

Interview questions & Answers

Saikrishna Nangunuri | SDE2 @HCCI |

<https://www.linkedin.com/in/saikrishnanangunuri/>

1. What is the difference between 'Pass by Value' and 'Pass by Reference'?

In JavaScript, whenever a function is called, the arguments can be passed in two ways, either pass by value or pass by reference.

- Primitive datatypes such as string, number, boolean, null and undefined are passed by value.
- Non -primitive datatypes such as object, arrays or functions are passed by reference.

In Pass by value, parameters passed as an arguments creates their own copy. So any changes made inside the function are made to the copied value so it will not affect the original value.

In Pass by reference, parameters passed as an arguments does not creates their own copy. so any changes made inside the function will affect the original value.

2. What is the difference between map and filter ?

- Both map and filter are useful in JavaScript when working with an arrays.
- map transforms each element of an array and creates a new array which contains the transformed elements. whereas filter will creates a new array with only those elements which satisfies the specified condition.

3. What is the difference between map() and forEach()

- map method is used to transform the elements of an array. Whereas forEach method is used to loop through the elements of an array.
 - map method will return a new array with the transformed values. forEach method does not return a new array.
 - map method can be used with other array methods like filter method. whereas forEach method cannot be used with other array methods as it does not return any array.
-

4. What is the difference between Pure and Impure functions?

Pure Functions:

- Pure functions are the functions which will return same output for same arguments passed to the function.
- This will not have any side effects.
- It does not modify any non local state.

```
function greeting(name) {  
  return `Hello ${name}`;  
}  
console.log(greeting("Saikrishna Nangunuri"));
```

Impure Functions:

- Impure functions are the functions which will return inconsistent output for same arguments passed to the function.
- This will have side effects.
- This will modify non local state.

```
let message = "good morning";
function greeting1(name) {
    return `Hello ${name} , ${message}`;
}
console.log(greeting1("Saikrishna Nangunuri"));
```

Ref: <https://www.scaler.com/topics/pure-function-in-javascript/>

5. What is the difference between for-in and for-of ?

Both for-in and for-of are used to iterate over the datastructure.

for-in:

- for-in iterates over the enumerable property keys of an object.

for-of:

- for-of is used to iterate over the values of an iterable object.
- Examples of iterable objects are array, string, nodeLists etc. (for of on object returns error)

<https://stackoverflow.com/questions/29285897/difference-between-for-in-and-for-of-statements?answertab=scoredesc#tab-top>

6. What are the differences between call(), apply() and bind() ?

- Call method will invoke the function immediately with the given this value and allow us to pass the arguments one by one with comma separator.
- Apply method will invoke the function immediately with given this value and allow us to pass the arguments as an array.
- Bind method will return a new function with the given this value and arguments which can be invoked later.

7. List out some key features of ES6 ?

1. Let and Const declarations.
2. Arrow functions
3. Template literals
4. Destructuring assignment
5. Spread and Rest operators
6. Default parameters
7. Promises
8. Modules
9. Map, Set, Weakmap, Weakset
10. Classes

8. What's the spread operator in javascript ?

Spread operator is used to spread or expand the elements of an iterable like array or string into individual elements.

Uses:

1. Concatenating arrays.

```
let x = [1, 2];  
let y = [3, 4];  
  
let z = [...x, ...y]    ⇒⇒ 1, 2, 3, 4
```

2. Copying arrays or objects.

```
let a = [...x] // 1,2
```

3. Passing array of values as individual arguments to a function.

```
function createExample(arg1, arg2){  
  console.log(arg1, arg2);  
}  
  
createExample(...a)
```

9. What is rest operator in javascript ?

Rest operator is used to condense multiple elements into single array or object.

This is useful when we don't know how many parameters a function may receive and you want to capture all of them as an array.

```
function Example(...args){  
  console.log(args)  
}  
  
Example(1, 2, 3, 4);
```

10. What are DRY, KISS, YAGNI, SOLID Principles ?

- **DRY**: Do not repeat yourself.
 - Avoid duplicates. This makes software more maintainable and less error-prone.
- **KISS**: Keep it simple stupid.
 - Keep the software design and implementation as simple as possible. This makes software more testable, understandable and maintainable.

- **YAGNI**: You are not going to need it.
 - Avoid adding unnecessary features/functionalities to the software. This makes software focussed on essential requirements and makes it more maintainable.
 - **SOLID**:
 - **O - Open/Closed principle**: Classes must be open for extension and closed to modification. This way we can stop ourselves from modifying the existing code and causing potential bugs.
 - **S - Single responsibility**: means each class should have one job or responsibility.
 - **L - Liskov Substitution**: If class A is subtype of class B then classB should be able to replace classA with out disrupting the behaviour of our program.
 - **I - Interface segregation**: Larger interfaces must be split into smaller ones.
 - **D - Dependency inversion**: High level modules should not depend on low level modules. Both should depend on abstraction.
-

11. What is temporal dead zone ?

- It is a specific time period in the execution of javascript code where the variables declared with let and const exists but cannot be accessed until the value is assigned.
 - Any attempt to access them result in reference errors.
-

12. Different ways to create object in javascript ?

<https://www.scaler.com/topics/objects-in-javascript/>

- **Object literal** :

```
let userDetails = {  
  name: "Saikrishna",  
  city: "Hyderabad"  
}
```

- **Object constructor :**

```
let userDetails = new Object();  
userDetails.name = "Saikrishna";  
userDetails.city = "Hyderabad";
```

- **Object.Create() :**

This is used when we want to inherit properties from an existing object while creating a new object.

```
let animal = {  
  name: "Animal name"  
}  
  
let cat = Object.create(animal);
```

- **Object.assign() :**

This is used when we want to include properties from multiple other objects into new object we are creating.

```
let lesson = {  
  lessonName: "Data structures"  
};  
  
let teacher = {  
  teacher: "Saikrishna"  
};  
  
let course = Object.assign({}, lesson, teacher);
```

13. Whats the difference between Object.keys, values and entries

- **Object.keys()**: This will return the array of keys
- **Object.values()**: This will return the array of values
- **Object.entries()**: This will return array of [key,value] pairs.

```
let data = {  
  name: "Sai",  
  lang: "English"  
};  
  
Object.keys(data) // ["name","lang"]  
Object.values(data) // ["Sai","english"]  
Object.entries(data) // [["name","Sai"],["lang","English"]]
```

14. Whats the difference between Object.freeze() vs Object.seal()

- **Object.freeze:**
 - Will make the object immutable (prevents the addition of new propeties and prevents modification of existing properties)

```
let data = {  
  a : 10  
};  
  
Object.freeze(data);  
data.a= 20;  
data.c = 30;  
  
console.log(data)
```



```
output: {  
  a: 10  
}
```

- **Object.Seal():**

- Will prevent the addition of new properties but we can modify existing properties.

```
let data = {  
  a : 10  
};  
  
Object.seal(data);  
data.a = 20;  
data.c = 30;  
  
console.log(data)
```

Output:
data: {
 a: 20
}

15. What is a polyfill in javascript ?

- A polyfill is a piece of code which provides the modern functionality to the older browsers that does not natively support it.
- **Polyfill for foreach:**

```
Array.prototype.forEach = function(callback) {  
  for (var i = 0; i < this.length; i++) {  
    if (i in array) {  
      callback.call(array[i], i, array);  
    }  
  }  
}
```

```

    }
  }
};

let array = [1, 2, 3, 4, 5];

array.forEach((element, id, arrd) => {
  console.log(`${element}, ${id}`, arrd)
})

```

16. What is generator function in javascript ?

- A generator function is a function which can be paused and resumed at any point during execution.
- They are defined by using function* and it contains one or more yield expressions.
- The main method of generator is next(). when called, it runs the execution until the nearest yield.
- It returns an object which contains 2 properties. i.e., done and value.
 - **done:** the yielded value
 - **value:** true if function code has finished. else false.
- <https://javascript.info/generators>

```

function* generatorFunction() {
  yield 1;
  yield 2;
  yield 3;
  return 4
}

const generator = generatorFunction();

```

```
console.log(generator.next()); // Output: { value: 1, done: f
console.log(generator.next()); // Output: { value: 2, done: f
console.log(generator.next()); // Output: { value: 3, done: f
console.log(generator.next()); // Output: { value: 4, done: t
```

17. What is prototype in javascript ?

- If we want to add properties at later stage to a function which can be accessible across all the instances. Then we will be using prototype.
- <https://www.tutorialsteacher.com/javascript/prototype-in-javascript>

```
function Student(){
    this.name = "Saikrishna",
    this.exp= "8"
}

Student.prototype.company = "Hexagon"

let std1 = new Student();
std1.exp = "9"

let std2 = new Student();
std2.exp = "10"

console.log(std1);
console.log(std2)
```

18. What is IIFE ?

- IIFE means immediately invoked function expression.
- functions which are executed immediately once they are mounted to the stack is called iife.

- They does not require any explicit call to invoke the function.
- <https://www.geeksforgeeks.org/immediately-invoked-function-expressions-iife-in-javascript/>
- <https://www.tutorialsteacher.com/javascript/immediately-invoked-function-expression-iife>

```
(function(){  
    console.log("2222")  
})();
```

Arrow functions: <https://www.codingninjas.com/studio/library/difference-between-arrow-function-and-normal-function>

19. What is CORS ?

- CORS means cross origin resource sharing.
- It is a security feature that allows the webapplications from one domain to request the resources like Api's/scripts from another domain.
- cors works by adding specific http headers to control which origins have access to the resources and under what conditions.

20. What are the different datatypes in javascript ?

- Primitive datatypes:
 - String
 - number
 - boolean

- null
 - undefined
 - BigInt
 - symbol
 - **Non-Primitive datatypes:**
 - Object
 - Array
 - Date
-

21. What are the difference between typescript and javascript ?

- Typescript points out the compilation errors at the time of development. Because of this, getting runtime errors is less likely.
 - Typescript supports interfaces whereas javascript does not.
 - Typescript is better suited for large scale applications where as javascript is suited for small scale applications.
 - Typescript is the superset of javascript and has all the object oriented features.
 - Functions have optional parameters in typescript whereas in javascript does not have it.
 - Typescript takes longer time to compile code.
-

22. What is authentication vs authorization ?

- **Authentication:**
 - Its the process of verifying who the user is.
- **Authorization:**

- Its the process of verifying what they have access to. What files and data user has access to.
-

23. Difference between null and undefined ?

- **Null:**
 - If we assign null to a variable, it means it will not have any value
 - **Undefined:**
 - means the variable has been declared but not assigned any value yet.
-

24. What is the output of 3+2+"7" ?

- 57
-

25. Slice vs Splice in javascript ?

- **Slice:**
 - If we want to create an array that is subset of existing array with out changing the original array, then we will use slice.

```
let arr = [1, 2, 3, 4];  
let newArr = arr.slice(1, 3);  
  
console.log(newArr) // [2, 3]
```

- **Splice:**
 - If we want to add/delete/replace the existing elements in the array, then we will use splice.

```
let arr = [1,2,3,4,5,0,10];
let newArr = arr.splice(2,4,8,9,6);
// splice(startIndex,numberOfItemsToRemove,replaceElements

console.log(arr); // [1,28,9,6,10]
console.log(newArr); // [3,4,5,0]
```

26. What is destructuring ?

- It is introduced in Es6.
- It allows us to assign the object properties and array values to distinct variables.

```
const user = {
  "age": 10,
  "name": "Saikrishna"
}

const {age,name} = user;
console.log(age,name) // 10,"Saikrishna"
```

```
const [a,b] = [1,2];
console.log(a,b) // 1,2
```

27. What is setTimeout in javascript ?

- setTimeout is used to call a function or evaluate an expression after a specific number of milliseconds.

```
setTimeout(function(){
  console.log("Prints Hello after 2 seconds")
},2000);

// Logs message after 2 seconds
```

28. What is setInterval in javascript ?

- setInterval method is used to call a function or evaluate an expression at specific intervals.

```
setInterval(function(){
  console.log("Prints Hello after every 2 seconds");
},2000);
```

29. What are Promises in javascript ?

- Promise is an object which represents the eventual completion or failure of an asynchronous operation in javascript.
- At any point of time, promise will be in any of these below states.,
 - **Fulfilled**: Action related to promise is succeeded.
 - **Rejected**: Action related to the promise is failed.
 - **Pending**: Promise is neither fulfilled nor rejected
 - **Settled**: Promise has been fulfilled or rejected.
- Promise can be consumed by registering the functions using .then() and .catch() methods.
- **Promise constructor**: will take one argument which is a callback function. This callback function takes 2 arguments resolve and reject.

- If performed operations inside callback function goes well then we will call `resolve()` and if does not go well then we will call `reject()`

```
let promise = new Promise(function(resolve, reject){
    const x = "Saikrishna";
    const y = "Saikrishna";

    if(x === y){
        resolve("Valid")
    } else{
        let err = new Error("Invalid")
        reject(err)
    }
})

promise.then((response)=>{
    console.log("success", response)
}).catch((err)=>{
    console.log("failed", err)
})
```

30. What is a callstack in javascript ?

- Callstack will maintain the order of execution of execution contexts.

31. What is a closure ?

- **Definition:** A function along with its outer environment together forms a closure (or) Closure is a combination of a function along with its lexical scope bundled together.
- Each and every function in javascript has access to its outer lexical environment means access to the variables and functions present in the environments of its parents

- Even when this function is executed in some outer scope(not in original scope) it still remembers the outer lexical environment where it was originally present in the code.

```
function Outer(){
    var a = 10;
    function Inner(){
        console.log(a);
    }
    return Inner;
}

var Close = Outer();
Close();
```

32. What are callbacks in javascript ?

- A callback is a function which is passed as an argument to another function which can be executed later in the code.
- **Usescases:**
 - setTimeout
 - Higher order functions (Like map,filter,forEach).
 - Handling events (Like click/key press events).
 - Handling asynchronous operations (Like reading files, making Http requests).

```
function Print(){
    console.log("Print method");
}

function Hello(Print){
```

```
        console.log("Hello method");  
        Print();  
    }  
  
    Hello(Print);  
  
    Output:  
    Hello method  
    Print method
```

33. What are Higher Order Functions in javascript ?

- A function which takes another function as an argument or returns a function as an output.
- **Advantages:**
 - callback functions
 - Asynchronous programming (functions like setTimeout,setInterval often involves HOF. they allow to work with asynchronous code more effectively.)
 - Abstraction
 - Code reusability
 - Encapsulation
 - Concise and readable code

34. What is the difference between == and === in javascript ?

- == will check for equality of values where as === will check for equality as well as datatypes.

35. Is javascript a dynamically typed language or a statically typed language ?

- Javascript is a dynamically typed language.
- It means all type checks are done at run time (When program is executing).
- So, we can just assign anything to the variable and it works fine.

```
let a;  
a = 0;  
console.log(a) // 0  
a = "Hello"  
console.log(a) // "Hello"
```

- Typescript is a statically typed language. All checks are performed at compile time.
-

36. What is the difference between Indexeddb and sessionStorage ?

- **IndexedDb:**
 - It is used for storing large amount of structured data.
 - It uses object oriented storage model.
 - Persist data beyond the duration of page session.
 - **SessionStorage:**
 - Limited storage, around 5mb of data.
 - Simple key-value storage.
 - Available only for the duration of page session.
-

37. What are Interceptors ?

- Interceptors allows us to modify the request or response before its sent to the server or received from the server.

```
axios.interceptors.request.use((config)=>{
    if(longUrls.include(url)){
        config.timeout = 1000;
    }
    return config;
})

axios.interceptors.response.use((response)=>{
    return response;
})
```

38. What is Hoisting in javascript ?

- In other scripting/server side languages, variables or functions must be declared before using it.
- In javascript, variables and functions can be used before declaring it. The javascript compiler moves all the declarations of variables and functions on top. so there will not be any error. This is called hoisting.

39. What are the differences let, var and const ?

- **Scope:**
 - Variables declared with var are function scoped.(available through out the function where its declared) or global scoped(if defined outside the function).

- Variables declared with let and const are block scoped.
 - **Reassignment:**
 - var and let can be reassigned.
 - const cannot be reassigned.
 - **Hoisting:**
 - var gets hoisted and initialized with undefined.
 - let and const - gets hoisted to the top of the scope but does not get assigned any value.(temporary dead zone)
-

40. What is the output of below logic ?

```
const a = 1<2<3;  
const b = 1>2>3;  
  
console.log(a,b) //true,false
```

41. Differences between Promise.all, allSettled, any, race ?

- **Promise.all:**
 - Will wait for all of the promises to resolve or any one of the promise reject.
- **Promise.allSettled:**
 - Will wait for all the promises to settle (either fulfilled or rejected).
- **Promise.any:**
 - Will return if any one of the promise fulfills or rejects when all the promises are rejected.
- **Promise.race:**

- Will return as soon as when any one of the promise is settled.

<https://medium.com/@log2jeet24/javascript-different-types-of-promise-object-methods-to-handle-the-asynchronous-call-fc93d1506574>

42. What are limitations of arrow functions in javascript ?

Arrow functions are introduced in ES6. They are simple and shorter way to write functions in javascript.

1. Arrow functions cannot be accessed before initialization
 2. Arrow function does not have access to arguments object
 3. Arrow function does not have their own this. Instead, they inherit this from the surrounding code at the time the function is defined.
 4. Arrow functions cannot be used as constructors. Using them with the **new** keyword to create instances throws a **TypeError**.
 5. Arrow functions cannot be used as generator functions.
-

43. What is difference between find vs findIndex ?

- **find:**
 - It will return the first element of array that passes specified condition.

```
function findMethod(){
  let arr = [{id:1,name:"sai"}, {id:2,name:"krishna"}];
  let data = arr.find(x=> x.id==2)
  console.log(data)
}

findMethod()
```

```
Output:
{id:2, name:"krishna"}
```

- **findIndex:**

- It will return the index of first element of an array that passes the specified condition.

```
function findMethod(){
  let arr = [{id:1, name:"sai"}, {id:2, name:"krishna"}];
  let data = arr.findIndex(x=> x.id==2)
  console.log(data)
}
```

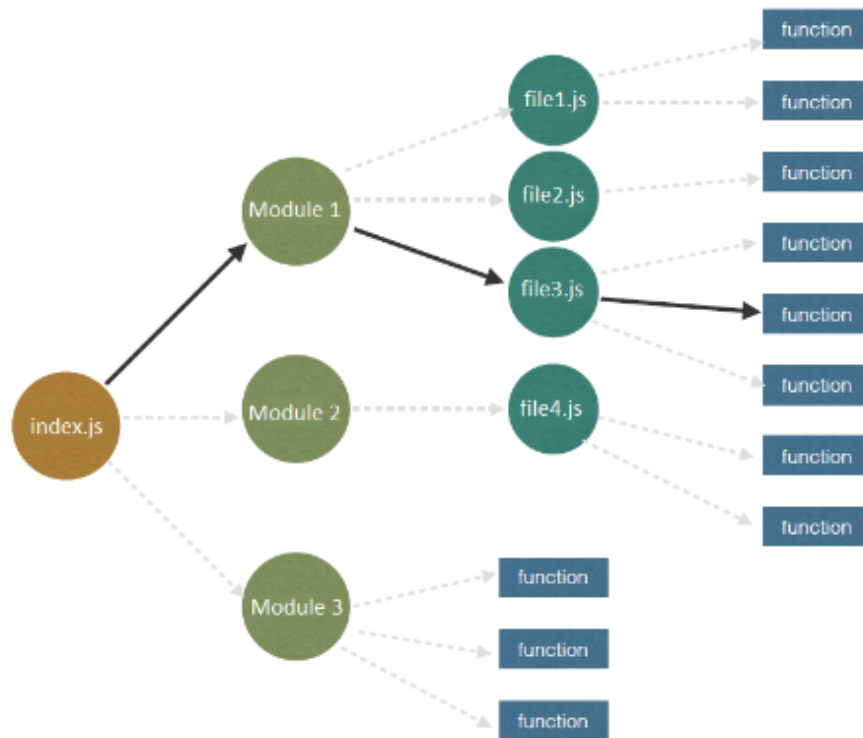
```
findMethod()
```

```
Output:
2
```

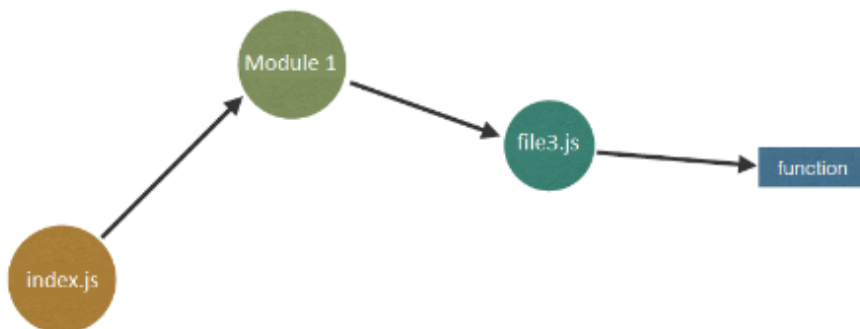
44. What is tree shaking in javascript ?

- It is one of the optimization technique in javascript which removes the unused code from the bundle during the build process.
- It is commonly used in bundling tools like Webpack and Rollup.
- **Advantages:**
 - It reduces the bundle size by eliminating unused modules and functions.
 - Faster load time.
 - Performance will be improved.
 - Cleaner and maintainable codebases.

Before Tree Shaking



After Tree Shaking



45. Guess the output ?

```
console.log("Start");
setTimeout(() => {
  console.log("Timeout");
});
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

Output:

- Start, End, Promise, Timeout.
 - "Start" is logged first because it's a synchronous operation.
 - Then, "End" is logged because it's another synchronous operation.
 - "Promise" is logged because `Promise.resolve().then()` is a microtask and will be executed before the next tick of the event loop.
 - Finally, "Timeout" is logged. Even though it's a `setTimeout` with a delay of 0 milliseconds, it's still a macrotask and will be executed in the next tick of the event loop after all microtasks have been executed.

46. What is the output of `3+2+"7"` ?

- 57

47. What is the output of below logic ?

```
const a = 1<2<3;
const b = 1>2>3;

console.log(a, b) //true, false
```

Output:

- true, false
 - In JavaScript, the comparison operators `<` and `>` have left-to-right associativity. So, `1 < 2 < 3` is evaluated as `(1 < 2) < 3`, which becomes `true < 3`. When comparing a boolean value (`true`) with a number (`3`), JavaScript coerces the boolean to a number, which is `1`. So, `true < 3` evaluates to `1 < 3`, which is `true`.
 - Similarly, `1 > 2 > 3` is evaluated as `(1 > 2) > 3`, which becomes `false > 3`. When comparing a boolean value (`false`) with a number (`3`), JavaScript coerces the boolean to a number, which is `0`. So, `false > 3` evaluates to `0 > 3`, which is `false`.
 - That's why `console.log(a, b)` prints `true false`.

48. Guess the output ?

```
const p = { k: 1, l: 2 };
const q = { k: 1, l: 2 };
let isEqual = p==q;
let isStartEqual = p=== q;

console.log(isEqual, isStartEqual)
```

• OutPut:

- False,False

In JavaScript, when you compare objects using `==` or `===`, you're comparing their references in memory, not their actual contents. Even if two objects have the same properties and values, they are considered unequal unless they reference the exact same object in memory.

In your code:

- `isEqual` will be `false` because `p` and `q` are two different objects in memory, even though they have the same properties and values.

- `isStartEqual` will also be `false` for the same reason. The `===` operator checks for strict equality, meaning it not only compares values but also ensures that the objects being compared reference the exact same memory location.

So, `console.log(isEqual, isStartEqual)` will output `false false`.

49. Guess the output ?

- a) `2+2 = ?`
- b) `"2"+"2" = ?`
- c) `2+2-2 = ?`
- d) `"2"+"2"- "2" = ?`

- **Output:**

```
// a) 2+2 = ?
console.log(2 + 2); // Output: 4

// b) "2"+"2" = ?
console.log("2" + "2"); // Output: "22" (string concatenation)

// c) 2+2-2 = ?
console.log(2 + 2 - 2); // Output: 2

// d) "2"+"2"- "2" = ?
console.log("2" + "2" - "2"); // Output: 20 (string "22" is converted to number 22 and then 22 - 2 = 20)
```

50. What is the output of below logic ?

```
let a = 'jscafe'
a[0] = 'c'
```

```
console.log(a)
```

- **Output:**

- "jsafe"
- Strings are immutable in javascript so we cannot change individual characters by index where as we can create a new string with desired modification as below.
- a = "csafe" // outputs "csafe"

51. Output of below logic ?

```
var x=10;
function foo(){
var x = 5;
console.log(x)
}

foo();
console.log(x)
```

Output: 5 and 10

In JavaScript, this code demonstrates variable scoping. When you declare a variable inside a function using the `var` keyword, it creates a new variable scoped to that function, which may shadow a variable with the same name in an outer scope. Here's what happens step by step:

1. `var x = 10;` : Declares a global variable `x` and initializes it with the value `10`.
2. `function foo() { ... }` : Defines a function named `foo`.
3. `var x = 5;` : Inside the function `foo`, declares a local variable `x` and initializes it with the value `5`. This `x` is scoped to the function `foo` and is different from the global `x`.

4. `console.log(x);` : Logs the value of the local variable `x` (which is `5`) to the console from within the `foo` function.
 5. `foo();` : Calls the `foo` function.
 6. `console.log(x);` : Logs the value of the global variable `x` (which is still `10`) to the console outside the `foo` function.
-

52. This code prints 6 everytime. How to print 1,2,3,4,5,6 ?

```
function x(){  
  
    for(var i=1;i<=5;i++){  
        setTimeout(()=>{  
            console.log(i)  
        }, i*1000)  
    }  
  
}
```

`x();`

Solution: Either use let or closure

```
function x(){  
    function closur(x){  
        setTimeout(()=>{  
            console.log(x)  
        }, x*1000)  
    };  
    for(var i=1;i<=5;i++){  
        closur(i)  
    }  
}
```

```
}  
  
x();
```

53. What will be the output or below code ?

```
function x(){  
  let a = 10;  
  function d(){  
    console.log(a);  
  }  
  a = 500;  
  return d;  
}  
  
var z = x();  
z();
```

Solution: 500 - Closures concept

In JavaScript, this code demonstrates lexical scoping and closure. Let's break it down:

1. `function x() { ... }`: Defines a function named `x`.
2. `let a = 10;`: Declares a variable `a` inside the function `x` and initializes it with the value `10`.
3. `function d() { ... }`: Defines a nested function named `d` inside the function `x`.
4. `console.log(a);`: Logs the value of the variable `a` to the console. Since `d` is defined within the scope of `x`, it has access to the variable `a` defined in `x`.
5. `a = 500;`: Changes the value of the variable `a` to `500`.
6. `return d;`: Returns the function `d` from the function `x`.

7. `var z = x();` : Calls the function `x` and assigns the returned function `d` to the variable `z`.
8. `z();` : Calls the function `d` through the variable `z`.

When you run this code, it will log the value of `a` at the time of executing `d`, which is `500`, because `d` retains access to the variable `a` even after `x` has finished executing. This behavior is possible due to closure, which allows inner functions to access variables from their outer scope even after the outer function has completed execution.

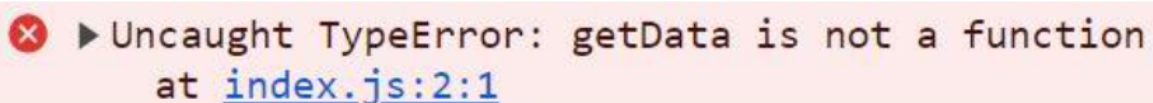
54. What's the output of below logic ?

```
getData1()
getData();

function getData1(){
  console.log("getData11")
}

var getData = () => {
  console.log("Hello")
}
```

Output:



✖ ▶ Uncaught TypeError: getData is not a function
at [index.js:2:1](#)

Explanation:

In JavaScript, function declarations are hoisted to the top of their scope, while variable declarations using `var` are also hoisted but initialized with `undefined`. Here's what happens in your code:

1. `getData1()` is a function declaration and `getData()` is a variable declaration with an arrow function expression assigned to it.

2. When the code runs:

- `getData1()` is a function declaration, so it's hoisted to the top and can be called anywhere in the code. However, it's not called immediately.
- `getData` is declared using `var`, so it's also hoisted to the top but initialized with `undefined`.
- The arrow function assigned to `getData` is not hoisted because it's assigned to a variable.

3. When `getData()` is invoked:

- It will throw an error because `getData` is `undefined`, and you cannot call `undefined` as a function.

Therefore, if you try to run the code as is, you'll encounter an error when attempting to call `getData()`.

If you want to avoid this error, you should either define `getData` before calling it or use a function declaration instead of a variable declaration for `getData`.

Here's how you can do it:

Modification needed for code:

```
var getData = () => {  
  console.log("Hello")  
}  
  
getData1(); // This will log "getData1"  
getData();  // This will log "Hello"
```

55. Whats the output of below code ?

```
function func() {  
  try {  
    console.log(1)  
    return  
  } catch (e) {  
    console.log(2)  
  }  
}
```

```

    } finally {
      console.log(3)
    }
    console.log(4)
  }

  func()

```

Output: 1 & 3

1. The function `func()` is defined.
2. Inside the `try` block:
 - `console.log(1)` is executed, printing `1` to the console.
 - `return` is encountered, which immediately exits the function.
3. The `finally` block is executed:
 - `console.log(3)` is executed, printing `3` to the console.

Since `return` is encountered within the `try` block, the control exits the function immediately after `console.log(1)`. The `catch` block is skipped because there are no errors, and the code in the `finally` block is executed regardless of whether an error occurred or not.

So, when you run this code, it will only print `1` and `3` to the console.

56. What's the output of below code ?

```

const nums = [1, 2, 3, 4, 5, 6, 7];
nums.forEach((n) => {
  if(n%2 === 0) {
    break;
  }
  console.log(n);
});

```

Explanation:

Many of you might have thought the output to be 1,2,3,4,5,6,7. But "break" statement works only loops like for, while, do...while and not for map(), forEach(). They are essentially functions by nature which takes a callback and not loops.

✖ Uncaught SyntaxError: Illegal break statement (at [Script snippet #5:4](#) [Script snippet #5:4:6](#))

57. Whats the output of below code ?

```
let a = true;
setTimeout(() => {
  a = false;
}, 2000)

while(a) {
  console.log(' -- inside whilee -- ');
}
```

Solution: <https://medium.com/@iamyashkhandelwal/5-output-based-interview-questions-in-javascript-b64a707f34d2>

This code snippet creates an infinite loop. Let's break it down:

1. `let a = true;` : This declares a variable `a` and initializes it to `true`.
2. `setTimeout(() => { a = false; }, 2000)` : This sets up a timer to execute a function after 2000 milliseconds (2 seconds). The function assigned to `setTimeout` will set the value of `a` to `false` after the timeout.
3. `while(a) { console.log(' -- inside whilee -- '); }` : This is a while loop that continues to execute as long as the condition `a` is `true`. Inside the loop, it prints `' -- inside whilee -- '`.

The issue here is that the while loop runs indefinitely because there's no opportunity for the JavaScript event loop to process the `setTimeout` callback and update the value of `a`. So, even though `a` will eventually become `false`

after 2 seconds, the while loop will not terminate because it doesn't yield control to allow other tasks, like the callback, to execute.

To fix this, you might consider using asynchronous programming techniques like Promises, async/await, or handling the setTimeout callback differently.

58. Whats the output of below code ?

```
setTimeout(() => console.log(1), 0);

console.log(2);

new Promise(res => {
  console.log(3)
  res();
}).then(() => console.log(4));

console.log(5);
```

This code demonstrates the event loop in JavaScript. Here's the breakdown of what happens:

1. `setTimeout(() => console.log(1), 0);` : This schedules a callback function to be executed after 0 milliseconds. However, due to JavaScript's asynchronous nature, it doesn't guarantee that it will execute immediately after the current synchronous code block.
2. `console.log(2);` : This immediately logs `2` to the console.
3. `new Promise(res => { console.log(3); res(); }).then(() => console.log(4));` : This creates a new Promise. The executor function inside the Promise logs `3` to the console and then resolves the Promise immediately with `res()`. The `then()` method is chained to the Promise, so once it's resolved, it logs `4` to the console.
4. `console.log(5);` : This logs `5` to the console.

When you run this code, the order of the output might seem a bit counterintuitive:

```
2
3
5
4
1
```

Here's why:

- `console.log(2);` is executed first because it's synchronous code.
- Then, the Promise executor is executed synchronously, so `console.log(3);` is logged.
- After that, `console.log(5);` is executed.
- Once the current synchronous execution is done, the event loop picks up the resolved Promise and executes its `then()` callback, logging `4`.
- Finally, the callback passed to `setTimeout` is executed, logging `1`. Although it was scheduled to run immediately with a delay of 0 milliseconds, it's still processed asynchronously and placed in the event queue, after the synchronous code has finished executing.

<https://medium.com/@iamyashkhandelwal/5-output-based-interview-questions-in-javascript-b64a707f34d2>

59. Output of below logic ?

```
async function foo() {
  console.log("A");
  await Promise.resolve();
  console.log("B");
  await new Promise(resolve => setTimeout(resolve, 0));
  console.log("C");
}

console.log("D");
```

```
foo();  
console.log("E")
```

Output:

D, A, E, B, C

Explanation:

The main context logs "D" because it is synchronous and executed immediately.

The foo() function logs "A" to the console since it's synchronous and executed immediately. await Promise.resolve();

: This line awaits the resolution of a Promise. The Promise.resolve() function returns a resolved Promise immediately. The control is temporarily returned to the caller function (foo()), allowing other synchronous operations to execute.

Back to the main context: console.log("E");

: This line logs "E" to the console since it's a synchronous operation. The foo() function is still not fully executed, and it's waiting for the resolution of the Promise inside it. Inside foo()

(resumed execution): console.log("B");

: This line logs "B" to the console since it's a synchronous operation.

await new Promise(resolve => setTimeout(resolve, 0));

This line awaits the resolution of a Promise returned by the setTimeout function. Although the delay is set to 0 milliseconds, the setTimeout callback is pushed into the callback queue, allowing the synchronous code to continue.

Back to the main context:

The control is still waiting for the foo() function to complete.

Inside foo() (resumed execution):

The callback from the setTimeout

is picked up from the callback queue, and the promise is resolved. This allows the execution of the next await . console.log("C");

: This line logs "C" to the console since it's a synchronous operation. foo() function completes.

60. Write a program to remove duplicates from an array ?

```
const removeDuplicates = (array) => {
  let uniqueArr = [];

  for (let i = 0; i <= array.length - 1; i++) {
    if (uniqueArr.indexOf(array[i]) === -1) {
      uniqueArr.push(array[i]);
    }
  }

  return uniqueArr;
};

function removeDuplicates(arr) {
  // Use the Set object to remove duplicates. This works beca
  return Array.from(new Set(arr));
}

removeDuplicates([1, 2, 1, 3, 4, 2, 2, 1, 5, 6]);
```

61. How to check whether a string is palindrome or not ?

```
const checkPallindrome = (str) => {
  const len = str.length;

  for (let i = 0; i < len/2; i++) {
    if (str[i] !== str[len - i - 1]) {
      return "Not pallindrome";
    }
  }
}
```

```

    }
    return "pallindrome";
  };

  console.log(checkPallindrome("madam"));

```

62. Program to find longest word in a given sentence ?

```

const findLongestWord = (sentence) => {
  let wordsArray = sentence.split(" ");
  let longestWord = "";

  for (let i = 0; i < wordsArray.length; i++) {
    if (wordsArray[i].length > longestWord.length) {
      longestWord = wordsArray[i];
    }
  }

  console.log(longestWord);
};

findLongestWord("Hi Iam Saikrishna Iam a UI Developer");

```

63. Program to find Reverse of a string without using built-in method ?

```

const findReverse = (sampleString) => {
  let reverse = "";

```



```

    for (let i = sampleString.length - 1; i >= 0; i--) {
        reverse += sampleString[i];
    }
    console.log(reverse);
};

findReverse("Hello Iam Saikrishna Ui Developer");

```

64. Find the max count of consecutive 1's in an array ?

```

const findConsecutive = (array) => {
    let maxCount = 0;
    let currentConsCount = 0;

    for (let i = 0; i <= array.length - 1; i++) {
        if (array[i] === 1) {
            currentConsCount += 1;
            maxCount = Math.max(currentConsCount, maxCount);
        } else {
            currentConsCount = 0;
        }
    }

    console.log(maxCount);
};

findConsecutive([1, 1, 9, 1, 9, 9, 19, 7, 1, 1, 1, 3, 2, 5, 1]
// output: 3

```

65. Find the factorial of given number ?

```
const findFactorial = (num) => {  
  if (num == 0 || num == 1) {  
    return 1;  
  } else {  
    return num * findFactorial(num - 1);  
  }  
};  
  
console.log(findFactorial(4));
```

66. Given 2 arrays that are sorted [0,3,4,31] and [4,6,30]. Merge them and sort [0,3,4,4,6,30,31] ?

```
const sortedData = (arr1, arr2) => {  
  
  let i = 1;  
  let j = 1;  
  let array1 = arr1[0];  
  let array2 = arr2[0];  
  
  let mergedArray = [];  
  
  while(array1 || array2){  
  
    if(array2 === undefined || array1 < array2){  
      mergedArray.push(array1);  
      array1 = arr1[i];  
      i++;  
    }else{  
      mergedArray.push(array2);  
      array2 = arr2[j];  
    }  
  }  
}
```

```

        j++;
    }

    console.log(mergedArray)

}

sortedData([1, 3, 4, 5], [2, 6, 8, 9])

```

67. Create a function which will accepts two arrays arr1 and arr2. The function should return true if every value in arr1 has its corresponding value squared in array2. The frequency of values must be same. (Effecient)

Inputs and outputs:

=====

[1,2,3],[4,1,9] ⇒ true

[1,2,3],[1,9] ==⇒ false

[1,2,1],[4,4,1] ==⇒ false (must be same frequency)

```

function isSameFrequency(arr1, arr2){

    if(arr1.length !== arr2.length){
        return false;
    }

    let arrFreq1={};
    let arrFreq2={};

    for(let val of arr1){

```

```

    arrFreq1[val] = (arrFreq1[val] || 0) + 1;
  }

  for(let val of arr2){
    arrFreq2[val] = (arrFreq2[val] || 0) + 1;
  }

  for(let key in arrFreq1){
    if(!key*key in arrFreq2) return false;
    if(arrFreq1[key] !== arrFreq2[key*key]) return false
  }
  return true;
}

console.log(isSameFrequency([1, 2, 5], [25, 4, 1]))

```

68. Given two strings. Find if one string can be formed by rearranging the letters of other string. (Efficient)

Inputs and outputs:

"aaz","zza" ⇒ false

"qwerty","qeywrt" ⇒ true

```

function isStringCreated(str1, str2){
  if(str1.length !== str2.length) return false
  let freq = {};

  for(let val of str1){
    freq[val] = (freq[val] || 0) + 1;
  }

  for(let val of str2){
    if(freq[val]){

```

```

        freq[val] -= 1;
    } else{
        return false;
    }
}
return true;
}

console.log(isStringCreated('anagram', 'nagaram'))

```

69. Write logic to get unique objects from below array ?

I/P: [{name: "sai"},{name:"Nang"},{name: "sai"},{name:"Nang"},{name: "111111"}];

O/P: [{name: "sai"},{name:"Nang"}]{name: "111111"}

```

function getUniqueArr(array){
    const uniqueArr = [];
    const seen = {};
    for(let i=0; i<=array.length-1;i++){
        const currentItem = array[i].name;
        if(!seen[currentItem]){
            uniqueArr.push(array[i]);
            seen[currentItem] = true;
        }
    }
    return uniqueArr;
}

let arr = [{name: "sai"}, {name:"Nang"}, {name: "sai"}, {name:"Nang"}, {name: "111111"}];
console.log(getUniqueArr(arr))

```

70. Write a JavaScript program to find the maximum number in an array.

```
function findMax(arr) {
    if (arr.length === 0) {
        return undefined; // Handle empty array case
    }

    let max = arr[0]; // Initialize max with the first element

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i]; // Update max if current element is greater
        }
    }

    return max;
}

// Example usage:
const numbers = [1, 6, -33, 9, 4, 8, 2];
console.log("Maximum number is:", findMax(numbers));
```

Time complexity: $O(N)$

71. Write a JavaScript function that takes an array of numbers and returns a new array with only the even numbers.

```
function findEvenNumbers(arr) {
    const result = [];

    for (let i = 0; i < arr.length; i++) {
        if (arr[i] % 2 === 0) {
            result.push(arr[i]); // Add even numbers to the result array
        }
    }

    return result;
}
```

```

}

// Example usage:
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, -8, 19, 9, 10];
console.log("Even numbers:", findEvenNumbers(numbers));

```

Time complexity: $O(N)$

72. Write a JavaScript function to check if a given number is prime.

```

function isPrime(number) {
    if (number <= 1) {
        return false; // 1 and numbers less than 1 are not prime
    }

    // Loop up to the square root of the number
    for (let i = 2; i <= Math.sqrt(number); i++) {
        if (number % i === 0) {
            return false; // If divisible by any number, not prime
        }
    }

    return true; // If not divisible by any number, it's prime
}

// Example usage:
console.log(isPrime(17)); // true
console.log(isPrime(19)); // false

```

Time complexity: $O(N)$

73. Write a JavaScript program to find the largest element in a nested array.

```

function findLargestElement(arr) {
    let max = Number.NEGATIVE_INFINITY; // Initialize max to

    // Helper function to traverse nested arrays
    function traverse(arr) {
        for (let i = 0; i < arr.length; i++) {
            if (Array.isArray(arr[i])) {
                // If element is an array, recursively call t
                traverse(arr[i]);
            } else {
                // If element is not an array, update max if
                if (arr[i] > max) {
                    max = arr[i];
                }
            }
        }
    }

    // Start traversing the input array
    traverse(arr);

    return max;
}

// Example usage:
const nestedArray = [[3, 4, 58], [709, 8, 9, [10, 11]], [111,
console.log("Largest element:", findLargestElement(nestedArray));

```

Time complexity: $O(N)$

74. Write a JavaScript function that returns the Fibonacci sequence up to a given number of terms.

```

function fibonacciSequence(numTerms) {
    if (numTerms <= 0) {
        return [];
    }
}

```



```

    } else if (numTerms === 1) {
        return [0];
    }

    const sequence = [0, 1];

    for (let i = 2; i < numTerms; i++) {
        const nextFibonacci = sequence[i - 1] + sequence[i - 2];
        sequence.push(nextFibonacci);
    }

    return sequence;
}

// Example usage:
const numTerms = 10;
const fibonacciSeries = fibonacciSequence(numTerms);
console.log(fibonacciSeries); // Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

```

Time complexity: $O(N)$

75. Given a string, write a javascript function to count the occurrences of each character in the string.

```

function countCharacters(str) {
    const charCount = {}; // Object to store character counts
    const len = str.length;

    // Loop through the string and count occurrences of each character
    for (let i = 0; i < len; i++) {
        const char = str[i];
        // Increment count for each character
        charCount[char] = (charCount[char] || 0) + 1;
    }

    return charCount;
}

```

```
// Example usage:
const result = countCharacters("hellaalo");
console.log(result); // Output: { h: 1, e: 1, l: 2, o: 1 }
```

Time complexity: $O(N)$

76. Write a javascript function that sorts an array of numbers in ascending order.

```
function quickSort(arr) {
    // Check if the array is empty or has only one element
    if (arr.length <= 1) {
        return arr;
    }

    // Select a pivot element
    const pivot = arr[0];

    // Divide the array into two partitions
    const left = [];
    const right = [];

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }

    // Recursively sort the partitions
    const sortedLeft = quickSort(left);
    const sortedRight = quickSort(right);

    // Concatenate the sorted partitions with the pivot and return
    return sortedLeft.concat(pivot, sortedRight);
}
```

```

}

// Example usage:
const unsortedArray = [5, 2, 9, 1, 3, 6];
const sortedArray = quickSort(unsortedArray);
console.log(sortedArray); // Output: [1, 2, 3, 5, 6, 9]

```

Time complexity: $O(n \log n)$

77. Write a javascript function that sorts an array of numbers in descending order.

```

function quickSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }

  const pivot = arr[0];
  const left = [];
  const right = [];

  for (let i = 1; i < arr.length; i++) {
    if (arr[i] >= pivot) {
      left.push(arr[i]);
    } else {
      right.push(arr[i]);
    }
  }

  return [...quickSort(left), pivot, ...quickSort(right)];
}

const arr = [3, 1, 4, 1, 5, 9, 2, 6, 5];
const sortedArr = quickSort(arr);
console.log(sortedArr); // Output: [9, 6, 5, 5, 4, 3, 2, 1, 1]

```

Time complexity: $O(n \log n)$

78. Write a javascript function that reverses the order of words in a sentence without using the built-in reverse() method.

```
const reverseWords = (sampleString) => {
  let reversedSentence = "";
  let word = "";

  // Iterate over each character in the sampleString
  for (let i = 0; i < sampleString.length; i++) {
    // If the character is not a space, append it to the current word
    if (sampleString[i] !== ' ') {
      word += sampleString[i];
    } else {
      // If it's a space, prepend the current word to the reversed sentence
      //reset the word
      reversedSentence = word + ' ' + reversedSentence;
      word = "";
    }
  }

  // Append the last word to the reversed sentence
  reversedSentence = word + ' ' + reversedSentence;

  // Trim any leading or trailing spaces and log the result
  console.log(reversedSentence.trim());
};

// Example usage
reverseWords("ChatGPT is awesome"); // "awesome is ChatGPT"
```

```
function reverseWords(sentence) {
  // Split the sentence into words
  let words = [];
  let wordStart = 0;
  for (let i = 0; i < sentence.length; i++) {
```

```

        if (sentence[i] === ' ') {
            words.unshift(sentence.substring(wordStart, i));
            wordStart = i + 1;
        } else if (i === sentence.length - 1) {
            words.unshift(sentence.substring(wordStart, i + 1));
        }
    }

    // Join the words to form the reversed sentence
    return words.join(' ');
}

// Example usage
const sentence = "ChatGPT is awesome";
console.log(reverseWords(sentence)); // Output: "awesome is C

```

Time complexity: $O(N)$

79. Implement a javascript function that flattens a nested array into a single-dimensional array.

```

function flattenArray(arr) {
    const stack = [...arr];
    const result = [];

    while (stack.length) {
        const next = stack.pop();
        if (Array.isArray(next)) {
            stack.push(...next);
        } else {
            result.push(next);
        }
    }

    return result.reverse(); // Reverse the result to maintain order
}

```

```
// Example usage:
const nestedArray = [1, [2, [3, 4], [7, 5]], 6];
const flattenedArray = flattenArray(nestedArray);
console.log(flattenedArray); // Output: [1, 2, 3, 4, 5, 6]
```

80. Write a function which converts string input into an object

```
// stringToObject("a.b.c", "someValue");
```

```
// output → {a: {b: {c: "someValue"}}}
```

```
function stringToObject(str, finalValue) {
  const keys = str.split('.');
  let result = {};
  let current = result;

  for (let i = 0; i < keys.length; i++) {
    const key = keys[i];
    current[key] = (i === keys.length - 1) ? finalValue : {};
    current = current[key];
  }

  return result;
}

// Test the function
const output = stringToObject("a.b.c", "someValue");
console.log(output); // Output: {a: {b: {c: "someValue"}}}
```

81. Given an array, return an array where the each value is the product of the next two items: E.g. [3, 4, 5] -> [20, 15, 12]

```
function productOfNextTwo(arr) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    if (i < arr.length - 1) {
      result.push(arr[i + 1] * arr[i + 2]);
    } else {
      result.push(arr[0] * arr[1]);
    }
  }
  return result;
}

// Example usage:
const inputArray = [3, 4, 5];
const outputArray = productOfNextTwo(inputArray);
console.log(outputArray); // Output: [20, 15, 12]
```

82. Guess the output ?

```
let output = (function(x){
  delete x;
  return x;
})(3);
console.log(output);
```

Output: 3

Let me break it down for you:

1. The code defines an immediately invoked function expression (IIFE) that takes a parameter `x`.
2. Inside the function, `delete x;` is called. However, `delete` operator is used to delete properties from objects, not variables. When you try to delete a variable, it doesn't actually delete the variable itself, but it's syntactically

incorrect and may not have any effect depending on the context (in strict mode, it throws an error). So, `delete x;` doesn't do anything in this case.

3. Finally, the function returns `x`. Since `x` was passed as `3` when calling the function `(function(x){ ... })(3)`, it returns `3`.
 4. The returned value is assigned to the variable `output`.
 5. `console.log(output);` then logs the value of `output`, which is `3`.
-

83. Guess the output of below code ?

```
for (var i = 0; i < 3; i++) {  
  setTimeout(function () {  
    console.log(i);  
  }, 1000 + i);  
}
```

Output: 3 3 3

This might seem counterintuitive at first glance, but it's due to how JavaScript handles closures and asynchronous execution.

Here's why:

1. The `for` loop initializes a variable `i` to `0`.
2. It sets up a timeout for `i` milliseconds plus the current value of `i`, which means the timeouts will be `1000`, `1001`, and `1002` milliseconds.
3. After setting up the timeouts, the loop increments `i`.
4. The loop checks if `i` is still less than `3`. Since it's now `3`, the loop exits.

When the timeouts execute after their respective intervals, they access the variable `i` from the outer scope. At the time of execution, `i` is `3` because the loop has already finished and incremented `i` to `3`. So, all three timeouts log `3`.

34. Guess the output ?


```
let output = (function(x){
  delete x;
  return x;
})(3);
console.log(output);
```

Output: 3

Let me break it down for you:

1. The code defines an immediately invoked function expression (IIFE) that takes a parameter `x`.
2. Inside the function, `delete x;` is called. However, `delete` operator is used to delete properties from objects, not variables. When you try to delete a variable, it doesn't actually delete the variable itself, but it's syntactically incorrect and may not have any effect depending on the context (in strict mode, it throws an error). So, `delete x;` doesn't do anything in this case.
3. Finally, the function returns `x`. Since `x` was passed as `3` when calling the function `(function(x){ ... })(3)`, it returns `3`.
4. The returned value is assigned to the variable `output`.
5. `console.log(output);` then logs the value of `output`, which is `3`.

85. Guess the output ?

```
let c=0;

let id = setInterval(() => {
  console.log(c++)
}, 10)

setTimeout(() => {
  clearInterval(id)
}, 2000)
```

This JavaScript code sets up an interval that increments the value of `c` every 200 milliseconds and logs its value to the console. After 2 seconds (2000 milliseconds), it clears the interval.

Here's what each part does:

- `let c = 0;` : Initializes a variable `c` and sets its initial value to 0.
- `let id = setInterval(() => { console.log(c++) }, 200)` : Sets up an interval that executes a function every 200 milliseconds. The function logs the current value of `c` to the console and then increments `c`.
- `setTimeout(() => { clearInterval(id) }, 2000)` : Sets a timeout function that executes after 2000 milliseconds (2 seconds). This function clears the interval identified by `id`, effectively stopping the logging of `c`.

This code essentially logs the values of `c` at 200 milliseconds intervals until 2 seconds have passed, at which point it stops logging.

86. What would be the output of following code ?

```
function getName1(){
    console.log(this.name);
}

Object.prototype.getName2 = () =>{
    console.log(this.name)
}

let personObj = {
    name: "Tony",
    print: getName1
}

personObj.print();
personObj.getName2();
```

Output: Tony undefined

Explanation: getName1() function works fine because it's being called from **personObj**, so it has access to **this.name** property. But when while calling **getName2** which is defined under **Object.prototype** doesn't have any property named **this.name**. There should be **name** property under prototype. Following is the code:

```
function getName1(){
    console.log(this.name);
}

Object.prototype.getName2 = () =>{
    console.log(Object.getPrototypeOf(this).name);
}

let personObj = {
    name: "Tony",
    print: getName1
}

personObj.print();
Object.prototype.name = "Steve";
personObj.getName2();
```

87. What is the main difference between Local Storage and Session storage ?

- LocalStorage is the same as SessionStorage but it persists the data even when the browser is closed and reopened(i.e it has no expiration time) whereas in sessionStorage data gets cleared when the page session ends.

88. Can you find is there any security issue in the javascript code?

```
const data = await fetch("api");
const div = document.getElementById("todo")
div.innerHTML = data;
```

The provided JavaScript code seems straightforward, but there's a potential security issue related to how it handles data from the API response.

1. Cross-Site Scripting (XSS):

The code directly assigns the fetched data (`data`) to the `innerHTML` property of the `div` element. If the data fetched from the API contains untrusted or user-controlled content (such as user-generated content or content from a third-party API), it could potentially contain malicious scripts. Assigning such data directly to `innerHTML` can lead to XSS vulnerabilities, as it allows execution of arbitrary scripts in the context of the page.

To mitigate this security risk, you should properly sanitize or escape the data before assigning it to `innerHTML`, or consider using safer alternatives like `textContent` or creating DOM elements programmatically.

Here's an example of how you could sanitize the data using a library like DOMPurify:

```
javascriptCopy code
const data = await fetch("api");
const div = document.getElementById("todo");
data.text().then(text => {
  div.innerHTML = DOMPurify.sanitize(text);
});
```

By using `DOMPurify.sanitize()`, you can ensure that any potentially harmful content is removed or escaped, reducing the risk of XSS attacks. Make sure to include the DOMPurify library in your project if you choose to use it.

Always remember to validate and sanitize any data that originates from external sources before inserting it into your DOM.

39. What is eval() ?

- eval function evaluates javascript code represented as a string. The string can be javascript expression, variable, statement or a sequence of statements.

```
console.log(eval("1 + 2")); // 3
```

90. What is the difference between Shallow copy and deep copy ?

- **Shallow copy:**

- A shallow copy creates a new object or array and copies the references of the original elements

```
let originalArray = [1, 2, [3, 4]];
let shallowCopy = [...originalArray];

shallowCopy[2][0] = 100;
console.log(originalArray); // Output: [1, 2, [100, 4]]
```

- **Deep copy:**

- A deep copy creates a new object or array that has its own copies of the properties of the original object.

```
let originalArray = [1, 2, [3, 4]];
let deepCopy = JSON.parse(JSON.stringify(originalArray));
deepCopy[2][0] = 100;
console.log(originalArray); // Output: [1, 2, [3, 4]]
```

91. What would be the output of following code ?

```
function test() {
  console.log(a);
  console.log(foo());
  var a = 1;
  function foo() {
    return 2;
  }
}

test();
```

Output: undefined and 2

In JavaScript, this code will result in `undefined` being logged for `console.log(a)` and `2` being logged for `console.log(foo())`. This is due to variable hoisting and function declaration hoisting.

Here's what's happening step by step:

1. The `test` function is called.
2. Inside `test` :
 - `console.log(a)` is executed. Since `a` is declared later in the function, it's hoisted to the top of the function scope, but not initialized yet. So, `a` is `undefined` at this point.
 - `console.log(foo())` is executed. The `foo` function is declared and assigned before it's called, so it returns `2`.
 - `var a = 1;` declares and initializes `a` with the value `1`.

Therefore, when `console.log(a)` is executed, `a` is `undefined` due to hoisting, and when `console.log(foo())` is executed, it logs `2`, the return value of the `foo` function.

92. What are the difference between undeclared and undefined variables ?

- undeclared:

- These variables does not exist in the program and they are not declared.
 - If we try to read the value of undeclared variable then we will get a runtime error.
 - **undefined:**
 - These variables are declared in the program but are not assigned any value.
 - If we try to access the value of undefined variables, It will return undefined.
-

93. What is event bubbling

- Event bubbling is a type of event propagation where the event first triggers on the innermost target element, and then successively triggers on the ancestors (parents) of the target element in the same nesting hierarchy till it reaches the outermost DOM element.
-

94. What is event capturing ?

Event capturing is a type of event propagation where the event is first captured by the outermost element, and then successively triggers on the descendants (children) of the target element in the same nesting hierarchy till it reaches the innermost DOM element.

95. What is the output of below logic ?

```
function job(){  
  return new Promise((resolve, reject)=>{  
    reject()  
  })  
}
```

```

    })
  }

  let promise = job();

  promise.then(()=>{
    console.log("1111111111")
  }).then(()=>{
    console.log("2222222222")
  }).catch(()=>{
    console.log("3333333333")
  }).then(()=>{
    console.log("4444444444")
  })
}

```

In this code, a Promise is created with the `job` function. Inside the `job` function, a Promise is constructed with the executor function that immediately rejects the Promise.

Then, the `job` function is called and assigned to the variable `promise`.

After that, a series of `then` and `catch` methods are chained to the `promise`:

1. The first `then` method is chained to the `promise`, but it is not executed because the Promise is rejected, so the execution jumps to the `catch` method.
2. The `catch` method catches the rejection of the Promise and executes its callback, logging "3333333333".
3. Another `then` method is chained after the `catch` method. Despite the previous rejection, this `then` method will still be executed because it's part of the Promise chain, regardless of previous rejections or resolutions. It logs "4444444444".

So, when you run this code, you'll see the following output:

```

3333333333
4444444444

```


96. Guess the output ?

```
var a = 1;

function data() {
  if(!a) {
    var a = 10;
  }
  console.log(a);
}

data();
console.log(a);
```

Explanation:

```
var a = 1;

function toTheMoon() {
  var a; // var has function scope, hence it's declaration
  if(!a) {
    a = 10;
  }
  console.log(a); // 10 precedence will be given to local
}

toTheMoon();
console.log(a); // 1 refers to the `a` defined at the top
```

97. Tests your array basics

```
function guessArray() {
  let a = [1, 2];
  let b = [1, 2];

  console.log(a == b);
  console.log(a === b);
}

guessArray();
```

In JavaScript, when you compare two arrays using the `==` or `===` operators, you're comparing their references, not their contents. So, even if two arrays have the same elements, they will not be considered equal unless they refer to the exact same object in memory.

In your `guessArray` function, `a` and `b` are two separate arrays with the same elements, but they are distinct objects in memory. Therefore, `a == b` and `a === b` will both return `false`, indicating that `a` and `b` are not the same object.

If you want to compare the contents of the arrays, you'll need to compare each element individually.

98. Test your basics on comparison ?

```
let a = 3;
let b = new Number(3);
let c = 3;

console.log(a == b);
console.log(a === b);
console.log(b === c);
```

`new Number()` is a built-in function constructor. Although it looks like a number, it's not really a number: it has a bunch of extra features and is an

object.

When we use the `==` operator (Equality operator), it only checks whether it has the same *value*. They both have the value of `3`, so it returns `true`.

However, when we use the `===` operator (Strict equality operator), both value *and* type should be the same. It's not: `new Number()` is not a number, it's an **object**. Both return `false`.

34 Reactjs Interview questions & Answers

99. What is React?

- React is an opensource component based JavaScript library which is used to develop interactive user interfaces.

100. What is useMemo ?

- useMemo is useful for performance optimization in react.
- It is used to cache the result of a function between re-renders.
- Example :
 - In our application we have a data vizualization component where we need to display charts based on performing complex calculations on some large data sets. By using useMemo we can cache the computed result, which ensures that the component does not recalculate on every re-renders.
 - This saves computational resources and provides smoother user experience.

```
import React, { useMemo } from 'react';
```

```
const DataVisualization = ({ data }) => {  
  const processedData = useMemo(() => {  
    // Perform expensive computations on data  
    // ...  
    return processedData;  
  }, [data]);  
  
  // Render the visualization using the processed data  
  // ...  
  
  return <div>{/* Visualization component */}</div>;  
};
```

In this example, the `processedData` is memoized using `useMemo` to avoid recomputing it on every render. The expensive computations are performed only when the `data` prop changes.

101. What are the features of React ?

- JSX
 - Virtual dom
 - one way data binding
 - Uses reusable components to develop the views
 - Supports server side rendering
-

102. What is JSX ?

- JSX means javascript xml. It allows the user to write the code similar to html in their javascript files.
- This jsx will be transpiled into javascript that interacts with the browser when the application is built.

103. What is DOM ?

- DOM means document object model. It is like a tree like structure that represents the elements of a webpage.
-

104. What is Virtual Dom ?

- When ever any underlying data changes or whenever user enters something in textbox then entire UI is rerendered in a virtual dom representation.
 - Now this virtual dom is compared with the original dom and creates a changeset which will be applied on the real dom.
 - So instead of updating the entire realdom, it will be updated with only the things that have actually been changed.
-

105. What is component life cycle of React class component ?

- React life cycle consists of 3 phases.
 - mounting
 - updating
 - unmounting
- **Mounting:**
 - In this phase the component is generally mounted into the dom.
 - It is an initialization phase where we can do some operations like getting data from api, subscribing to events etc.
- 1. **Constructor:**
 - It is a place to set the initial state and other initial values.

2. getDerivedStateFromProps:

- This is called right before rendering the elements into the dom.
- Its a natural place to set the state object based on the initial props.
- It takes state as an argument and returns an object with changes to the state.

```
getDerivedStateFromProps(props, state){  
  return { favColor: props.favColor }  
}
```

3. render():

- It contains all the html elements and is method that actually outputs the html to the dom.

4. ComponentDidMount():

- This is called once component is mounted into the dom.
- Eg: fetch api calls, subscribing to events etc.

◦ Updating phase:

- This is when the component is updated. The component will be updated when ever there is change in state or props.

1. getDerivedStateFromProps: same as above

2. ShouldComponentUpdate:

- This will return boolean value that specifies whether react should continue with the rendering or not. default is true.

```
shouldComponentUpdate(){  
  return true/false  
}
```

3. Render: same as above

4. getSnapshotBeforeUpdate:

- It will have access to the props and state before update. means that even after the update you can check what are the values were before update.

```
getSnapshotBeforeUpdate(prevProps, prevState){  
  console.log(prevProps, prevState)  
}
```

5. ComponentDidUpdate:

- Called after the component is updated in the dom.

◦ Unmounting phase:

- In this phase the component will be removed from the dom. here we can do unsubscribe to some events or destroying the existing dialogs etc.

1. ComponentWillUnmount:

- This is called when component is about to be removed from the dom.

106. What are fragments in react ?

- React fragments allows us to wrap or group multiple elements without adding extra nodes to the dom.

107. What are props in react ?

- Props are input to a component.
 - They are used to send data from parent component to child component.
-

108. What are synthetic events in react ?

- Synthetic events are the wrapper around native browser events.
 - By using this react will abstract browser inconsistencies and provides a unified api interface for event handling
 - They optimise the performance by event pooling and reusing event objects.
-

109. What are the difference between Package.json and Package.lock.json

- **Package.json**: is a metadata file that contains information about your project, such as the name, version, description, author, and most importantly, the list of dependencies required for your project to run. This file is used by npm (Node Package Manager) to install, manage, and update dependencies for your project.
 - **Package.lock.json**: is a file that npm generates after installing packages for your project. This file contains a detailed description of the dependencies installed in your project, including their versions and the dependencies of their dependencies. This file is used by npm to ensure that the same version of each package is installed every time, regardless of the platform, environment, or the order of installation.
 - Package.json is used to define the list of required dependencies for your project, while package-lock.json is used to ensure that the exact same versions of those dependencies are installed every time, preventing version conflicts and guaranteeing a consistent environment for your project.
-

110. What are the differences between client side and server side rendering ?

- **Rendering location:** In csr, rendering occurs on the client side after receiving raw data from the server where as in SSR, rendering occurs on server side and server returns the fully rendered HTML page to the browser.
 - **Initial Load time:** CSR has slow initial load time as browser needs to interpret the data and render the page. Whereas SSR has faster initial load times as server sends pre-rendered HTML page to the browser.
 - **Subsequent interactions:** Subsequent interactions in CSR involve dynamic updates without requiring full page reload whereas, SSR requires full page reload as server generates new HTML for every interaction.
 - **SEO:** SSR is SEO friendly when compared to CSR as fully rendered HTML content is provided to the search engine crawlers whereas CSR needs to parse JavaScript-heavy content.
-

111. What is state in Reactjs?

- State is an object which holds the data related to a component that may change over the lifetime of a component.
- When the state changes, the component re-renders.
- Eg: for functional component and class component

```
import React, { useState } from "react";

function User() {
  const [message, setMessage] = useState("Welcome to React");

  return (
    <div>
      <h1>{message}</h1>
    </div>
  );
}
```

```

import React from 'react';
class User extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      message: "Welcome to React world",
    };
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    );
  }
}

```

112. What are props ?

- Props are inputs to the component.
- They are used to send data from parent component to child component.
- Props are immutable, so they cannot be modified directly within the child component.
- **Example:**

```

// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const name = "John";

```

```

    return (
      <div>
        <h1>Parent Component</h1>
        <ChildComponent name={name} />
      </div>
    );
  }

  export default ParentComponent;

```

```

// ChildComponent.js
import React from 'react';

function ChildComponent(props) {
  return (
    <div>
      <h2>Child Component</h2>
      <p>Hello, {props.name}!</p>
    </div>
  );
}

export default ChildComponent;

```

113. What are the differences between State and Props in react ?

- Both props and state are used to manage the data of a component.
- State is used to hold the data of a component whereas props are used to send data from one component to another component.
- State is mutable but props are immutable.
- Any change in state causes rerender of component and its children.

114. What is props drilling ?

- Props drilling is the process of sending the data from one component to the component that needs the data from several interconnected components
-

115. What are the disadvantages of props drilling and How we can avoid props drilling ?

- **Code complexity:**

Prop drilling can make code difficult to read and maintain, especially in large applications with many components. This is because props need to be passed down through multiple levels of components, and it can be difficult to keep track of which components are using which props.

- **Reduced maintainability:**

Prop drilling can also make code less maintainable. This is because if a prop needs to be changed, the change needs to be propagated through all of the components that use it. This can be a time-consuming and error-prone process.

- **Increased risk of errors:**

Prop drilling can also increase the risk of errors. This is because it can be difficult to keep track of which components are using which props, and it can be easy to forget to pass a prop down to a component that needs it. This can lead to errors in the application.

- **Performance overhead:**

Prop drilling can also have a performance overhead. This is because every time a prop is passed down to a component, the component needs to re-render. This can be a significant performance overhead in large applications with many components.

Makes application slower.

We can avoid props drilling using context api or Redux or by using any state management libraries.

116. What are Pure components in React ?

- A component which renders the same output for the same props and state is called as pure component.
- It internally implements **shouldComponentUpdate** lifecycle method and performs a shallow comparison on the props and state of the component. If there is no difference, the component is not re-rendered.
- **Advantage:**
 - It optimizes the performance by reducing unnecessary re-renders.

Example for class component:

```
import React, { PureComponent } from 'react';

class MyPureComponent extends PureComponent {
  render() {
    return (
      <div>
        <h1>Pure Component Example</h1>
        <p>Props: {this.props.text}</p>
      </div>
    );
  }
}

export default MyPureComponent;
```

Reference: <https://react.dev/reference/react/PureComponent>

```
import { PureComponent, useState } from 'react';
```

```

class Greeting extends PureComponent {
  render() {
    console.log("Greeting was rendered at", new Date().toLocaleTimeString());
    return <h3>Hello{this.props.name}&& ', '{this.props.name}!</h3>;
  }
}

export default function MyApp() {
  const [name, setName] = useState('');
  const [address, setAddress] = useState('');
  return (
    <>
      <label>
        Name{' ': ' '}
        <input value={name} onChange={e => setName(e.target.value)} />
      </label>
      <label>
        Address{' ': ' '}
        <input value={address} onChange={e => setAddress(e.target.value)} />
      <Greeting name={name} />
    </>
  );
}

```

117. What are Ref's in React?

- ref's are the way to access the dom elements created in the render method.
- they are helpful when we want to update the component without using state and props and prevents triggering rerender.

Common useCases:

- Managing input focus and text selection
- Media control/Playback

- Communicating between react components that are not directly related through the component tree.

Examples:

1. Managing input focus

```
function App() {  
  const inputRef = useRef();  
  
  const focusOnInput = () => {  
    inputRef.current.focus();  
  };  
  
  return (  
    <div>  
      <input type='text' ref={inputRef} />  
      <button onClick={focusOnInput}>Click Me</button>  
    </div>  
  );  
}
```

2. Managing Audio playback:

```
function App() {  
  const audioRef = useRef();  
  
  const playAudio = () => {  
    audioRef.current.play();  
  };  
  
  const pauseAudio = () => {  
    audioRef.current.pause();  
  };  
  
  return (  
    <div>  
      <audio  
        ref={audioRef}  
        type='audio/mp3'  

```

```

        src='https://s3-us-west-2.amazonaws.com/s.cdpn.io/
    ></audio>
    <button onClick={playAudio}>Play Audio</button>
    <button onClick={pauseAudio}>Pause Audio</button>
  </div>
);
}

```

Reference: <https://www.memberstack.com/blog/react-refs>

118. What is meant by forward ref ?

- In React, forwardref is a technique which is used to send the ref from parent component to one of its children. This is helpful when we want to access the child component dom node from the parent component.

Example:

1. **Creating the Forward Ref Component:** Define a component that forwards the ref to a child component.

```

import React, { forwardRef } from 'react';

const ChildComponent = forwardRef((props, ref) => {
  return <input ref={ref} />;
});

export default ChildComponent;

```

2. **Using the Forward Ref Component:** Use this component and pass a ref to it.

```

import React, { useRef } from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const inputRef = useRef(null);

```



```

    return (
      <div>
        <ChildComponent ref={inputRef} />
        <button onClick={() => inputRef.current.focus()}>Foc
      </div>
    );
  }

  export default ParentComponent;

```

In this example, `ChildComponent` is a functional component that forwards the ref it receives to the `input` element it renders. Then, in the `ParentComponent`, a ref is created using `useRef`, passed to `ChildComponent`, and used to focus the input when a button is clicked.

Reference: <https://codedamn.com/news/reactjs/what-are-forward-refs-in-react-js>

119. What are Error boundaries ?

- Error boundaries are one of the feature in react which allows the developers to catch the errors during the rendering of component tree, log those errors and handle those errors without crashing the entire application by displaying a fallback ui.

```

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback
    return { hasError: true };
  }

```

```

componentDidCatch(error, info) {
  // Example "componentStack":
  //   in ComponentThatThrows (created by App)
  //   in ErrorBoundary (created by App)
  //   in div (created by App)
  //   in App
  logErrorToMyService(error, info.componentStack);
}

render() {
  if (this.state.hasError) {
    // You can render any custom fallback UI
    return this.props.fallback;
  }

  return this.props.children;
}
}

```

Then you can wrap a part of your component tree with it:

```

<ErrorBoundary fallback={<p>Something went wrong</p>}>
  <Profile />
</ErrorBoundary>

```

120. What are Higher order components in react ?

- Higher order component is a function which takes the component as an argument and returns a new component that wraps the original component.

For example if we wanted to add a some style to multiple components in our application, Instead of creating a `style` object locally each time, we can create a HOC that adds the `style` objects to the component that we pass to it.

```
function withStyles(Component) {
  return props => {
    const style = { padding: '0.2rem', margin: '1rem' }
    return <Component style={style} {...props} />
  }
}

const Button = () = <button>Click me!</button>
const Text = () => <p>Hello World!</p>

const StyledButton = withStyles(Button)
const StyledText = withStyles(Text)
```

121. What are the differences between controlled and uncontrolled components ?

◦ Controlled components:

- In controlled components, the form data is handled by the react component
- We use event handlers to update the state.
- React state is the source of truth.

◦ Uncontrolled components:

- In uncontrolled components, the form data is handled by the dom.
- We use Ref's to update the state.
- Dom is the source of truth.

feature	uncontrolled	controlled
one-time value retrieval (e.g. on submit)	✓	✓
validating on submit	✓	✓
instant field validation	✗	✓
conditionally disabling submit button	✗	✓
enforcing input format	✗	✓
several inputs for one piece of data	✗	✓
dynamic inputs	✗	✓

Ref: <https://goshacmd.com/controlled-vs-uncontrolled-inputs-react/>

122. What is useCallback ?

- useCallback caches the function definition between the re-renders
- It takes two arguments: the callback function and an array of dependencies. The callback function is only recreated if one of the dependencies has changed.

Good Ref: <https://deadsimplechat.com/blog/usecallback-guide-use-cases-and-examples/#the-difference-between-usecallback-and-declaring-a-function-directly>

123. What are the differences between useMemo and useCallback ?

- Both useMemo and useCallback are useful for performance optimization.

- useMemo will cache the result of the function between re-renders whereas useCallback will cache the function itself between re-renders.
-

124. What are keys in React ?

- Keys are used to uniquely identify the elements in the list.
- react will use this to indentify, which elements in the list have been added, removed or updated.

```
function MyComponent() {
  const items = [
    { id: 1, name: "apple" },
    { id: 2, name: "banana" },
    { id: 3, name: "orange" }
  ];

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

125. What is Lazy loading in React ?

- It is a technique used to improve the performance of a webapplication by splitting the code into smaller chunks and loading them only when its required intead of loading on initial load.

```
import React, { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'))

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

126. What is suspense in React ?

The lazy loaded components are wrapped by **Suspense**. The Suspense component receives a fallback prop which is displayed until the lazy loaded component is rendered.

```
import React, { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'))

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}
```

```
}  
  
export default App;
```

127. What are custom hooks ?

- Custom hooks helps us to extract and reuse the stateful logic between components.
- Eg:
 - Fetch data
 - To find user is in online or offline.

<https://react.dev/learn/reusing-logic-with-custom-hooks>

```
import { useState, useEffect } from 'react';  
  
// Custom hook to fetch data from an API  
function useFetch(url) {  
  const [data, setData] = useState(null);  
  const [loading, setLoading] = useState(true);  
  const [error, setError] = useState(null);  
  
  useEffect(() => {  
    const fetchData = async () => {  
      try {  
        const response = await fetch(url);  
        if (!response.ok) {  
          throw new Error('Network response was not ok');  
        }  
        const result = await response.json();  
        setData(result);  
      } catch (error) {  
        setError(error);  
      } finally {  
        setLoading(false);  
      }  
    }  
  });  
}
```

```

    }
  };

  fetchData();

  // Cleanup function
  return () => {
    // Cleanup logic if needed
  };
}, [url]); // Dependency array to watch for changes in t

return { data, loading, error };
}

// Example usage of the custom hook
function MyComponent() {
  const { data, loading, error } = useFetch('https://api.e

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <div>
      {data && (
        <ul>
          {data.map(item => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      )}
    </div>
  );
}

```



```
export default MyComponent;
```

128. What is useReducer hook ?

- It is an alternative to useState hook which is used when state of the component is complex and requires more than one state variable.

129. **Practical question:** Create a increment decrement counter using useReducer hook in react ?

```
import React, { useReducer } from 'react';

// Initial state
const initialState = {
  count: 0
};

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

// Component
```

```
const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState)

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' } )}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' } )}>Decrement</button>
    </div>
  );
};

export default Counter;
```

130. What is context in react ?

- It is a way of managing state globally in react.
- By using context we can share data between nested components globally with out having to manually pass data as props down at every level of component tree.
- It solves the problem of props drilling in react.

131. Give an example of context api usage ?

```
import React, { createContext, useState, useContext } from 'react';

// Create a context
const UserContext = createContext();

// Create a provider component
```

```

const UserProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  const login = (username, password) => {
    // Simulate login logic, for example purposes
    setUser({ username });
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <UserContext.Provider value={{ user, login, logout }}>
      {children}
    </UserContext.Provider>
  );
};

// Create a consumer hook
const useUser = () => useContext(UserContext);

// Example usage
const App = () => {
  return (
    <UserProvider>
      <Content />
    </UserProvider>
  );
};

const Content = () => {
  const { user, login, logout } = useUser();

  const handleLogin = () => {
    login('exampleUser', 'password');
  };
};

```

```

const handleLogout = () => {
  logout();
};

return (
  <div>
    {user ? (
      <div>
        <p>Welcome, {user.username}!</p>
        <button onClick={handleLogout}>Logout</button>
      </div>
    ) : (
      <div>
        <p>Please login</p>
        <button onClick={handleLogin}>Login</button>
      </div>
    )}
  </div>
);
};

export default App;

```

In this example, the `UserProvider` wraps the `Content` component, providing it access to the `user` state, `login`, and `logout` functions via the `useUser` hook. This allows any component within the `Content` component to access and manipulate user information without having to pass props down manually through the component tree.

132. What is the purpose of callback function as an argument of `setState()` ?

- If we want to execute some logic once state is updated and component is rerendered then we can add it in callback function.

133. **Practical question:** create a custom hook for increment/decrement counter ?

Consider the increment/decrement custom hook:

```
const useCounter = () => {  
  const [counter, setCounter] = useState(0);  
  
  return {  
    counter, // counter value  
    increment: () => setCounter(counter + 1), // function 1  
    decrement: () => setCounter(counter - 1) // function 2  
  };  
};
```

and then in your component you can use it as follows:

```
const Component = () => {  
  const { counter, increment, decrement } = useCounter();  
  
  return (  
    <div>  
      <span onClick={decrement}>-</span>  
      <span style={{ padding: "10px" }}>{counter}</span>  
      <span onClick={increment}>+</span>  
    </div>  
  );  
}
```

134. Which lifecycle hooks in class component are replaced with useEffect in functional components ?

1. **componentDidMount()**: equivalent to useEffect with empty array.

```
useEffect(()=>{  
  console.log("Called on initial mount only once")  
}, [])
```

2. **componentDidUpdate()**: equivalent to useEffect with array of dependencies

```
useEffect(()=>{  
  console.log("Called on every dependency update")  
}, dependencies)
```

```
}, [props.isFeature, props.content])
```

This will be called whenever dependency value changes (here Eg: isFeature or content).

3. **componentDidUnmount()**: equivalent to useEffect with return statement.

```
useEffect(()=>{  
  return ()=>{  
    console.log("Any cleanup activities/unsubscribing  
  }  
})
```

135. What is Strict mode in react ?

It identifies the commonly occurred bugs in the development time itself.

- Checks if there are any deprecated apis usage.
- Checks for deprecated lifecycle methods.
- Checks if Duplicate keys in list.
- Warns about Possible memory leaks. etc.

```
// Enabling strict mode for entire App.
```

```
import { StrictMode } from 'react';  
import { createRoot } from 'react-dom/client';  
  
const root = createRoot(document.getElementById('root'));  
root.render(  
  <StrictMode>  
    <App />  
  </StrictMode>  
);
```

```

    </StrictMode>
  );

  =====

  // Any part of your app

  =====

import { StrictMode } from 'react';

function App() {
  return (
    <>
      <Header />
      <StrictMode>
        <main>
          <Sidebar />
          <Content />
        </main>
      </StrictMode>
      <Footer />
    </>
  );
}

```

Good Ref: <https://dev.to/codeofrelevancy/what-is-strict-mode-in-react-3p5b>

136. What are the different ways to pass data from child component to parent component in react ?

There are 4 common ways to send data from child component to parent component. They are.,

1. Callback Functions
2. Context API
3. React Hooks (useRef)

4. Redux

137. **Practical question:** How to send data from child to parent using callback functions ?

- Define a function in the parent component that takes data as an argument.
- Pass this function as a prop to the child component.
- When an event occurs in the child component (like a button click), call this function with the data to be passed to the parent.

Parent Component:

```
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const [dataFromChild, setDataFromChild] = useState('');

  const handleDataFromChild = (data) => {
    setDataFromChild(data);
  };

  return (
    <div>
      <ChildComponent onData={handleDataFromChild} />
      <p>Data from child: {dataFromChild}</p>
    </div>
  );
}

export default ParentComponent;
```

Child Component:


```
import React from 'react';

function ChildComponent({ onData }) {
  const sendDataToParent = () => {
    const data = 'Hello from child';
    onData(data);
  };

  return (
    <button onClick={sendDataToParent}>Send Data to Parent
  );
}

export default ChildComponent;
```

138. **Practical question:** How to send the data from child component to parent using useRef ?

- It is used to store the data without re-rendering the components.
- It will not trigger any event by itself whenever the data is updated.

Parent component:

```
import React from "react";
import TextEditor from "../TextEditor";

export default function Parent() {
  const valueRef = React.useRef("");

  return (
    <>
      <TextEditor valueRef={valueRef} />
      <button onClick={() => console.log(valueRef.current)} />
    </>
  );
}
```

```
);  
}
```

Child component:

```
export default function TextEditor({ valueRef }) {  
  return <textarea onChange={e => (valueRef.current = e)}  
}
```

139. How do you optimize your react application ?

- **Code Splitting:** Break down large bundles into smaller chunks to reduce initial load times
- **Lazy Loading:** Load non-essential components asynchronously to prioritize critical content.
- **Caching and Memoization:** Cache data locally or use memoization libraries to avoid redundant API calls and computations.
- **Memoization:** Memoize expensive computations and avoid unnecessary re-renders using tools like React.memo and useMemo.
- **Optimized Rendering:** Use shouldComponentUpdate, PureComponent, or React.memo to prevent unnecessary re-renders of components.
- **Virtualization:** Implement virtual lists and grids to render only the visible elements, improving rendering performance for large datasets.
- **Server-Side Rendering (SSR):** Pre-render content on the server to improve initial page load times and enhance SEO.
- **Bundle Analysis:** Identify and remove unused dependencies, optimize images, and minify code to reduce bundle size.
- **Performance Monitoring:** Continuously monitor app performance using tools like Lighthouse, Web Vitals, and browser DevTools.
- **Optimize rendering with keys:** Ensure each list item in a mapped array has a unique and stable key prop to optimize rendering performance.

Keys help React identify which items have changed, been added, or removed, minimizing unnecessary DOM updates.

- **CDN Integration:** Serve static assets and resources from Content Delivery Networks (CDNs) to reduce latency and improve reliability.

140. What are Portals in react ?

- Portals are the way to render the child components outside of the parent's dom hierarchy.
- We mainly need portals when a React parent component has a hidden value of overflow property(overflow: hidden) or z-index style, and we need a child component to openly come out of the current tree hierarchy.
- It is commonly used in Modals, tooltips, loaders, notifications toasters.
- This helps in improving the performance of application.

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyPortal extends React.Component {
  render() {
    return ReactDOM.createPortal(
      this.props.children,
      document.getElementById('portal-root')
    );
  }
}

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>My App</h1>
        <MyPortal>
```

```
        <p>This is rendered in a different part of the D
    </MyPortal>
  </div>
);
}
}

ReactDOM.render(<App />, document.getElementById('root'));
```
