = length;**

**this.width = width;**

**}**

**// Class method**

**getArea() {**

**return this.length \* this.width;**

**}**

**}**

**// Square class inherits from the Rectangle class**

**class Square extends Rectangle {**

**// Child class constructor**

**constructor(length) {**

**// Call parent's constructor**

**super(length, length);**

**}**

**// Child class method**

**getPerimeter() {**

**return 2 \* (this.length + this.width);**

**}**

**}**

**let rectangle = new Rectangle(5, 10);**

**alert(rectangle.getArea()); // 50**

**let square = new Square(5);**

**alert(square.getArea()); // 25**

**alert(square.getPerimeter()); // 20**

In the above example the Square class inherits from Rectangle using the extends keyword. Classes that inherit from other classes are referred to as derived classes or child classes.

Also, you must call super() in the child class constructor before accessing the context (this). For instance, if you omit the super() and call the getArea() method on square object it will result in an error, since getArea() method require access to this keyword.

* [**Promises**](https://www.w3schools.com/js/js_es6.asp#mark_promise)

A Promise is a JavaScript object that links "Producing Code" and "Consuming Code".

"Producing Code" can take some time and "Consuming Code" must wait for the result.

Promises are used for asynchronous execution.

We can use promise with the arrow function as demonstrated below.

**const myPromise = new Promise(function(myResolve, myReject) {**

**// "Producing Code" (May take some time)**

**myResolve(); // when successful**

**myReject(); // when error**

**});**

**// "Consuming Code" (Must wait for a fulfilled Promise).**

**myPromise.then(**

**function(value) { /\* code if successful \*/ },**

**function(error) { /\* code if some error \*/ }**

**);**

**const myPromise = new Promise(function(myResolve, myReject) {**

**setTimeout(function() { myResolve("I love You !!"); }, 3000);**

**});**

**myPromise.then(function(value) {**

**document.getElementById("demo").innerHTML = value;**

**});**

**var asyncCall = new Promise((resolve, reject) => {**

**// do something**

**resolve();**

**}).then(()=> {**

**console.log('DON!');**

**})**

* [**Symbol**](https://www.w3schools.com/js/js_es6.asp#mark_symbol)

A JavaScript Symbol is a primitive data type just like Number, String, or Boolean.

It represents a unique "hidden" identifier that no other code can accidentally access.

For instance, if different coders want to add a person.id property to a person object belonging to a third-party code, they could mix each others values.

Using Symbol() to create a unique identifiers, solves this problem:

**const person = {**

**firstName: "John",**

**lastName: "Doe",**

**age: 50,**

**eyeColor: "blue"**

**};**

**let id = Symbol('id');**

**person[id] = 140353;**

**// Now person[id] = 140353**

**// but person.id is still undefined**

* [**Default Parameters**](https://www.w3schools.com/js/js_es6.asp#mark_param)

ES6 allows function parameters to have default values.

**function myFunction(x, y = 10) {**

**// y is 10 if not passed or undefined**

**return x + y;**

**}**

**myFunction(5);**

* [**Function Rest Parameter**](https://www.w3schools.com/js/js_es6.asp#mark_rest)

ES6 introduces rest parameters that allow us to pass an arbitrary number of parameters to a function in the form of an array. This is particularly helpful in situations when you want to pass parameters to a function but you have no idea how many you will need.

A rest parameter is specified by prefixing a named parameter with rest operator (...) i.e. three dots. Rest parameter can only be the last one in the list of parameters, and there can only be one rest parameter. Take a look at the following example, to see how it works:

**function sortNames(...names) {**

**return names.sort();**

**}**

**alert(sortNames("Sarah", "Harry", "Peter")); // Harry,Peter,Sarah**

**alert(sortNames("Tony", "Ben", "Rick", "Jos")); // John,Jos,Rick,Tony**

When the rest parameter is the only parameter in a function, it gets all the arguments passed to the function, otherwise it gets the rest of the arguments that exceeds the number of named parameters.

**function myFunction(a, b, ...args) {**

**return args;**

**}**

**alert(myFunction(1, 2, 3, 4, 5)); // 3,4,5**

**alert(myFunction(-7, 5, 0, -2, 4.5, 1, 3)); // 0,-2,4.5,1,3**

* [**String.includes()**](https://www.w3schools.com/js/js_es6.asp#mark_includes)

The includes() method returns true if a string contains a specified value, otherwise false:

**let text = "Hello world, welcome to the universe.";**

**text.includes("world") // Returns true**

* [**String.startsWith()**](https://www.w3schools.com/js/js_es6.asp#mark_startswith)

The startsWith() method returns true if a string begins with a specified value, otherwise false:

**let text = "Hello world, welcome to the universe.";**

**text.startsWith("Hello") // Returns true**

* [**String.endsWith()**](https://www.w3schools.com/js/js_es6.asp#mark_endswith)

The endsWith() method returns true if a string ends with a specified value, otherwise false:

**var text = "John Doe";**

**text.endsWith("Doe") // Returns true**

* [**Array.from()**](https://www.w3schools.com/js/js_es6.asp#mark_array_from)

The Array.from() method returns an Array object from any object with a length property or any iterable object.

**Array.from("ABCDEFG") // Returns [A,B,C,D,E,F,G]**

* [**Array keys()**](https://www.w3schools.com/js/js_es6.asp#mark_array_keys)

The keys() method returns an Array Iterator object with the keys of an array.

**const fruits = ["Banana", "Orange", "Apple", "Mango"];**

**const keys = fruits.keys();**

**let text = "";**

**for (let x of keys) {**

**text += x + "<br>";**

**}**

**o/p**

**0**

**1**

**2**

**3**

* [**Array find()**](https://www.w3schools.com/js/js_es6.asp#mark_array_find)

The find() method returns the value of the first array element that passes a test function.

This example finds (returns the value of ) the first element that is larger than 18

**const numbers = [4, 9, 16, 25, 29];**

**let first = numbers.find(myFunction);**

**function myFunction(value, index, array) {**

**return value > 18;**

**}**

**o/p First number over 18 is 25**

Note that the function takes 3 arguments:

* The item value
* The item index
* The array itself
* [**Array findIndex()**](https://www.w3schools.com/js/js_es6.asp#mark_array_findIndex)

The findIndex() method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

**const numbers = [4, 9, 16, 25, 29];**

**let first = numbers.findIndex(myFunction);**

**function myFunction(value, index, array) {**

**return value > 18;**

**}**

**o/p First number over 18 has index 3**

Note that the function takes 3 arguments:

* The item value
* The item index
* The array itself
* [**New Math Methods**](https://www.w3schools.com/js/js_es6.asp#mark_math_methods)
* **Math.trunc()**

Math.trunc(x) returns the integer part of x:

Math.trunc(4.9); // returns 4

Math.trunc(4.7); // returns 4

Math.trunc(4.4); // returns 4

Math.trunc(4.2); // returns 4

Math.trunc(-4.2); // returns -4

* **Math.sign()**

Math.sign(x) returns if x is negative, null or positive:

Math.sign(-4); // returns -1

Math.sign(0); // returns 0

Math.sign(4); // returns 1

* **Math.cbrt()**

Math.cbrt(x) returns the cube root of x:

Math.cbrt(8); // returns 2

Math.cbrt(64); // returns 4

Math.cbrt(125); // returns 5

* **Math.log2()**

Math.log2(x) returns the base 2 logarithm of x:

Math.log2(2); // returns 1

* **Math.log10()**

Math.log10(x) returns the base 10 logarithm of x:

Math.log10(10); // returns 1

* [**New Number Properties**](https://www.w3schools.com/js/js_es6.asp#mark_number_properties)
* **EPSILON**

let x = Number.EPSILON;

o/p 2.220446049250313e-16

* **MIN\_SAFE\_INTEGER**

let x = Number.MIN\_SAFE\_INTEGER;

o/p -9007199254740991

* **MAX\_SAFE\_INTEGER**

let x = Number.MAX\_SAFE\_INTEGER;

9007199254740991

* [**New Number Methods**](https://www.w3schools.com/js/js_es6.asp#mark_number_methods)

The Number.isInteger() method returns true if the argument is an integer.

Number.isInteger(10); // returns true

Number.isInteger(10.5); // returns false

* [**New Global Methods**](https://www.w3schools.com/js/js_es6.asp#mark_global_methods)
* [**Object entries**](https://www.w3schools.com/js/js_es6.asp#mark_entries)
* [**JavaScript Modules**](https://www.w3schools.com/js/js_es6.asp#mark_modules)

**3.** **What is memoization?**

**Quick answer:**

It is a way to optimize application performance by caching results of time-consuming pure functions calculations.

**Longer answer:**

Pure functions are basically functions that return the same result if you pass the same data and they don't change anything outside of its scope.

**let pureAdd = (a, b) => a + b**

This pureAdd function doesn't change anything outside it just returns the answer and it always returns the same answer for the same arguments.

With these restrictions come the benefits. If the result is the same every time we run the functions, then why don't we just calculate it once and save the result?

Real-life applications:

It's not only a theoretical benefit but actually a practical one.

For example, there is React.memo which does memoization.

If your component renders the same result given the same props ... React will skip rendering the component, and reuse the last rendered result.

Also, there is a useMemo hook, which also does memoization. useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

**function fibonnaci(n) {**

**if(n <2)**

**return 1;**

**return fibonnaci(n-1) + fibonnaci(n-2);**

**}**

**function fibocache(n, cache) {**

**cache = cache || [1,1];**

**if(cache[n]) {**

**return cache[n];**

**}**

**return cache[n] = fibocache(n-1,cache) + fibocache(n-2, cache);**

**}**

**4. What is Hoisting?**

Hoisting is a concept that enables us to extract values of variables and functions even before initializing/assigning value without getting errors and this happens during the 1st phase (memory creation phase) of the Execution Context.

**Features of Hoisting:**

* In JavaScript, Hoisting is the default behaviour of moving all the declarations at the top of the scope before code execution. Basically, it gives us an advantage that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.
* It allows us to call functions before even writing them in our code.

Note: JavaScript only hoists declarations, not initializations.

JavaScript allocates memory for all variables and functions defined in the program before execution.

Sequence of variable declaration: The following is the sequence in which variable declaration and initialization occur.

**Declaration –> Initialisation/Assignment –> Usage**

**Variable lifecycle:**

**let a; // Declaration**

**a = 100; // Assignment**

**console.log(a); // Usage**

However, since JavaScript allows us to both declare and initialize our variables simultaneously, so we can declare and initialize at the same time.

**let a = 100;**

Note: Always remember that in the background the Javascript is first declaring the variable and then initializing them. It is also good to know that variable declarations are processed before any code is executed.

However, in javascript, undeclared variables do not exist until the code assigning them is executed. Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed. This means that all undeclared variables are global variables.

**5. What is Closure?**

JavaScript closure is a feature that allows inner functions to access the outer scope of a function. Closure helps in binding a function to its outer boundary and is created automatically whenever a function is created. A block is also treated as a scope since ES6. Since JavaScript is event-driven so closures are useful as it helps to maintain the state between events.

**Lexical Scoping:**

A function scope’s ability to access variables from the parent scope is known as lexical scope. We refer to the parent function’s lexical binding of the child function as “lexically binding.”

Let’s see and understand closure through an example.

**function foo() {**

**let b = 1;**

**function inner() {**

**return b;**

**}**

**return inner;**

**}**

**let get\_func\_inner = foo();**

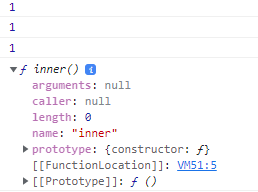
**console.log(get\_func\_inner());**

**console.log(get\_func\_inner());**

**console.log(get\_func\_inner());**

We can access the variable b which is defined in the function foo() through function inner() as the later preserves the scope chain of the enclosing function at the time of execution of the enclosing function i.e. the inner function knows the value of b through its scope chain.

This is closure in action that is inner function can have access to the outer function variables as well as all the global variables.



**Closure in JavaScript**

Closure is **the concept of function + lexical environment** in which function it was created. so every function declared within another function then it has access to the scope chain of the outer function and the variables created within the scope of the outer function will not get destroyed.

Now let’s look at another example.

**Example 2:** This example shows the basic use of closure.

**function foo(outer\_arg) {**

**function inner(inner\_arg) {**

**return outer\_arg + inner\_arg;**

**}**

**return inner;**

**}**

**let get\_func\_inner = foo(5);**

**console.log(get\_func\_inner(4));**

**console.log(get\_func\_inner(3));**

**Output:** In the above example we used a parameter function rather than a default one. Not even when we are done with the execution of foo(5) we can access the outer\_arg variable from the inner function. And on the execution of the inner function produce the summation of outer\_arg and inner\_arg as desired.

**9**

**8**

Now let’s see an example of closure within a loop.

In this example, we would store an anonymous function at every index of an array.

**Example 3:** This example shows the basic use of closure.

**// Outer function**

**function outer() {**

**let arr = [];**

**let i;**

**for (i = 0; i < 4; i++) {**

**// storing anonymous function**

**arr[i] = function () { return i; }**

**}**

**// returning the array.**

**return arr;**

**}**

**let get\_arr = outer();**

**console.log(get\_arr[0]());**

**console.log(get\_arr[1]());**

**console.log(get\_arr[2]());**

**console.log(get\_arr[3]());**

**Output:**

Did you guess the right answer? In the above code, we have created four closures that point to the variable i which is the local variable to the function outer. Closure doesn’t remember the value of the variable it only points to the variable or stores the reference of the variable and hence, returns the current value. In the above code when we try to update the value it gets reflected all because the closure stores the reference.

**4**

**4**

**4**

**4**

Let’s see the correct way to write the above code so as to get different values of i at different indexes.

**5. What is currying ?**

Currying in JavaScript transforms a function with multiple arguments into a nested series of functions, each taking a single argument. Currying helps you avoid passing the same variable multiple times, and it helps you create a higher order function.

That is, when we turn a function call sum(1,2,3) into sum(1)(2)(3).

The number of arguments a function takes is also called arity.

**function sum(a, b) {**

**// do something**

**}**

**function \_sum(a, b, c) {**

**// do something**

**}**

The function sum takes two arguments (two-arity function) and \_sum takes three arguments (three-arity function).

Curried functions are constructed by chaining closures and by defining and immediately returning their inner functions simultaneously.

**6.Why Is Currying in JavaScript Useful?**

Currying helps you avoid passing the same variable again and again.

* It helps to create a higher order function.
* Currying transforms a function with multiple arguments into a sequence/series of functions, each taking a single argument.

**For example:**

**function sum(a, b, c) {**

**return a + b + c;**

**}**

**sum(1,2,3); // 6**

As you can see, this is a function with full arguments. Let’s create a curried version of the function and see how we would call the same function (and get the same result) in a series of calls:

**function sum(a) {**

**return (b) => {**

**return (c) => {**

**return a + b + c**

**}**

**}**

**}**

**console.log(sum(1)(2)(3)) // 6**

We could even separate this sum(1)(2)(3) to understand it better:

**const sum1 = sum(1);**

**const sum2 = sum1(2);**

**const result = sum2(3);**

**console.log(result); // 6**

**7.What are Callbacks?**

A callback is a function that is passed as an argument to another function, and is called after the main function has finished its execution. The main function is called with a callback function as its argument, and when the main function is finished, it calls the callback function to provide a result. Callbacks allow you to handle the results of an asynchronous operation in a non-blocking manner, which means that the program can continue to run while the operation is being executed.

**8.Why use Callbacks?**

Callbacks are used to handle the results of asynchronous operations in a non-blocking manner. Asynchronous operations are operations that take a significant amount of time to complete, such as network requests, file I/O, and database queries. If these operations were executed synchronously, the program would freeze and wait for the operation to complete before continuing. This can lead to a poor user experience, as the program would appear unresponsive.

Callbacks allow you to continue executing code while the operation is being executed in the background. Once the operation has completed, the callback function is called with the result of the operation. This way, you can ensure that the program remains responsive and the user experience is not impacted

**Real-Life Examples:**

**Loading images on a website:** When you load a website, images can take a while to load, especially if they’re large. If images were loaded synchronously, the website would freeze and wait for each image to load before continuing. With callbacks, you can load the images asynchronously, which means that the website continues to load while the images are being loaded in the background.

**Handling form submissions:** When a user submits a form, it takes time to process the data and send it to the server. If the form submission was executed synchronously, the user would have to wait for the data to be processed and sent before the form can be submitted. With callbacks, you can handle the form submission asynchronously, which means that the user can continue to interact with the form while the data is being processed and sent in the background.

**function mainFunction(callback) {**

**console.log("Performing operation...");**

// Use setTimeout to simulate an asynchronous operation

**setTimeout(function() {**

**callback("Operation complete");**

**}, 1000);**

**}**

// Define the callback function

**function callbackFunction(result) {**

**console.log("Result: " + result);**

**}**

// Call the main function with the callback function

**mainFunction(callbackFunction);**

**Output**

**Performing operation...**

**Result: Operation complete**

**Explanation:**

* We first define a mainFunction that takes a callback as an argument.
* The mainFunction uses setTimeout to simulate an asynchronous operation. The setTimeout function takes two arguments: a callback function and a delay time in milliseconds.
* The setTimeout function calls the callback function with the result of the operation after the specified delay time.
* We then define a callbackFunction that logs the result of the operation.

Finally, we call the mainFunction with the callbackFunction as its argument.

**Callback with array.forEach:**

// Define an array of numbers

**var numbers = [1, 2, 3, 4, 5];**

// Define the main function

**function mainFunction(callback) {**

**console.log("Performing operation...");**

// Use Array.forEach to loop through the array of numbers

**numbers.forEach(callback);**

**}**

// Define the callback function

**function callbackFunction(number) {**

**console.log("Result: " + number);**

**}**

// Call the main function with the callback function

**mainFunction(callbackFunction);**

**Output**

**Performing operation...**

**Result: 1**

**Result: 2**

**Result: 3**

**Result: 4**

**Result: 5**

**Explanation:**

* We first define an array of numbers numbers.
* We then define a mainFunction that takes a callback as an argument.
* The mainFunction uses Array.forEach to loop through the numbers array and call the callback function for each element in the array.
* We then define a callbackFunction that logs each number in the numbers array.
* Finally, we call the mainFunction with the callbackFunction as its argument.

In conclusion, callbacks are an important aspect of JavaScript programming and are used to handle the results of asynchronous operations in a non-blocking manner. With the help of these examples, you should have a better understanding of how to use callbacks in your own projects.

<https://interviewprep.org/document-object-model-dom-interview-questions/>

<https://interviewprep.org/javascript-objects-interview-questions/>