232 Interview questions & Answers - Saikrishna Nangunuri

Important Note:

These are the frequent interview questions so revise this ebook at least 5 times (including examples) and i can guarantee you that you will crack your interview for sure 😎😈

Most important Interview Tip:

Always give theoritical answers as well as a practical example by sharing your screen to the interviewer, so that you can delay some time of the interview and interviewer will think you are a practical person 😉

This Pdf will contain.,

- 67 Javascript theory Interview questions
- 50 Output based javascript questions
- 27 Problems and solutions in javascript
- 52 Reactjs theory and practical questions
- 42 Scenario based Reactjs Practical questions
- 20 Scenario based Angular Practical questions

Important :

In interviews, mostly interviewers will focus on 5 areas.

1. Theory interview questions
2. Algorithm based interview questions
3. Output based interview questions
4. Scenario based interview questions

5. Machine coding round interview questions

---

1. 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.
- Typescript is a statically typed language. All checks are performed at compile time.

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

---

2. What are the different datatypes in javascript ? (Most asked)

- **Primitive datatypes:**
  - String
  - number
  - boolean
  - null
  - undefined
  - Bigint

    - symbol
- **Non-Primitive datatypes:**
    - Object
    - Array
    - Date

---

3. What is Hoisting in javascript ? (Most asked)

- 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.

👉 **Interview Tip:** Mention buzz word **temporal dead zone** in above answer so that interviewer will ask What is temporal dead zone. 😉

---

4. What are the various things hoisted in javascript ?

```
Function declarations: Fully hoisted.
var - Hoisted
Arrow functions: Not hoisted
Anonymous Function expressions: Not hoisted
let - Hoisted but not initialized. (Temporal dead zone).
const -Hoisted but not initialized. (Temporal dead zone)
class declarations - Hoisted but not initialized.
```

Ref: https://stackabuse.com/hoisting-in-javascript/

---

5. 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.

---

6. What are the differences let, var and const ? (Most asked)

- **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.(temporal dead zone)

---

7. List out some key features of ES6 ? (Most asked)

1. Arrow functions
2. Let and Const declarations.
3. Destructuring assignment
4. Default parameters
5. Template literals
6. Spread and Rest operators

7. Promises
8. Classes
9. Modules
10. Map, Set, Weakmap, Weakset

👉 **Interview Tip:** Here try to explain defination (provided in below questions) for these features so that you can kill 2-3 min of interview time 😉

---

8. 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.

👉 **Note:** Arrow functions + this combination questions will be asked here. Please explore on this combinations

---

9. 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)
```

👉 **Interview Tip:** Practice the above examples mentioned and showcase them in interviews to make interviewer think that you are a practical person. 😉

---

10. What is rest operator in javascript ?

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

This is useful when we dont 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);
```

---

11. 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
```

---

12. What are the differences between Map and Set ?

Map	Set
Map is the collection of key value pairs	Set is a collection of unique values
Map is two dimensional	Set is one dimensional
Eg: let data = new Map();	Eg: let data = new Set();

data.set(”name”,”saikrishna”);
data.set(”id”,”1”);
for(let item of data){
console.log(item)
}
 	
O/P
[”name”,”saikrishna”]
[”id”,”1”]
	data.add(1);
	data.add(”saikrishna”);
	for(let item of data){
	console.log(item)
	}
	
	O/P
	1
	Saikrishna

new Map([iterable]) – creates the map, with optional iterable (e.g. array) of [key,value] pairs for initialization. map.set(key, value) – stores the value by the key, returns the map itself map.get(key) – returns the value by the key, undefined if key doesn’t exist in map map.has(key) – returns true if the key exists, false otherwise. map.delete(key) – removes the element by the key, returns true if key existed at the moment of the call, otherwise false. map.clear() – removes everything from the map. map.size – returns the current element count.	new Set([iterable]) – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set. set.add(value) – adds a value, returns the set itself . set.delete(value) – removes the value, returns true if value existed at the moment of the call, otherwise false. set.has(value) – returns true if the value exists in the set, otherwise false. set.clear() – removes everything from the set. set.size – is the elements count.

Ref: https://javascript.info/map-set

---

13. What are modules in javascript ?

- Modules allows us to break down the large piece of code into smaller parts.

- Modules helps us to write more reusable and maintenable code.
- Modules can be imported and exported using import and export statements.

---

14. 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.

```
// Pass by value example
let num = 10;

function changeNum(value) {
   value = 20;
   console.log(value);  // Output: 20
}

changeNum(num);
console.log(num);  // Output: 10
```

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.

```
// Pass by reference example
let arr = [1, 2, 3];
```

```
function addToArr(value) {
    value.push(4);
    console.log(value);  // Output: [1, 2, 3, 4]
}

addToArr(arr);
console.log(arr);  // Output: [1, 2, 3, 4]
```

15. What is the difference between map and filter ? (Frequently asked)

- 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.

16. What is the difference between map() and forEach() (Frequently asked)

- 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.

17. 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

---

18. 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
```

19. 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/

---

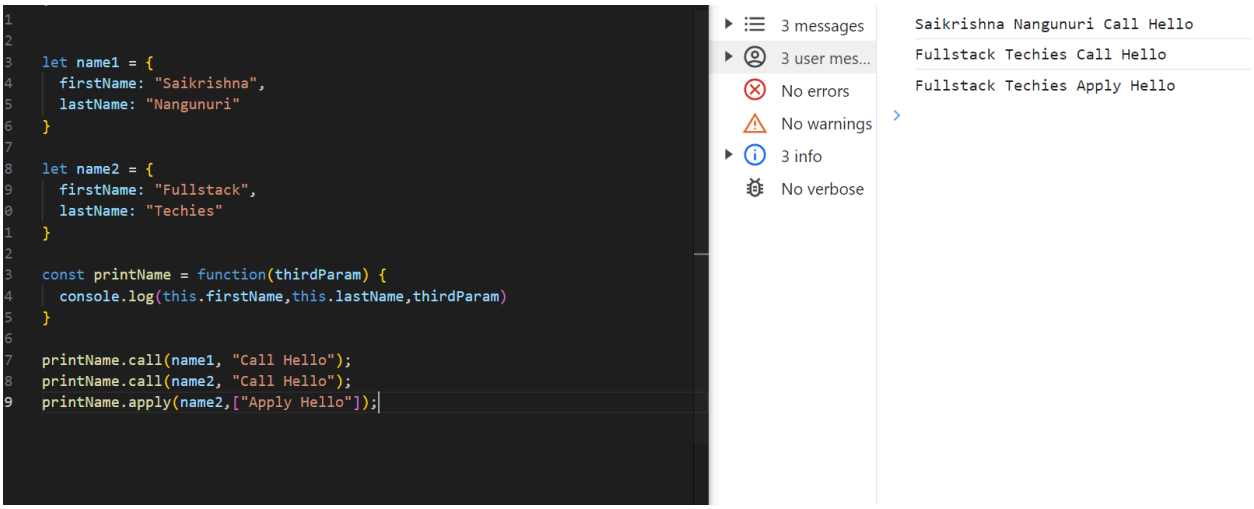
20. What are the differences between call(), apply() and bind() ? (Very Frequently asked)

👉 **Interview Tip:** Here give below example and then execute the examples and explain differences.

Interviewer will have positive impression on you.

- Call method will invokes the function immediately with the given this value and allow us to pass the arguments one by one with comma separator.
- Apply method will invokes 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.

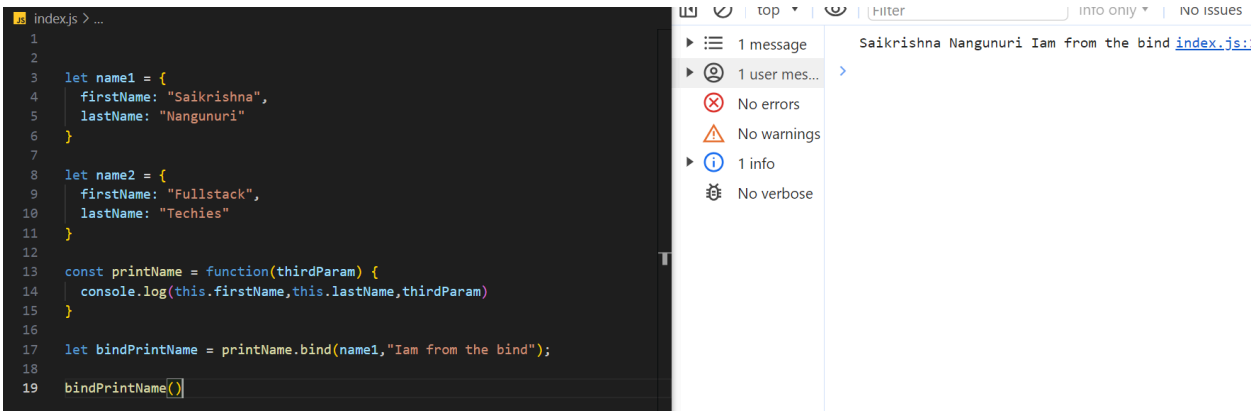
Brief examples:



The only difference between call and apply is that syntax of how we pass the arguments.

bind: This gives us a copy which can be invoked or executed later rather than directly invoking it whereever we are writing this line of code.

We can use bind() for events like onClick where you dont know when they will be fired but you know the desired context.



21. Different ways to create object in javascript ? (Most asked)

👉 **Interview Tip:** Give the examples and then explain it.

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);
```

---

22. 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. **(Practice example for this - this might be asked)**

```
let data = {
 name: "Sai",
 lang: "English"
};

Object.keys(data)  // ["name","lang"]
Object.values(data) // ["Sai","english"]
Object.entries(data) // [["name","Sai"],["lang","English"]]
```

---

23. 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
}
```

24. What is a polyfill in javascript ?

👉 **Interview Tip:** If polyfill is asked, then 99% they will ask you to write a polyfill. So practice atleast 2-3 polyfills (map,foreach compulsary)

- 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:**

```
let data = ["sai","krishna"];

data.forEach((item,i)=>{
  console.log(item,i)
})

Array.prototype.forEach((callback)=>{
  for(let i=0;i<=this.length-1;i++){
    callback(this[i],i)
  }
})
```

- **polyfill for map:**

```
let data = [1,2,3,4];

let output = data.map((item, ix)=>{
  return `${item}_hello`
})

Array.prototype.map=((callback)=>{
  let temp = [];
  for(let i=0;i<=this.length-1;i++){
```

```
    temp.push(callback(this[i]))
  }
  return temp;
})

console.log(output)
```

ref: https://dev.to/umerjaved178/polyfills-for-foreach-map-filter-reduce-in-javascript-1h13

---

25. 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)
```

---

26. 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()); // {value:1,done:false}
console.log(generator.next()); // {value:2,done:false}
console.log(generator.next()); // {value:3,done:false}
console.log(generator.next()); // {value: 4,done:true}
```

---

27. 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")
})()
```

---

28. What is CORS ? (Most asked)

👉 **Interview Tip:** This defination is more than enough so prepare this below answer well.

- 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.

https://dev.to/lydiahallie/cs-visualized-cors-5b8h

---

29. What are the difference between typescript and javascript ?

👉 **Interview Tip:** If your interview contains typescript then this is a 99% dam sure question. Prepare these differences blindly.

- Typescript is the superset of javascript and has all the object oriented features.
- Javascript is a dynamically typed language whereas typescript is a statically typed language.
- Typescript is better suited for large scale applications where as javascript is suited for small scale applications.
- 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.
- Functions have optional parameters in typescript whereas in javascript does not have it.
- Typescript takes longer time to compile code.

---

30. What is authentication vs authorization ? (Most asked)

- **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.

👉 **Interview Tip:** For this question, learn **jwt token mechanism** and tell that you have implemented this in your project. This helps a lot.This kills atleast 3-4 min of interview time 😉

https://www.youtube.com/watch?v=7Q17ubqLfaM

31. 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.

---

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

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

---

33. Slice vs Splice in javascript ? (Most helpful in problem solving)

- **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); //  [1,2,8,9,6,10]
console.log(newArr); // [3,4,5,0]
```

---

34. 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
```

👉 **Interview Tip:** Most asked in output based and problem solving so learn syntax more. Practice some examples.

---

35. 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);
```

👉 **Interview Tip:** Most asked in output based and problem solving so learn syntax more. Practice some examples.

36. What are Promises in javascript ?

👉 **Interview Tip:** When this is asked cover all below points so that he will not ask any other question on promises 😈.

- 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 succeded.
  - **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 wents 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)
})
```

---

37. 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

👉 **Interview Tip:** practice some examples on this concepts. This is a practical question. You can expect some scenario based questions from interviewer on this concept so prepare well from above link

---

38. What is a callstack in javascript ? (Very rare)

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

---

39. What is a closure ? (Most asked in all the interviews 99% chance)

- **Defination:** A function along with its outer environment together forms a closure
- 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();
```

---

40. 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.
- **Usecases:**
  - 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
```

---

41. 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

---

42. What is the main difference between Local Storage and Session storage ? (Most asked)

- Local storage and session storage are two ways of storing data using key value pairs in web browsers.
- LocalStorage is the same as SessionStorage but it persists the data even when the browser is closed and reopened and on reload(i.e it has no expiration time) whereas in sessionStorage data gets cleared when the page session ends.
- Both provides same methods,
  - `setItem(key, value)` – store key/value pair.
  - `getItem(key)` – get the value by key.
  - `removeItem(key)` – remove the key with its value.
  - `clear()` – delete everything.
  - `key(index)` – get the key on a given position.
  - `length` – the number of stored items.

Ref: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API#concepts_and_usage

---

43. 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.

---

44. Possible followup questions on local storage :

👉 **Interview Tip: When asked about local storage 100% compulsary they will confuse you with practical questions.**😖😖

All the scenarios are covered here so 100% sure nothing more than this will be asked. 😊😊

1. I created a localstorage and closed the browser and repoened it. Will local storage data persists ?

Yes local storage data persists even when i close and reopen the browser

2. I want to access Local storage data in another tab of same browser is it possible ?

Yes we can access local storage data in another tab as well.

3. I reloaded the page after creating local storage. Will it persists ?

Yes local storage data persists on page reload.

4. If i open multiple tabs with same url how local storage behaves ?

I can access localstorage data in multiple tabs if its same url

5. If i open multiple windows with same url how local storage behaves

I can access local storage data even for different windows with same url.

6. When local storage data will be removed ?

It stays indefnitely until its deleted manually by the user.

7. Is Local storage synchronous or asynchronous ?

Localstorage is synchronous. If i perform operations on local storage, It blocks the execution of other javascript code until the current operation is completed.

8. I want asynchronous operations and large data sets to be stored then what you will suggest ?👉 Remember this question)

I can go with Indexeddb where asynchronous operations are supported and we can work with large data sets.

9. What is the max storage limit of local storage ?

We can store max of 5mb.

10. I will try to store lets say an image of size more than 5mb in localstorage then what happens ?

It will throw QuotaExceededException if it exceeds the limit.

11. What If I turn off my laptop and reopen the browser, will local storage data persists ?

Yes localstorage data still persists even if i shutdown my laptop and reopen the browser

---

45. Possible followup questions on session storage :

👉 Interview Tip: When asked about session storage 100% compulsary they will confuse you with practical questions.😖😖

All the scenarios are covered here so 100% sure nothing more than this will be asked. 😊😊

1. I created a sessionstorage and closed the browser and repoened it and restored tab. Will session storage data persists ?
No session storage data does not persists on browser close & reopen.

1. I created a sessionstorage and closed the browser and repoened it and restored tab. Will session storage data persists ?
No session storage data does not persists on browser close & reopen.

2. Can i access session storage data in another tab of same browser ?
No we cannot access session storage data of one tab in another tab.

3. I reloaded the page after creating session storage. Will it persists ?
Yes session storage data persists on page reload.

4. If i open multiple tabs with same url how session storage behaves ?
We cannot access session storage data in multiple tabs even if its same url

5. If i open multiple windows with same url how session storage behaves ?
We cannot access session storage data in multiple windows even if its same url

6. When session storage data will be removed ?
once tab closes or session ends session storage data will be removed.

7. Is session storage synchronous or asynchronous ?
Session storage is synchronous. If i perform operations on session storage, It blocks the execution of other javascript code until the current operation is completed.

8. I want asynchronous operations and large data sets to be stored then what you will suggest ? 👉 Remember this question)
I can go with Indexeddb where asynchronous operations are supported and we can work with large data sets.

9. What is the max storage limit of session storage ?
We can store max of 5mb.

10. I will try to store lets say an image of size more than 5mb in localstorage then what happens ?
It will throw QuotaExceededException if it exceeds the limit.

46. What are cookies ?

- Cookies are used to store information about the user in the webpages.
- Cookies are stored as key value pairs and hold 4kb of data.
- When user logins to the application, server uses the set-cookie http header in the response to set a cookie with a unique session identifier. Next time when user makes the api requests, cookie will be sent in the http header by using which server will identify who the user is.

Eg:

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";

Ref: https://www.w3schools.com/js/js_cookies.asp

---

47. Interview questions on this keyword.

👉 **Interview Tip:Asked frequently so all scenarios are covered here. Learn carefully. Thank me later** 😇

1. **What is this in javascript** ?

   this refers to the object that is currently executing the code.

2. **What is the value of this in global scope** ?

   Its a global object. its value can be global or window. It depends on where you are running javascript code.(like browser or node environment etc)

3. **I will use non strict mode. Now if i use this inside a function, what will be the output** ?

   Its a global object.

   In non strict mode, when ever this keyword value is null or undefined, javascript will replace it's value with global object.(Due to this substitution)

4. In strict mode, What will the value of this inside a function ?

```
function x(){
console.log(this)
}
```

In strict mode, the value of this will be undefined.

5.Now In strict mode, If i call above function using window.x() then what is the result of this ?

```
function x(){
    console.log(this)
}
window.x()
```

It will log window object.

6.What will be the value of this inside object method below ?

```
let obj = {
    x:"Hello",
    y: function(){
    console.log(this.x)
    }
}

obj.y()
```

It will print Hello. Because, When ever we are inside the method, the value of this keyword is the object where this method is present.

7. What will be the this value if it's logged inside arrow function ?

```
let obj = {
    x:"Hello",
    y: ()=>{
    console.log(this)
```

```
    }
}

obj.y()
```

It will print window object.Because, Arrow function does not have their own this binding. they take the this value of their lexical environment where they are enclosed.

8.What will be this value if i use in button element

```
<button onclick="alert(this)">click</button>
```

It will display [object HTMLElement]

---

48. 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;
})
```

---

49. What is eval() ? (Very rare)

- 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
```

---

50. What is the difference between Shallow copy and deep copy ? (Most asked)

👉 **Interview Tip: Give this example and explain that's it.**

- **Shallow copy:**
  - A shallow copy creates a new object or array and copies the references of the original elements
  - https://www.linkedin.com/posts/saikrishnanangunuri_javascript-javascriptdeveloper-reactjs-activity-7211747675635900416-h9IB?utm_source=share&utm_medium=member_desktop
    (Give above post example)

```
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]]

using lodash:
let array = _.cloneDeep(originalArray)
```

---

51. 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.

---

52. 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.

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Event Bubbling</title>
</head>
<body>
  <div id="div1" style="padding: 50px;
    border: 1px solid black;">
    Div 1
    <div id="div2" style="padding: 50px;
    border: 1px solid red;">
      Div 2
      <button id="button1">Click Me</button>
    </div>
  </div>

  <script>
    document.getElementById('div1').addEventListener(
      'click', function() {
      console.log('Div 1 clicked');
    });

    document.getElementById('div2').addEventListener(
      'click', function() {
      console.log('Div 2 clicked');
    });

    document.getElementById('button1').addEventListener(
      'click', function(event) {
      console.log('Button clicked');
      // To stop the event from bubbling up
      event.stopPropagation();
    });
  </script>
</body>
</html>
```

53. 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.
- You can enable event capturing by passing `true` as the third argument to `addEventListener`.

```
<!DOCTYPE html>
<html>
<head>
  <title>Event Capturing</title>
</head>
<body>
  <div id="div1" style="padding: 50px;
   border: 1px solid black;">
    Div 1
    <div id="div2" style="padding: 50px;
    border: 1px solid red;">
      Div 2
      <button id="button1">Click Me</button>
    </div>
  </div>

  <script>
    document.getElementById('div1').addEventListener(
    'click', function() {
      console.log('Div 1 clicked');
    }, true);

    document.getElementById('div2').addEventListener(
    'click', function() {
```

```
    console.log('Div 2 clicked');
  }, true);

  document.getElementById('button1').addEventListener(
  'click', function(event) {
    console.log('Button clicked');
    // To stop the event from propagating further
    // event.stopPropagation();
  }, true);
 </script>
</body>
</html>
```

54. What are the various array methods in javascript ?

- **toString():** returns array as acomma separated string.
- **join():** same as toString but we can specify the separator. Eg: [”a”,”b”,”c”].join(”=”) // a=b=c
- **at():** returns element at specific index. Eg: [”ayan”,”saikrishna”].at(1)
- **pop():** removes the last element from an array.
- **push():** adds new element to array at the end.
- **shift():** removes the first element at the beginning and shifts all the elements to lower index.
- **unshift():** adds new element at beginning and moves other elements one index further.
- **concat():** creates new array by concatenating existing arrays.
- **copyWithin():** copies array element to another position Eg: [”a”,”b”,”c”].copyWithin(2,0)
- **flat():** creates a new array with all the sub array elements

- **fill():** will fill all the elements of an array from a start index ti an end index with a stati value.
  - arr.fill(value,start,end) ⇒ [1,23,46,58].fill(88,1,3) ⇒ [1,88,88,58]

Ref: https://www.w3schools.com/js/js_array_methods.asp

55. What are the differences between some and every in javascript ?

- **some():** will check if atleast one of the element in the array satisfies the specified condition. It returns true if any element passes the condition. else returns false.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const someGreaterThan10 = numbers.some(
number ⇒ number > 10
);
console.log(someGreaterThan10); // true
```

- **every():** will check if all the elements of the array satisfies the specified condition. it return true if all the elements satisfies the condition, else returns false.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const everyGreaterThan10 = numbers.every(
number ⇒ number > 10
);
console.log(everyGreaterThan10); // false
```

56. What are the different types of errors in javascript ?

- **Syntax errors:** These are occured when code is not written according to the javascript syntax rules.

```
Eg: let x=;
```

- **Reference errors:** These occurs when we refer any variables or methods that does not exists.

```
let val = y;//y does not exist

function x(){
 data() // This method is not exists.
}
```

- **Type errors:** These errors occurs when operation is performed on the value of wrong datatype

```
try {
  null.f();
} catch (e) {
  console.error(e);
  // TypeError: Cannot read property 'f' of null
}
```

- **Range errors:** These errors occurs when the value is not present in allowed range.

```
try {
  new Array(-1);
} catch (e) {
  console.error(e);
  // RangeError: Invalid array length
}
```

```
Eg2:
 if (num < 30) throw new RangeError("Wrong number");
              ^
 RangeError: Wrong number
```

- **Eval or Evaluation errors:** These errors occurs in eval functions. Not commonly used in modern browsers.

```
try {
  eval('eval("invalid code")');
} catch (e) {
  console.error(e);
  // EvalError (in some older JavaScript versions)
}
```

- **URI errors:** These erros occurs when wrong characters used in URI functions

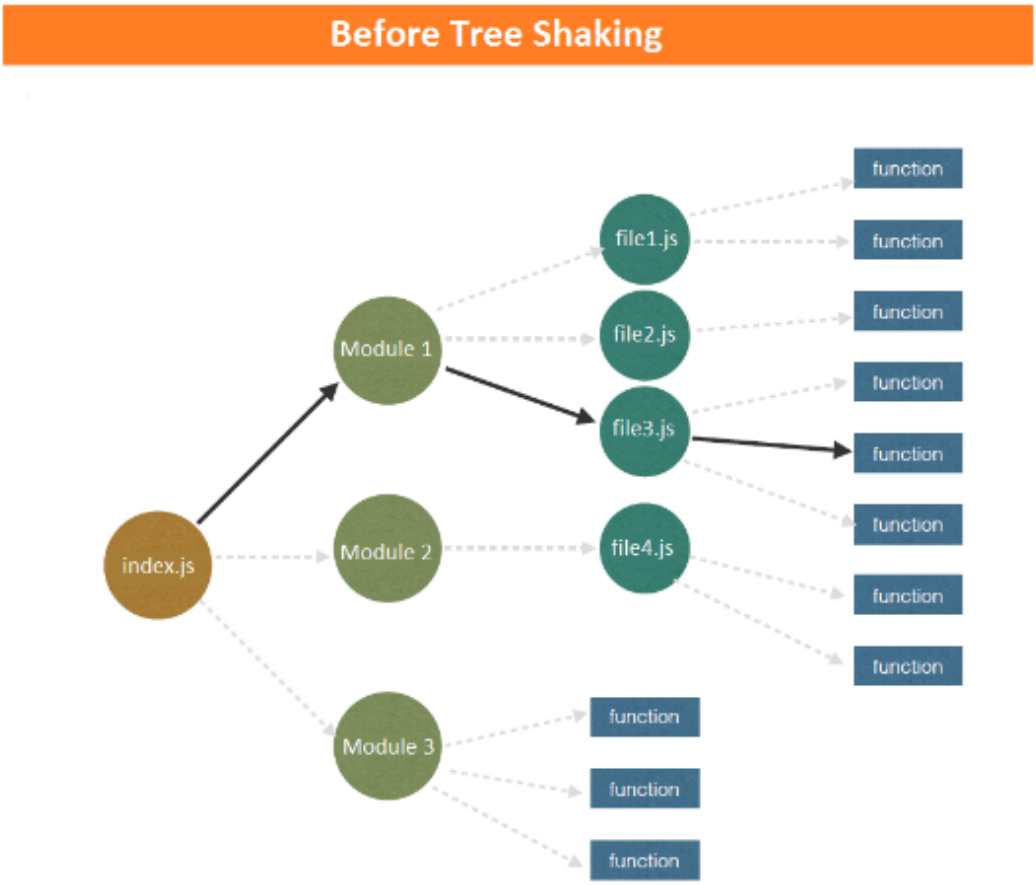
```
console.log(decodeURI("https://www.learndepth")) //works fine
console.log(decodeURI("%sdfk")); //throws error
decodeURIComponent('%');
```

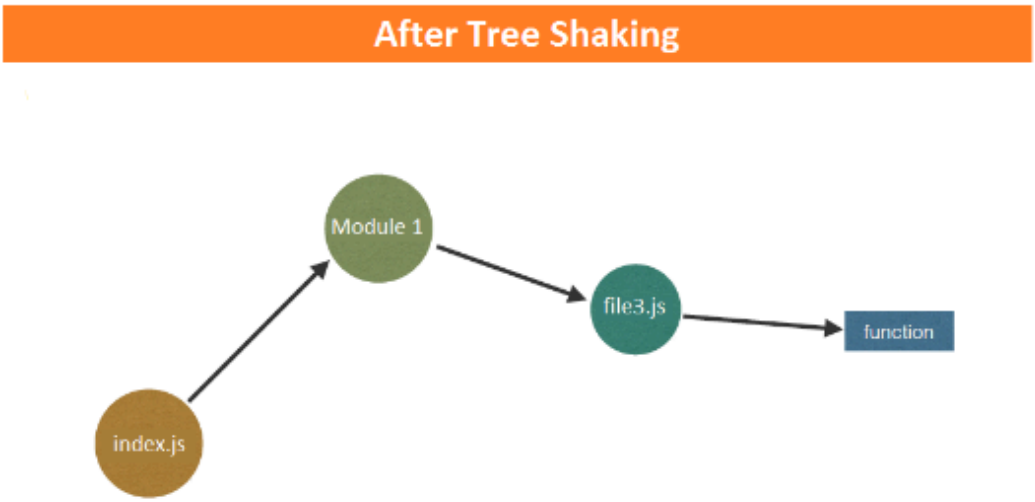
---

57. 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 eleminating unused modules and functions.

- Faster load time.
- Performance will be improved.
- Cleaner and maintainable codebases.





58. What is prototype inheritance in javascript ?

- **Prototype inheritance** in javascript is the linking of prototypes of a parent object to a child object so that we can share and utilize the properties of a **parent class** using a **child class**.
- The ES5 introduced two differnd methods such as Object.create() and Object.getPrototypeOf().

```
let package = {
  version: "2.0",
};
let application = Object.create(package, {
  name: { value: "game" },
}); // inherited from package
console.log(application);
console.log(Object.getPrototypeOf(application));
```

Ref: https://www.scaler.com/topics/javascript/prototype-inheritance-in-javascript/

59. What are the differences between fetch and axios ?

- fetch is inbuilt webapi method present in most of the modern browsers where as axios is a third party library is built on top of **XMLHttpRequest** object.
- axios performs **automatic parsing** of response data where as in fetch we need to manually parse the response data(eg: response.json()).
- axios supports **interceptors** by using which we can modify the request or response before they are sent/received from the server.

- axios prevents **csrf (cross site request forgery)**attacks.

ref: https://apidog.com/blog/axios-vs-fetch/

```
fetch:

fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok'
      + response.statusText);
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(`There has been a problem with
    your fetch operation:`, error);
  });

axios:

axios.get(url)
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error(`There has been a problem
     with your axios operation:`, error);
  });
```

60. What are DRY, KISS, YAGNI, SOLID Principles ? (rarely asked)

- **DRY:** Do not repeat yourself.
  - Avoid duplicates. This make software more maintainable and less error-prone.
- **KISS:** Keep it simple stupid.
  - Keep the software design and implementation as simple as possible. This make 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:**
  - **S - Single responsibility:** means each class should have one job or responsibility.
  - **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.
  - **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.

---

61. What is Babel ?

- Babel is a javascript compiler which is used to convert the modern javascript code into the version that is understandable by all the modern

and older browsers.

---

62. What are the main advantages of arrow functions ?

- Arrow functions allow us to write shorter and concise syntax and reduces the size of code.
- It increases the readability of the code.

```
const square = x => x*x;
```

- Arrow functions will solve the common pain points of this binding in traditional funtional expressions. Arrow functions does not have their own this. It will take the this value from the parent's scope (i.e., code that contains the arrow function). For example, look at the `setTimeout` function below.

```
// ES5
var obj = {
 id: 42,
 counter: function counter() {
  setTimeout(function() {
   console.log(this.id);
  }.bind(this), 1000);
 }
};

obj.counter(); // 42

/* In the ES5 example, .bind(this) is required to
 help pass the this context into the function.
 Otherwise, by default this would be undefined.*/
```

```
// ES6
var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(() => {
      console.log(this.id);
    }, 1000);
  }
};
obj.counter() // 42
```

ES6 arrow functions can't be bound to a `this` keyword, so it will lexically go up a scope, and use the value of `this` in the scope in which it was defined.

Helpful for understanding this binding for arrow functions:

In this link check No binding of this section : https://www.freecodecamp.org/news/when-and-why-you-should-use-es6-arrow-functions-and-when-you-shouldnt-3d851d7f0b26/

https://stackoverflow.com/questions/22939130/when-should-i-use-arrow-functions-in-ecmascript-6

https://tc39wiki.calculist.org/es6/arrow-functions/

---

63. What is clearInterval in javascript ?

- This method takes the interval ID returned by `setInterval` as an argument and stops the corresponding interval from executing further.

```
let count = 0;
const intervalId = setInterval(() => {
  console.log(count++);
```

```
  if (count > 5) {
    clearInterval(intervalId);
    // Stop the interval after 5 iterations
  }
}, 1000); // Execute every 1 second
```

This code will print the numbers 0 to 5 with a one-second delay between each number. After printing 5, the `clearInterval` method will be called, and the interval will stop.

---

64. What are webworkers and give an example ?

- Webworkers allows us to run the jsvsacript code in the background with out blocking the main thread of the web application.
- This is mainly useful for performing computationally intensive tasks which will make the webapplication unresponsive.
    - Eg: Larger attachment upload of 2gb.

Create a Web Worker Script :

```
// worker.js

// This code runs in the worker context
self.onmessage = function(event) {
   //event.data contain data sent from main thread
   let result = 0;
   for (let i = 0; i < event.data; i++) {
      result += i;
   }
   // Send the result back to the main thread
```

```
    self.postMessage(result);
};
```

Create the Main Script :

```
// main.js

// Check if the browser supports Web Workers
if (window.Worker) {
    // Create a new Web Worker
    const myWorker = new Worker('worker.js');

    // Send data to the worker
    myWorker.postMessage(1000000); // Example data

    // Listen for messages from the worker
    myWorker.onmessage = function(event) {
     //event.data contains data sent from worker
     console.log('Result:', event.data);
    };

    // Handle errors from the worker
    myWorker.onerror = function(event) {
        console.error('Error:', event.message);
    };
} else {
    console.log('WebWorker not supported in this browser');
}
```

65. What are the differences between ^ and ~ symbol in package.json ?

Caret (^)

The caret (`^`) allows updates to the most recent minor version (1.x.x), but it will not allow changes that could potentially introduce breaking changes (major version updates).

- Example: `"^1.2.3"`
  - This means that the package manager will install any version from `1.2.3` to less than `2.0.0`.
  - It will install newer patch versions (e.g., `1.2.4`, `1.2.5`) and minor versions (e.g., `1.3.0`, `1.4.0`), but not major versions (e.g., `2.0.0`).

Tilde (~)

The tilde (`~`) allows updates to the most recent patch version (x.x.1), but it will not allow updates to the minor or major versions.

- Example: `"~1.2.3"`
  - This means that the package manager will install any version from `1.2.3` to less than `1.3.0`.
  - It will install newer patch versions (e.g., `1.2.4`, `1.2.5`), but not new minor versions (e.g., `1.3.0`).

---

66. Plain javascript basic questions:

- Write code to change style of div ?

```
let element = document.getElementById("newele");
element.style.color = "red";
element.style.backgroundColor = "blue";
element.style.fontSize = "20px";

// or

let elem = document.getElementById("newele");
elem.setAttribute("style",
"color:white;background-color: brown");
```

- Write code to add class to div ?

```
// Select the div element
var element = document.getElementById("myDiv");

// Add a class
element.classList.add("newClass");
```

---

67. Is javascript synchronous or asynchronous and single threaded or multithreaded ?

- Javascript is a **synchronous single threaded language**. This means that it executes line by line in order and each line must finish executing before the next line starts.
- However, javascript has can handle asynchronous operations using mechanisms like callbacks, promises and async/await. These mechanisms allows javascript to perform tasks such as network requests, file reading, setTimeout/setInterval without blocking the main thread.
- These mechanisms allow JavaScript to delegate tasks to the browser and then continue executing other code while waiting for those tasks to complete. This asynchronous behavior gives the illusion of concurrency, even though JavaScript itself remains single-threaded.

---

36 Output based questions:

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

- "57"

Explaination:

- In javascript, the addition associativity is from left to right. Due to this initially 3+2 will be evaluated and it will become 5. Now expression becomes 5+"7".
- In javascript when ever we try to perform addition between a numeric value and a string, javascript will perform type coercion and convert numeric value in to a string.
- Now expression becomes "5"+ "7" this performs string concatenation and results in "57".

---

2. 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`.

---

3. Guess the ouput ?

```
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`.

---

4. Guess the output ?

```
a) 2+2 = ?
b) "2"+"2" = ?
c) 2+2-2 = ?
d) "2"+"2"-"2" = ? (tricky remember this)
e) 4+"2"+2+4+"25"+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 a number, then subtracted by the number 2)

e) 4+"2"+2+4+"25"+2+2
console.log(4+"2"+2+4+"25"+2+2);
// "42242522"
```

---

5. What is the output of below logic ?

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

console.log(a)
```

- **Output:**
  - “jscafe”
  - 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 = “cscafe” // outputs “cscafe”

---

6. 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.

7. 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.

8. This code prints 6 everytime. How to print 1,2,3,4,5,6 ? (Most asked)

```
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) {
    // Set a timeout to log value of x after x seconds
    setTimeout(() ⇒ {
      console.log(x);
    }, x * 1000);
  };

  // Loop from 1 to 5
  for (var i = 1; i <= 5; i++) {
    // Call the closure function with current value of i
    closur(i);
  }
}

// Call the outer function x
x();
```

The function we have written defines an inner function `closur` which is supposed to log the value of `x` after `x` seconds. The outer function `x` calls this inner function for values from 1 to 5.

The code will log the values 1 to 5 after 1 to 5 seconds respectively. Here's an explanation of how it works:

1. The outer function `x` is called.
2. Inside `x`, a loop runs from `i=1` to `i=5`.
3. For each iteration of the loop, the inner function `closur` is called with the current value of `i`.
4. Inside `closur`, a `setTimeout` is set to log the value of `x` after `x` seconds.

Each call to `closur(i)` creates a new closure that captures the current value of `i` and sets a timeout to log that value after `i` seconds.

When you run this code, the output will be:

```
1 (after 1 second)
2 (after 2 seconds)
3 (after 3 seconds)
4 (after 4 seconds)
5 (after 5 seconds)
```

This happens because each iteration of the loop calls `closur` with a different value of `i`, and each `setTimeout` inside `closur` is set to log that value after `i` seconds.

---

9. 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.

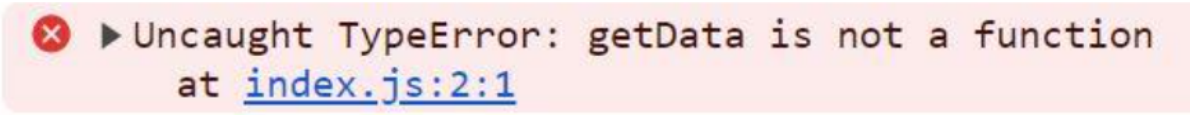
---

10. What's the output of below logic ?

```
getData1()
getData();
```

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

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

 /* Here declaring getData with let causes
 reference error.ie.,"ReferenceError: Cannot
 access 'getData' before initialization As we
 are declaring with var it throws type error
 as show below */
```

Output:



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 "getData11"
getData();  // This will log "Hello"
```

---

11. 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.

---

12. 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)
```

---

13. 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.

---

14. 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

---

15. 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.

---

16. 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`.

---

17. 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`.

---

18. 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`.

---

19. 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.

---

20. 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

Explaination: **getName1()** function works fine because it's being called from ***personObj***, so it has access to *this.name* property. But when while calling **getnName2** which is defined under *Object.prototype* doesn't have any

proprty 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();
```

---

21. 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.

---

22. 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("22222222222")
```

```
}).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
```

---

23. 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 will be hoisted */

 if(!a) {
   a = 10;
 }
 console.log(a);
 // 10 precendence will be given to local scoped variable.
}


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

---

24. 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.

---

25. Test your basics on comparision ?

```
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.`

---

26. Guess the output ?

```
var x = 23;
(function(){
  var x = 43;

  (function random(){
    x++;
    console.log(x);
    var x = 21;
  })();
})();
```

Solution:

The provided code snippet demonstrates the behavior of variable hoisting and function scope in JavaScript. Let's analyze the code step-by-step to understand the output:

```
var x = 23;
(function(){
  var x = 43;

  (function random(){
    x++;
```

```
    console.log(x);
    var x = 21;
  })();

})();
```

Breakdown

1. **Global Scope:**

   ```
   var x = 23;
   ```

   - A global variable `x` is declared and initialized with the value `23`.

2. **First IIFE (Immediately Invoked Function Expression):**

   ```
   (function(){
     var x = 43;
     // ...
   })();
   ```

   - A new function scope is created. Inside this function, a local variable `x` is declared and initialized with the value `43`. This `x` shadows the global `x`.

3. **Second IIFE (Nested function, named `random`):**

   ```
   (function random(){
     x++;
     console.log(x);
     var x = 21;
   })();
   ```

   - Another function scope is created inside the first IIFE. The function `random` is invoked immediately.

4. **Inside the `random` function:**

```
x++;
console.log(x);
var x = 21;
```

- Here, variable hoisting comes into play. The declaration `var x = 21;` is hoisted to the top of the function `random`, but not its initialization. Thus, the code is interpreted as:

  ```
  var x;   // x is hoisted, but not initialized
  x++;
  console.log(x);
  x = 21;
  ```

- Initially, `x` is `undefined` because the hoisted declaration of `x` does not include its initialization.
- `x++` attempts to increment `x` when it is still `undefined`. In JavaScript, `undefined++` results in `NaN` (Not a Number).
- Therefore, `console.log(x);` outputs `NaN`.
- After the `console.log` statement, `x` is assigned the value `21`, but this assignment happens after the `console.log` and thus does not affect the output.

Summary

When `random` function is executed, the following sequence occurs:

1. `var x;` (hoisting, `x` is `undefined` at this point)
2. `x++;` (`undefined++` results in `NaN`)
3. `console.log(x);` outputs `NaN`
4. `x = 21;` (assigns `21` to `x`, but this is after the `console.log`)

Output

Thus, the output of the code is:

```
NaN
```

---

27. Answer below queries on typeOf operator in javascript ?

```
typeof [1,2,3,4]     // Returns object
typeof null          // Returns object
typeof NaN           // Returns number
typeof 1234n         // Returns bigint
typeof 3.14          // Returns number
typeof Symbol()      // Returns symbol

typeof "John"        // Returns string
typeof 33            // Returns number
typeof true          // Returns boolean
typeof undefined     // Returns undefined
```

---

28. 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:

```
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.

---

29. Can you guess the output for below code ?

```
console.log(typeof typeof 1)
```

In the above code, we are using typeof operator. So what is this typeof ?

- typeof is an operator in javascript which will return the data type of the operand which is passed to it
- In the above example, Initially typeof 1 will be executed which will return “number”.
- Now the expression will become typeof “number” which will result in “string”.

---

30. Guess the output of below code ?

```
x++;
console.log(x);
var x = 21;
```

Explaination:

- In this example, First we need to understand how variables declared with var are hoisted ?
  - In javascript variables declared with var are hoisted on to the top of their scope and initialized with the value of undefined. Due to this, in the first line, value of x is undefined.
- In the first line we are using postfix operator. so javascript will perform type cohercion and tries to convert this undefined into a number.
- As a result, value of x will become Nan and in the 2nd line x value ie., Nan is logged to the console.

31. Guess the output ?

```
const x = [1];
const y = [2];
console.log(x+y);
```

Explanation:

1. **Variable Declaration:**
   - `const x = [1];` declares a constant variable `x` and assigns it an array containing a single element: the number 1.
   - `const y = [2];` declares another constant variable `y` and assigns it an array containing the number 2.
2. **Array Concatenation (Attempt):**
   - `x+y` attempts to concatenate the arrays `x` and `y`.
   - **However, the `+` operator performs string concatenation when used with arrays in JavaScript.**
3. **Implicit Type Conversion:**
   - JavaScript implicitly converts the arrays `x` and `y` to strings.
   - The array `[1]` is converted to the string `"1"`.
   - The array `[2]` is converted to the string `"2"`.
4. **String Concatenation:**
   - The `+` operator then concatenates the strings `"1"` and `"2"`, resulting in the string `"12"`.
5. **Console Output:**
   - `console.log(x+y);` prints the resulting string `"12"` to the console.

Key Points:

- The `+` operator does not perform array concatenation in JavaScript.
- When used with arrays, the `+` operator implicitly converts the arrays to strings and then concatenates them.

---

32. Guess the output ?

```
const data = {
   name: "sai",
   name: "krishna"
}

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

Output: “krishna”.

Explanation:

In JavaScript, if an object has multiple properties with the same name, the last one defined in the object will overwrite the previous ones. This behavior occurs because object properties in JavaScript are treated as key-value pairs, where keys must be unique.

---

33. What is the output of below code ?

```
let x = 10+2*3
console.log(x);
```

Output: 16

Video Explaination: https://www.instagram.com/p/DHqg2pXCXmw/

Explaination:

- The expression `10 + 2 * 3` is evaluated according to operator precedence rules in JavaScript. Multiplication (`*`) has a higher precedence than addition (`+`), so the multiplication is performed first:
  - First, compute `2 * 3` which equals `6`.
  - Then, add `10` to `6`, resulting in `16`.

---

34. Guess the output ?

```
const x = [1,2,3];
const y = [1,3,4];
console.log(x+y);
```

Output: “1,2,31,3,4”;

Video Explaination: https://www.instagram.com/p/DHoMoC0ii_x/

Explaination:

Here we are defining two arrays:

- `x` is `[1, 2, 3]`
- `y` is `[1, 3, 4]`

When you use the `+` operator with arrays, JavaScript converts each array to a string. Under the hood, this conversion happens by calling the `toString()` method on the arrays. For example:

- `x.toString()` becomes `"1,2,3"`
- `y.toString()` becomes `"1,3,4"`

Then, the `+` operator concatenates these two strings together. So the result of `x + y` is the string:

Finally, `console.log` prints "1,2,31,3,4" this string to the console.

35. Guess the output ?

```
console.log(+true);
console.log(!"sai");
```

Output: 1,false

Video Explaination:https://www.instagram.com/p/DHlWUMaChl9/

Explaination:

`console.log(+true);`

- The unary plus operator (`+`) converts its operand into a number.
- `true` is converted to `1`.
- **Result:** This prints `1` to the console.

`console.log(!"sai");`

- The logical NOT operator (`!`) converts its operand to a boolean and then negates it.
- The string `"sai"` is non-empty, which makes it a truthy value.
- Negating a truthy value results in `false`.
- **Result:** This prints `false` to the console.

In summary:

- `+true` **becomes** `1`.
- `!"sai"` **becomes** `false`.

36. What is the output of below code ?

```
console.log([]+[]);
console.log([1]+[]);
console.log([1]+"abc");
```

Output: “”,”1”,”1abc”

Video Explaination: https://www.instagram.com/p/DHixQfaiwKE/

Explaination:

When you use the `+` operator with arrays, JavaScript first converts the arrays into strings using their `toString()` method:

- `[] + []`

  An empty array (`[]`) converts to an empty string (`""`). So, `"" + ""` results in an empty string.

- `[1] + []`

  The array `[1]` converts to `"1"`, and `[]` converts to `""`. So, `"1" + ""` results in `"1"`.

- `[1] + "abc"`

  Again, `[1]` converts to `"1"`. Then `"1" + "abc"` concatenates to give `"1abc"`.

In summary, the arrays are turned into strings before they are concatenated.

---

37. Guess the output ?

```
function getAge(...args){
  console.log(typeof args)
}

getAge(21)
```

Output: “object”

Video Explaination: https://www.instagram.com/p/DHdo1CLi4ud/

Explaination:

The output of the code will be **"object"**. This is because :

1. **Rest Parameters (`...args`):**

   The `...args` in the function definition is JavaScript's **rest parameter syntax**. It collects all passed arguments into a single array. So even though you pass `21`, `args` becomes `[21]` (an array).

2. `typeof` **Behavior:**

   In JavaScript, arrays are technically a type of **object**. When you use `typeof` on an array (like `args`), it returns `"object"`, not `"array"`.

---

38. Guess the output ?

```
const obj = {
  a: "one",
  b: "two",
  a: "three"
}

console.log(obj);
```

Output:

{

a: "three",

b: "two"

}

Video Explaination: https://www.instagram.com/p/DHDwosvCVdx/

Explaination:

This is because.,

1. **Duplicate Keys in Objects:**

   In JavaScript, if an object has **duplicate keys**, the **last occurrence** of the key will overwrite the previous value. This happens silently (no error is thrown).

2. **What Happens Here:**

   - The key `a` is first assigned `"one"`.
   - Then, `a` is redefined with `"three"` (overwriting the first `a`).
   - The key `b` stays `"two"`.

```
const obj = {
  a: "one",    // overwritten by the next 'a'
  b: "two",
  a: "three"   // this is the final value of 'a'
};
```

---

39. Guess the output ?

```
var z = 1, y = z = typeof y;
console.log(y);
```

Output: `"undefined"`

Video Explaination: https://www.instagram.com/p/DGlAOTsimyX/

Explaination:

The output of the code will be `"undefined"`. Here's why:

Step-by-Step Breakdown:

1. **Variable Hoisting**:

   Variables `z` and `y` are declared (due to `var` hoisting) but not yet initialized. At this stage:

- `z` is `undefined` (temporarily).
- `y` is `undefined` (temporarily).

2. **Initial Assignments**:

```
va = 1; // z is now assigned `1`.
```

3. **The Tricky Line**:

```
y = z = typeof y; // Evaluated from right-to-left!
```

   - **Step 1**: `typeof y` is evaluated.

     Since `y` is declared (due to hoisting) but **not yet initialized**, `typeof y` returns `"undefined"`.

   - **Step 2**: Assign `"undefined"` to `z`.

     Now, `z = "undefined"` (overwriting its earlier value of `1`).

   - **Step 3**: Assign `z`'s value (`"undefined"`) to `y`.

     Now, `y = "undefined"`.

4. **Final Result**:

   `console.log(y);` → Outputs `"undefined"`.

Key Takeaways:

- **Hoisting**: Variables declared with `var` are hoisted and initialized to `undefined` before assignment.
- **Assignment Order**: `a = b = c` is evaluated right-to-left (`b = c` first, then `a = b`).
- **Overwriting**: The initial value of `z` (`1`) is overwritten by `typeof y`.

---

40. Guess the output ?

```
console.log(false || null || "Hello");
console.log(false && null && "Hello");
```

Video Explaination: https://www.instagram.com/p/DGYC5Kqiwm1/

Explaination:

The outputs will be:

`"Hello"` and `false`.

Explanation:

1. First Line: `false || null || "Hello"`

- **Logical OR (`||`)** returns the **first truthy value** it finds.
- Check each value left-to-right:
  - `false` → falsy → move to next.
  - `null` → falsy → move to next.
  - `"Hello"` → truthy (non-empty string) → **return "Hello"**.

2. Second Line: `false && null && "Hello"`

- **Logical AND (`&&`)** returns the **first falsy value** it finds.
- Check each value left-to-right:
  - `false` → falsy → **return** `false` **immediately**.

Key Rules:

- `||` **(OR)**: Stops at the first truthy value.
- `&&` **(AND)**: Stops at the first falsy value.
- If all values are checked (e.g., all truthy for `||` or all falsy for `&&`), the last value is returned.

---

41. Guess the output ?

```
const numbers = [1,2,3,4,5];
const [x,...y] = numbers;
console.log(x,y);
```

Video Explaination: https://www.instagram.com/p/DGQT6ioCreI/

Explaination:

The output will be:

`1 [2, 3, 4, 5]`

1. **Array Destructuring**:

   ```
   const [x, ...y] = [1, 2, 3, 4, 5];
   ```

   - `x` is assigned the **first element** of the array (`1`).
   - `...y` uses the **rest operator** (`...`) to collect the **remaining elements** into a new array `y`.

2. **Result**:
   - `x` → `1`
   - `y` → `[2, 3, 4, 5]`

Key Rules:

- The rest operator (`...`) **must always be the last element** in destructuring.
- It captures all remaining elements into an array.

---

42. Guess the output ?

```
const str = "abc"+ + "def";
console.log(str);
```

Video Explaination: https://www.instagram.com/p/DGNt9DOC5C3/

Explaination:

The output will be:

`"abcNaN"`

1. **Code Breakdown**:

```
const str = "abc" + + "def";
```

   - The `+ + "def"` part is key. Let's break it down:
     - The first `+` is a **string concatenation operator**.
     - The second `+` is a **unary plus operator** (attempting to convert `"def"` to a number).
2. **Unary Plus Operation**:
   - `+ "def"` tries to convert the string `"def"` to a number.
   - Since `"def"` is not a valid number, this results in `NaN` (Not-a-Number).
3. **Concatenation**:
   - Now the expression becomes: `"abc" + NaN`.
   - JavaScript converts `NaN` to the string `"NaN"` during concatenation.
   - Final result: `"abc" + "NaN"` → `"abcNaN"`.

Key Takeaway:

- The unary `+` operator converts values to numbers.
- Invalid conversions (like `"def"`) produce `NaN`.
- `NaN` becomes `"NaN"` when concatenated with a string.

---

43. Guess the output ?

```
let newlist = [1].push(2);
console.log(newlist.push(3));
```

Video Explaination: https://www.instagram.com/p/DGLKah0iB9G/

Explaination:

The code will throw an error:

`"TypeError: newlist.push is not a function"`

1. `[1].push(2)`:
    - The `push()` method adds `2` to the array `[1]`, making it `[1, 2]`.
    - **But** `push()` returns the **new length of the array** (not the array itself).
    - So `newlist` is assigned the value `2` (the new length of the array).

```
let newlist = [1].push(2); // newlist = 2 (a number)
```

2. `newlist.push(3)`:
    - `newlist` is `2` (a number), **not an array**.
    - Numbers do **not** have a `push()` method → this causes an error.

Why This Happens:

- `push()` modifies the original array but returns the **length**, not the array.
- We tried to use `push()` on a number (`2`), which is invalid.

Key Takeaway:

`push()` returns the array's **length**, not the array itself. Always work directly with the array variable.

---

44. Guess the output ?

```
console.log(0 || 1);
console.log(1 || 2);
```

```
console.log(0 && 1);
console.log(1 && 2);
```

Video Explaination: https://www.instagram.com/p/DGGCcEUisBu/

Explaination:

The outputs will be:

`1`, `1`, `0`, `2`

1. `console.log(0 || 1);`

- **Logical OR (`||`)** returns the **first truthy value**.
- `0` is falsy → check next value.
- `1` is truthy → **return** `1`.

  Output: `1`

---

2. `console.log(1 || 2);`

- **Logical OR (`||`)** stops at the first truthy value.
- `1` is truthy → **return** `1` **immediately**.

  Output: `1`

---

3. `console.log(0 && 1);`

- **Logical AND (`&&`)** returns the **first falsy value**.
- `0` is falsy → **return** `0` **immediately**.

  Output: `0`

---

4. `console.log(1 && 2);`

- **Logical AND (`&&`)** returns the **last value** if all are truthy.
- `1` is truthy → check next value.
- `2` is truthy → **return** `2`.

  Output: `2`

Key Rules:

- `||` **(OR)**:
  - Returns the first **truthy** value.
  - If all are falsy, returns the last value.
- `&&` **(AND)**:
  - Returns the first **falsy** value.
  - If all are truthy, returns the last value.

---

45. Guess the output ?

```
console.log(data());
var data = function(){
 return "1";
}
```

Video Explaination: https://www.instagram.com/p/DF7sDc8C7Zw/

Explaination:

The code will throw an error:

`TypeError: data is not a function`

Why This Happens:

1. **Variable Hoisting**:

   JavaScript hoists the declaration of `data` (using `var`) to the top of the scope, but **not its initialization**.

   At the time of `console.log(data())`:

   - `data` exists (due to hoisting), but its value is `undefined`.
   - `undefined` is not a function → error when trying to call `data()`.

2. **Execution Order**:

```
console.log(data()); // ❌ Called BEFORE `data` is assigned a function
var data = function() { // Function expression (not hoisted)
  return "1";
};
```

Key Fix:

Use a **function declaration** (which is fully hoisted):

```
console.log(data()); // Works (output: "1")
function data() {    // Function declaration (hoisted)
  return "1";
}
```

Comparison:

Scenario	Hoisting Behavior	Result
`var data = function()`	Only `data` is hoisted (as `undefined`)	Error (not a function)
`function data()`	Entire function is hoisted	Works

Key Takeaway:

Function expressions assigned to variables (`var x = function(){}`) are **not hoisted** as callable functions. Function declarations (`function x(){}`) are fully hoisted.

---

46. Guess the output ?

```
const arr = [1,2,3];
arr[5] = 6;
console.log(arr.length);
```

Video Explaination: https://www.instagram.com/p/DF5IdHRijnU/

Explaination:

The output will be `6`.

1. **Initial Array**:

   ```
   const arr = [1, 2, 3]; // arr.length is 3
   ```

   The array starts with indexes 0, 1, and 2.

2. **Assigning to Index 5**:

   ```
   arr[5] = 6;
   ```

   - JavaScript automatically **expands the array** to accommodate index 5.
   - Indexes 3 and 4 become "empty slots" (they are not initialized with any value).

   The array now looks like:

   ```
   [1, 2, 3, empty, empty, 6]
   ```

3. **Array Length**:

   - The `length` of an array is always **1 more than the highest index** assigned.
   - Here, the highest index is 5 → `length = 5 + 1 = 6`.

Final Output:

```
console.log(arr.length); // 6
```

Key Takeaways:

- Arrays in JavaScript are **dynamic** and expand when you assign values to non-existing indexes.
- Empty slots are not the same as `undefined` – they are gaps in the array.

- The `length` property reflects the highest assigned index + 1, even if there are gaps.

---

47. Guess the output ?

```
const obj = {
  a:1
}
obj.a = 2;
console.log(obj);
```

Video Explaination: https://www.instagram.com/p/DF2i864Ca1E/

Explaination:

The output will be:
`{ a: 2 }`

1. **Object Creation**:

   ```
   const obj = { a: 1 };
   ```

   - Creates an object with a property `a` initialized to `1`.

2. **Modifying the Property**:

   ```
   obj.a = 2;
   ```

   - `const` only prevents reassigning the **variable** `obj` to a new object.
   - **Properties of the object can still be modified**.
   - The value of `a` changes from `1` to `2`.

3. **Final Output**:

   ```
   console.log(obj); // { a: 2 }
   ```

Key Takeaways:

- `const` protects the **variable** (`obj`) from being reassigned, not its **contents**.
- Objects declared with `const` can have their properties updated.

---

48. Guess the output ?

```
let x = {
  a: undefined,
  b: null
}
console.log(JSON.stringify(x))
```

Video Explaination: https://www.instagram.com/p/DF0BHY1iDV7/

Explaination:

The output will be:

`{"b":null}`

1. **Object** `x`:

   ```
   let x = {
     a: undefined,
     b: null
   };
   ```

2. `JSON.stringify()` **Behavior**:
   - `undefined` → **Excluded** from JSON output.
   - `null` → **Included** as `null` in JSON.
3. **Result**:
   - Property `a` (with value `undefined`) is **removed**.
   - Property `b` (with value `null`) is **kept**.

Final Output:

```
console.log(JSON.stringify(x)); // {"b":null}
```

Key Takeaways:

- `JSON.stringify()` ignores properties with `undefined` values.
- `null` is a valid JSON value and is preserved.

---

49. Guess the output ?

```
console.log(true + 1);
console.log(true+"1");
```

Video Explaination: https://www.instagram.com/p/DFxaRLyCu6n/

Explaination:

The outputs will be:

`2` and `"true1"`

1. `console.log(true + 1);`

- **Boolean Conversion**: JavaScript converts `true` to its numeric equivalent:
  - `true` → `1`
- **Addition**:

  ```
  1 (true) + 1 = 2
  ```

  Output: `2`

2. `console.log(true + "1");`

- **String Concatenation**: When `+` is used with a string (`"1"`), JavaScript converts both values to strings.

- `true` → `"true"`
- `"1"` stays as `"1"`

- **Concatenation**:

```
"true" + "1" = "true1"
```

Output: `"true1"`

Key Rules:

- `+` **with numbers**: Booleans convert to numbers (`true` → `1`, `false` → `0`).
- `+` **with strings**: Converts all values to strings and concatenates them.

---

50. Guess the output ?

```
const str = "hello";
str.data = "val";
console.log(str.data);
```

Video Explaination: https://www.instagram.com/p/DFu24SVCn3X/

Explaination:

The output will be `undefined`.

1. **Primitive Strings vs. String Objects**:
   - `"hello"` is a **primitive string**, not an object.
   - Primitives **cannot have properties** added to them.
2. **What Happens When You Try**:
   - When you write `str.data = "val"`, JavaScript temporarily converts the primitive string to a **String object** to allow the assignment.
   - However, this temporary object is **discarded immediately**. The property `data` is not saved.

3. **Accessing** `str.data`:
    - When you read `str.data`, JavaScript creates a **new temporary String object** (which doesn't have the `data` property).
    - Thus, `console.log(str.data)` returns `undefined`.

Key Takeaway:

Primitive strings (like `"hello"`) **cannot hold custom properties**. Use an object (e.g., `{}`) if you need to add properties.

---

27 Problem solving questions:

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

(Most Most Asked question)

```
const removeDuplicatesWay1 = (array) => {
  let uniqueArr = [];

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

  return uniqueArr;
};

removeDuplicatesWay1([1, 2, 1, 3, 4, 2, 2, 1, 5, 6]);
```

```
// ------------------------- (or)-----------------------------

function removeDuplicatesWay2(arr) {
  /* Use the Set object to remove duplicates. This works
   because Sets only store unique values */
  return Array.from(new Set(arr));
  // return [...new Set(arr)] ⇒ another way
}

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

2. 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)

3. 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"));
```

4. 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));
```

---

5. 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");
```

---

6. 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)

7. 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)

8. 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");
```

9. Find the smallest word in a given sentence ?

```
function findSmallestWord() {
  const sentence = "Find the smallest word";
  const words = sentence.split(' ');
  let smallestWord = words[0];

  for (let i = 1; i < words.length; i++) {
     if (words[i].length < smallestWord.length) {
        smallestWord = words[i];
     }
  }
  console.log(smallestWord);
}
```

```
findSmallestWord();
```

---

10. Write a function sumOfThirds(arr), which takes an array arr as an argument. This function should return a sum of every third number in the array, starting from the first one.

Directions:

If the input array is empty or contains less than 3 numbers then return 0.

The input array will contain only numbers.

```
export const sumOfThirds = (arr) => {
  let sum = 0;
  for (let i = 0; i < arr.length; i += 3) {
    sum += arr[i];
  }
  return sum;
};
```

---

11. 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)

12. 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, 19, 7, 1, 1, 1, 2, 5, 1]);
// output: 3
```

---

13. 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])
```

---

14. 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]))
```

---

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

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'))
```

---

16. 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))
```

---

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

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

    // 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 array,recursively call traverse function
                traverse(arr[i]);
            } else {
    // If element is not an array, update max if needed
                if (arr[i] > max) {
                    max = arr[i];
                }
            }
        }
    }

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

    return max;
}
```

```
// Example usage:
const array = [
[3, 4, 58],
[709, 8, 9, [10, 11]], [111, 2]
];
console.log("Largest element:", findLargestElement(array));
```

Time complexity: O(N)

---

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

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

// Loop through string & 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("helaalo");
console.log(result);
// Output: { h: 1, e: 1, l: 2, o: 1 }
```

Time complexity: O(N)

19. 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 sorted partitions with the pivot & 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)

20. 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)

21. 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 and
       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 ChatGPT"
```

Time complexity: O(N)

22. 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 original 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]
```

---

23. 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"}}}
```

---

24. 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]
```

---

25. Find the 2nd largest element from a given array ? [100,20,112,22]

```
function findSecondLargest(arr) {
    if (arr.length < 2) {
        throw new Error(`Array must contain
         at least two elements.`);
    }

    let largest = -Infinity;
    let secondLargest = -Infinity;

    for (let i = 0; i < arr.length; i++) {
        if (arr[i] > largest) {
            secondLargest = largest;
            largest = arr[i];
        } else if (arr[i] > secondLargest && arr[i] < largest) {
            secondLargest = arr[i];
        }
    }

    if (secondLargest === -Infinity) {
        throw new Error(`There is no second largest
        element in the array.`);
    }

    return secondLargest;
}

// Example usage:
```

```
const array = [10, 5, 20, 8, 12];
console.log(findSecondLargest(array)); // Output: 12
```

26. Program challenge: Find the pairs from given input ?

input1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];

input2 = 10;

output = [[4, 6], [3, 7], [2, 8], [1, 9]]

```
function findPairs(input1, input2) {
  const pairs = [];
  const seen = new Set();

  for (const num of input1) {
    const complement = input2 - num;
    if (seen.has(complement)) {
      pairs.push([complement, num]);
    }
    seen.add(num);
  }

  return pairs;
}

const input1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
const input2 = 10;

const output = findPairs(input1, input2);
console.log(output);
// [[1, 9], [2, 8], [3, 7], [4, 6], [5, 5]]
```

27. Write a javascript program to get below output from given input ?

I/P: abbcccddddeea

O/P: 1a2b3c4d2e1a

```
function encodeString(input) {
    if (input.length === 0) return "";

    let result = "";
    let count = 1;

    for (let i = 1; i < input.length; i++) {
        if (input[i] === input[i - 1]) {
            count++;
        } else {
            result += count + input[i - 1];
            count = 1;
        }
    }

    // Add the last sequence
    result += count + input[input.length - 1];

    return result;
}

const input = "abbcccddddeea";
const output = encodeString(input);
console.log(output);
// Outputs: 1a2b3c4d2e1a
```

52 Reactjs Interview questions & Answers

1. What is React?

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

2. What are the features of React ?

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

3. 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.

4. What is DOM ?

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

---

5. 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.

Rendering Efficiency:

- **Virtual DOM**: Updates to the Virtual DOM are typically faster because they occur in memory and are not directly reflected in the browser.
- **Actual DOM**: Manipulating the actual DOM is slower due to its complexity and the potential for reflows and repaints.

---

6. 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 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>
    );
  }
}
```

---

7. 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.

---

8. 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;
```

---

9. 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.

---

10. What is props drilling ?

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

---

11. 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.

---

12. 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.

---

13. 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.

---

14. What is useCallback ?

- useCallback caches the function defination 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

---

15. 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.

---

16. 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")
},[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
      or unsubscribing etc here`)
    }
})
```

---

17. 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.

---

18. 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>
  );
}
```

---

19. What are fragments in react ?

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

---

20. 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 comparision 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)}
        } />
      </label>
      <Greeting name={name} />
    </>
  );
}
```

21. 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/

---

22. Whatare 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 whith out 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/keyboard-32-12.mp3'
      ></audio>
      <button onClick={playAudio}>Play Audio</button>
      <button onClick={pauseAudio}>Pause Audio</button>
    </div>
```

```
  );
}
```

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

---

23. 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()}>
```

```
      Focus Input
      </button>
    </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

---

24. 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 erros 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 UI. */
    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>
```

25. 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)
```

26. 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;
```

---

27. 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 in 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;
```

---

28. 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
```

```
  return { data, loading, error };
}

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

  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;
```

29. What is context api in react ?

- Context api is a feature in react by using which we can share the data across the application with out having to pass the data manually through every level of component tree.

- It solves the problem of props drilling in react.

We need to follow 3 main steps to implement context api in our application.

- Create context
- Create Provider and wrap this provider around root level component and pass the data.
- Create consumer and utilize data using useContext.

Examples Where we can use context api:

- Shopping cart: Managing cart data in ecommerce application and share between product listings and checkout pages.
- Authentication: sharing user information across the application.
- Data fetching : Sharing fetched data across multiple components that need to display this data. (for displaying search results when user makes a search request).

---

30. Give an example of context api usage ?

Example: Once user loggedin, I wanted to share the user information between the components.

- **Step - 1:** We need to create a context using **createContext** method in userContext.js file.

UserContext.js

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

const UserContext = createContext();
const UserProvider = UserContext.Provider;
const UserConsumer = UserContext.Consumer;
```

```
export {UserProvider, UserConsumer};
```

- **Step - 2:** We need to wrap the root level component with provider and pass the user Information.

App component:

```
import React from "react";
import {ComponentA} from "./ComponentA.js";
import {UserProvider} from "./UserContext.js";

const App = () => {
  const userinfo = {
    id:"1",
    name:"saikrishna"
  }

  return (
    <div>
      <UserProvider value={userinfo}>
        <ComponentA/>
      </UserProvider>
    </div>
  );
};
```

- **Step - 3:** In componentA, We can get the data using useContext and utilize the user Information.

ComponentA:

```
import React,{useContext} from "react";
import {UserConsumer} from "./UserContext.js";
```

```
export const ComponentA = () => {
  const {id,name} = useContext(UserConsumer);

  return <div>Hello {id}{name}</div>
}
```

---

31. What is Strict mode in react ?

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

- Checks if there are any depricated 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>
);

============================================================
// 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

---

32. 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

---

33. 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.

---

34. 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 DOM!
          </p>
        </MyPortal>
      </div>
    );
  }
}

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

35. What are synthetic events in react ?

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

36. 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.

37. 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 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. where as ssr has faster initial load times as server send pre-rendered html page to the browser.
- **Subsequent interactions:** subsequent interactions in csr involves dynamic updates with out requiring full page reload whereas, ssr requires full page reload as server generates new html for every interactions.
- **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.

---

38. What is react-router ?

React-router is a javascript library which is used to implement routing in react applications.
It allows us to navigate between views or pages with out refreshing the page.

- **Router:** It wraps the entire application and provides the routing context for the application. It contains 2 types of routers,
  Browser router and Hash router.
- **Route:** It contains mapping between urlpath and the component. When the url matches, respective component will be rendered.
- **Link:** is used to create the navigation links which user can click to navigate to different routes.
- **switch:** is used to render the first matching route among its children. It ensures only one route is rendered.

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dc

function Home() {
  return <h2>Home</h2>;
}

function About() {
  return <h2>About</h2>;
}

function Contact() {
  return <h2>Contact</h2>;
}

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">
              Home
              </Link>
            </li>
            <li>
              <Link to="/about">
              About
              </Link>
            </li>
            <li>
              <Link to="/contact">
              Contact
              </Link>
```

```
                </li>
              </ul>
            </nav>

            <Switch>
              <Route exact path="/"
              component={Home}
              />
              <Route path="/about"
              component={About}
              />
              <Route path="/contact"
              component={Contact}
              />
            </Switch>
          </div>
        </Router>
    );
}

export default App;
```

39. What are protected routes in react ?

- By using the protected routes, we can define which users/user roles have access to specific routes or components.

40. What is the difference between useEffect and useLayoutEffect ?

- useeffect is asynchronous where as uselayouteffect is synchronous

- uselayouteffect runs after browser calculation of dom measurements and before browser repaints the screen whereas useffect runs parallely.
- uselayouteffect really needed when we need to do something based on layout of the dom, if we want to measure dom elements and do something etc.

Ref : https://www.youtube.com/watch?v=wU57kvYOxT4

https://mtg-dev.tech/blog/real-world-example-to-use-uselayouteffect

---

41. 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
    </button>
  );
}

export default ChildComponent;
```

---

42. 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)}
      }
      >Get</button>
    </>
  );
}
```

Child component:

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

---

43. 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;
```

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

- **Create a custom hook for counter:**

useCounter.js:

```
export const useCounter = () => {
  const [counter, setCounter] = useState(0);

  const increment = () => {
    setCounter(counter + 1);
  };

  const decrement = () => {
    setCounter(counter - 1);
  };

  return {
    counter,
    increment,
    decrement,
  };
};
```

- **Utilize useCounter in our component:**

component.js:

```
import { useCounter } from './useCounter.js';

const App = () => {
  const { counter, increment, decrement } = useCounter();

  return (
```

```
    <div>
      <button onClick={decrement}>Decrease</button>
      <div>Count: {counter}</div>
      <button onClick={increment}>Increase</button>
    </div>
  );
};
```

---

45. Machine coding question: Dynamic checkbox counter

Display 4 checkboxes with different names and a button named selectall

User can select each checkbox

Select all button click will check all checkboxes

Button should be disabled once all checkboxes are selected.

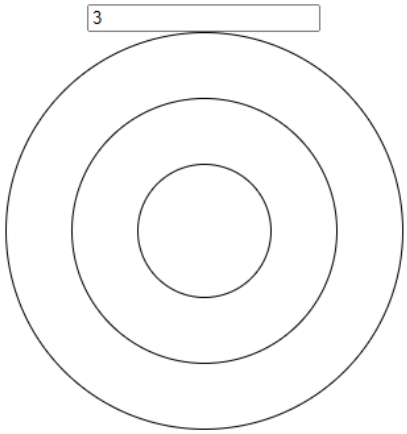
Display selected checkboxes count and names in ui.

```
import React, { useState } from 'react';
import { render } from 'react-dom';

const Checkbox = ({ label, checked, onChange }) => {
  return (
    <div>
      <label>
        <input type="checkbox"
        checked={checked}
        onChange={onChange} />
        {label}
      </label>
    </div>
  );
```

```
};
const App = () => {
const [checkboxes, setCheckboxes] = useState([
   { id: 1, label: 'Checkbox 1', checked: false },
   { id: 2, label: 'Checkbox 2', checked: false },
   { id: 3, label: 'Checkbox 3', checked: false },
   { id: 4, label: 'Checkbox 4', checked: false },
 ]);

const [selectAllDisabled,setSelectAllDisabled]= useState(
false
);

 const handleCheckboxChange = (id) => {
  const updatedCheckboxes = checkboxes.map((checkbox) =>
   checkbox.id === id
    ? { ...checkbox, checked: !checkbox.checked }
    : checkbox
  );
  setCheckboxes(updatedCheckboxes);
  const allChecked = updatedCheckboxes.every(
  (checkbox) => checkbox.checked
  );
  setSelectAllDisabled(allChecked);
 };

 const handleSelectAll = () => {
  const updatedCheckboxes = checkboxes.map(
  (checkbox) => ({
   ...checkbox,
   checked: !selectAllDisabled,
  }));
  setCheckboxes(updatedCheckboxes);
  setSelectAllDisabled(!selectAllDisabled);
 };
```

```
  const selectedCheckboxes = checkboxes.filter(
  (checkbox) => checkbox.checked
  );
  const selectedCount = selectedCheckboxes.length;

  return (
    <div>
      {checkboxes.map((checkbox) => (
        <Checkbox
          key={checkbox.id}
          label={checkbox.label}
          checked={checkbox.checked}
          onChange={
          () => handleCheckboxChange(checkbox.id)
          }
        />
      ))}
      <button
      onClick={handleSelectAll}
      disabled={selectAllDisabled}>
        {selectAllDisabled ? 'Deselect All' : 'Select All'}
      </button>
      <p>Selected: {selectedCount}</p>
      <ul>
        {selectedCheckboxes.map((checkbox) => (
          <li key={checkbox.id}>{checkbox.label}</li>
        ))}
      </ul>
    </div>
  );
};

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

46. Create a nested circles based on user input like below image.



Solution:

Create App component and circle component. We will use recursion concept to achieve this.

App.js:

```
import React, { useState } from "react";
import { Circle } from "./Circle.js";
import "./styles.css";

export default function App() {
 const [num, setNum] = useState(0);

 const handleInput = (e) => {
  setNum(e.target.value);
 };

 return (
  <div className="nested-circles-container">
```

```
      <input
        onChange={(e) => handleInput(e)}
        placeholder="Enter number of circles"
        type="number"
      />
      <Circle numbCircles={num}></Circle>
    </div>
  );
}
```

App.css/Style.css:

```
.nested-circles-container {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
}
```

Circle.js:

```
import React from "react";
import "./Circle.css";

export const Circle = ({ numbCircles }) => {
  let size = numbCircles * 100;
  return (
    <div
      className="circle-new"
      style={{
        width: `${size}px`,
        height: `${size}px`,
      }}
    >
```

```
      {numbCircles > 1 &&
       <Circle numbCircles={numbCircles - 1}>
       </Circle>
       }
    </div>
  );
};
```

Circle.css:

```
.circle-new {
  border-radius: 100%;
  border: 1px solid black;
  display: flex;
  justify-content: center;
  align-items: center;
}
```

---

47. What are the various design patterns in react ?

- Compound pattern
- HOC Pattern
- Render props pattern
- Container/Presentational pattern

---

48. What is compound pattern ?

- By using this compound pattern we will create multiple components that work together to perform a single task.

- We can achieve this using context api.
- This promotes separation of concerns between parent and child components, promotes reusability of logic and maintainability.

---

49. What is render props pettern ?

- With the Render Props pattern, we pass components as props to other components. The components that are passed as props can in turn receive props from that component.
- Render props make it easy to reuse logic across multiple components.

https://javascriptpatterns.vercel.app/patterns/react-patterns/render-props

---

50. What is Container/Presentational pattern ?

It enforces separation of concerns by separating the view from the application logic.

- We can use the Container/Presentational pattern to separate the logic of a component from the view. To achieve this, we need to have a:
  - **Presentational** Component, that cares about **how** data is shown to the user.
  - **Container** Component, that cares about **what** data is shown to the user.

For example, if we wanted to show listings on the landing page, we could use a container component to fetch the data for the recent listings, and use a presentational component to actually render this data.

https://javascriptpatterns.vercel.app/patterns/react-patterns/conpres

---

51. What is React.memo ?

- React.memo is a Higher order component provided by react that memoizes the functional components.
- It caches the result of component's rendering and rerenders the component only when the props have changed.

```
const MyComponent = React.memo((props) => {
  console.log('Rendering MyComponent');
  return (
    <div>
      <h1>Hello, {props.name}!</h1>
      <p>{props.message}</p>
    </div>
  );
});
```

---

52. Differences between dependencies and devdependencies ?

- **dependencies:** These are the packages that your project needs to run in production. These are essential for the application to function correctly.
- **devDependencies:** These packages are only needed during the development phase.
  - Eg: Jest, react-developer-tools etc.

Reactjs Scenario based Questions:

1. How to send the data from parent component to child component in react ?

Ans. To send data from a parent component to a child component in React, We will use **props** (short for "properties"). Here's a step-by-step explanation with examples:

a. Pass Data via Props in the Parent Component

- Include the child component in the parent's JSX.
- Attach data to it as a custom attribute (e.g., `dataName={value}`).

```
// ParentComponent.jsx

import ChildComponent from './ChildComponent';

function ParentComponent() {
  const message = "Hello from Parent!";// Data to send
  const user = { name: "Alice", age: 30 };

  return (
    <div>
      {/* Pass data as props */}
      <ChildComponent text={message} userData={user} isAdmin={false}/>
    </div>);
}
```

b. Access Props in the Child Component

- The child receives data via its `props` parameter (or via destructuring).

Method A: Using `props` Directly

```
// ChildComponent.jsx

function ChildComponent(props) {
  return (
    <div>
      <h1>{props.text}</h1>
```

```
      <p>Name: {props.userData.name}, Age: {props.userData.age}</p>
      {props.isAdmin && <button>Admin Panel</button>}
    </div>);
}
```

Method B: Destructuring Props

```
// ChildComponent.jsx (cleaner with destructuring)

function ChildComponent({ text, userData, isAdmin }) {
  return (
    <div>
      <h1>{text}</h1>
      <p>Name: {userData.name}, Age: {userData.age}</p>
      {isAdmin && <button>Admin Panel</button>}
    </div>);
}
```

---

2. How to call parent component method from child component in reactjs ?

To call a parent component's method from a child component in React, follow these steps:

1. Pass the Parent's Method as a Prop to the Child

- In the parent component, pass the method as a prop (e.g., `onAction`).
- In the child component, invoke the prop when needed (e.g., on a button click).

Example with Functional Components:

```jsx
// ParentComponent.jsx
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const handleChildAction = (data) => {
    console.log('Method called from child!', data);
  };

  return (
    <div>
      <ChildComponent onAction={handleChildAction} />
    </div>);
}
```

```jsx
// ChildComponent.jsx
import React from 'react';

function ChildComponent({ onAction }) {
  const handleClick = () => {
// Call parent method with data
        onAction('Hello from child');
  };

  return <button onClick={handleClick}>
  Trigger Parent Method
  </button>;
}
```

Example with Class Components:

```jsx
// ParentComponent.jsx
import React from 'react';
import ChildComponent from './ChildComponent';
```

```
class ParentComponent extends React.Component {
  handleChildAction = (data) => {
    console.log('Method called from child!', data);
  };

  render() {
    return <ChildComponent onAction={this.handleChildAction} />;
  }
}
```

```
// ChildComponent.jsx
import React from 'react';

class ChildComponent extends React.Component {
  render() {
    return (
      <button onClick={() =>
      this.props.onAction('Hello from child')}
      >
        Trigger Parent Method
      </button>);
  }
}
```

3.How to bind array/array of objects to dropdown in react ?

Step 1: Define Array or Array of Objects in your component

```jsx
import React, { useState } from 'react';

const DropdownExample = () => {
  // Simple Array
  const fruits = ['Apple', 'Banana', 'Orange'];

  // Array of objects
  const countries = [
    { id: 1, name: 'India' },
    { id: 2, name: 'USA' },
    { id: 3, name: 'Canada' },
  ];

  // State hooks for selected values
  const [selectedFruit, setSelectedFruit] = useState('');
  const [selectedCountryId, setSelectedCountryId] = useState('');

  return (
    <div>
      {/* Dropdown using simple array */}
      <select value={selectedFruit} onChange={(e) => setSelectedFruit(e.target.
value)}>
        <option value="" disabled>Select Fruit</option>
        {fruits.map((fruit, index) => (
          <option key={index} value={fruit}>
            {fruit}
          </option>
        ))}
      </select>

      <p>Selected Fruit: {selectedFruit}</p>

      {/* Dropdown using array of objects */}
      <select value={selectedCountryId} onChange={(e) => setSelectedCountryI
```

```
d(e.target.value)}>
        <option value="" disabled>Select Country</option>
        {countries.map((country) => (
          <option key={country.id} value={country.id}>
            {country.name}
          </option>
        ))}
      </select>

      <p>Selected Country ID: {selectedCountryId}</p>
    </div>
  );
};

export default DropdownExample;
```

📌 Explanation:

- **Mapping (`array.map`)**: Used to loop through the array and render options.
- **Keys (`key`)**: Essential for React to track each rendered item.
- **State (`useState`)**: Maintains selected values.
- **onChange handler**: Updates the selected values in the state.

🖥️ Output Example:

- Selecting **Banana** sets `selectedFruit` to `'Banana'`.
- Selecting **India** sets `selectedCountryId` to `1`.

---

4. Create a lazy loaded component in react ?

1. Create your Lazy Loaded Component (e.g., `MyComponent.jsx`)

```jsx
// MyComponent.jsx
import React from 'react';

const MyComponent = () => {
  return <div>This is a lazy loaded component!</div>;
};

export default MyComponent;
```

2. Lazy load this component in your main application file (e.g., `App.jsx`)

```jsx
// App.jsx
import React, { lazy, Suspense } from 'react';

const LazyMyComponent = lazy(() => import('./MyComponent'));

const App = () => {
  return (
    <div>
      <h1>Main App Component</h1>

      <Suspense fallback={<div>Loading component...</div>}>
        <LazyMyComponent />
      </Suspense>
    </div>
  );
};

export default App;
```

Explanation:

- `lazy`: This function allows you to dynamically import a component.
- `Suspense`: Wraps lazy-loaded components and provides a fallback UI (e.g., a spinner or loader) while the component loads.

---

5. How to display data entered by the user in another textbox ?

Step-by-step code:

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

const DisplayInputExample = () => {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <div>
      <inputtype="text"
        placeholder="Enter something..."
        value={inputValue}
        onChange={handleChange}
      />

      <br /><br />

      <inputtype="text"
        placeholder="Displayed value..."
```

```
        value={inputValue}
        readOnly
      />
    </div>
  );
};

export default DisplayInputExample;
```

Explanation:

- `useState` hook manages the entered value.
- The first `input` captures user input via `onChange`.
- The second `input` displays the entered text, set as `readOnly` to prevent manual editing.

This allows you to see the entered text mirrored immediately in another textbox.

---

6. How to conditionally render an element or text in react ?

You can conditionally render elements or text in React using these common approaches:

Using the ternary (`? :`) operator:

```
function MyComponent({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
```

```
    </div>
  );
}
```

Using logical AND (`&&`) operator:

```
function MyComponent({ showMessage }) {
  return (
    <div>
      {showMessage && <p>This message appears if true.</p>}
    </div>
  );
}
```

Conditional rendering via variables:

```
function MyComponent({ isAdmin }) {
  let content;

  if (isAdmin) {
    content = <p>Admin Dashboard</p>;
  } else {
    content = <p>User Dashboard</p>;
  }

  return <div>{content}</div>;
}
```

Inline conditional rendering (short-circuiting):

```
function MyComponent({ notifications }) {
  return (
    <div>
      {notifications.length > 0 && <p>You have notifications!</p>}
    </div>
  );
}
```

Choose the approach based on clarity and complexity of your conditions.

---

7. How to perform debouncing in react ?

Step-by-step example:

```
import React, { useState, useEffect, useRef } from 'react';

function DebounceExample() {
  const [inputValue, setInputValue] = useState('');
  const [debouncedValue, setDebouncedValue] = useState('');

  const timerRef = useRef(null);

  const handleInputChange = (event) => {
    setInputValue(event.target.value);
  };

  useEffect(() => {
    // Clear the previous timer if inputValue changes
    if (timerRef.current) clearTimeout(timerRef.current);

    // Set new timer for debounce
```

```
    timerRef.current = setTimeout(() => {
      setDebouncedValue(inputValue);
    }, 500); // 500ms debounce interval

    // Cleanup function
    return () => clearTimeout(timerRef.current);
  }, [inputValue]);

  return (
    <div>
      <inputtype="text"
        value={inputValue}
        placeholder="Type something..."
        onChange={handleInputChange}
      />

      <p>Debounced Input: {debouncedValue}</p>
    </div>
  );
}

export default DebounceExample;
```

Explanation:

- `useState`: Manages the immediate (`inputValue`) and debounced values (`debouncedValue`).
- `useRef`: Stores the timer reference to clear previous timers effectively.
- `useEffect`: Triggers on each change to `inputValue`, resets the debounce timer, and updates `debouncedValue` only after the specified delay (500ms).

This technique ensures actions are performed **only after the user stops typing for a short period**, optimizing performance and preventing unnecessary function calls.

---

8. Create a component to fetch data from api in reactjs ?

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

const DataFetcher = ({ apiUrl }) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    setLoading(true);
    setError(null);

    fetch(apiUrl)
      .then((response) => {
        if (!response.ok) {
          throw new Error("Network response was not ok");
        }
        return response.json();
      })
      .then((json) => {
        setData(json);
        setLoading(false);
      })
      .catch((err) => {
        setError(err.message || "Something went wrong");
        setLoading(false);
      });
  }, [apiUrl]);
```

```
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
};

export default DataFetcher;
```

Usage:

```
<DataFetcher apiUrl="https://jsonplaceholder.typicode.com/posts/1" />
```

Explanation:

- Uses `useEffect` to fetch data when the component mounts or when `apiUrl` changes.
- Shows loading text while fetching.
- Handles and displays errors.
- Displays fetched data in a formatted JSON block.

---

9. Given two dropdowns, select 2nd dropdown options based on value selected in one dropdown in reactjs ? (Load states based on country selected)

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

const countries = [
  { id: "in", name: "India" },
  { id: "us", name: "USA" },
];

const states = {
  in: [
    { id: "mh", name: "Maharashtra" },
    { id: "tg", name: "Telangana" },
  ],
  us: [
    { id: "ca", name: "California" },
    { id: "ny", name: "New York" },
  ],
};

const CountryStateDropdown = () => {
  const [selectedCountry, setSelectedCountry] = useState("");
  const [selectedState, setSelectedState] = useState("");

  const handleCountryChange = (e) => {
    setSelectedCountry(e.target.value);
    setSelectedState(""); // reset state on country change
  };

  const handleStateChange = (e) => {
    setSelectedState(e.target.value);
  };

  return (
    <div>
      <label>
```

```
      Country:{" "}
      <select value={selectedCountry} onChange={handleCountryChange}>
        <option value="">-- Select Country --</option>
        {countries.map((country) => (
          <option key={country.id} value={country.id}>
            {country.name}
          </option>
        ))}
      </select>
    </label>

    <br />

    <label>
      State:{" "}
      <selectvalue={selectedState}
        onChange={handleStateChange}
        disabled={!selectedCountry}
      >
        <option value="">-- Select State --</option>
        {selectedCountry &&
          states[selectedCountry]?.map((state) => (
            <option key={state.id} value={state.id}>
              {state.name}
            </option>
          ))}
      </select>
    </label>
  </div>
 );
};

export default CountryStateDropdown;
```

How it works:

- When you select a country, it updates `selectedCountry`.
- The second dropdown populates with states corresponding to the selected country.
- If no country is selected, the states dropdown is disabled.
- Selecting a new country resets the selected state.

---

10. How to display dynamic html data in reactjs ?

To display dynamic HTML content (like a string containing HTML tags) safely in React, you use `dangerouslySetInnerHTML`.

```
function DynamicHtml({ htmlString }) {
  return (
    <div dangerouslySetInnerHTML={{ __html: htmlString }} />
  );
}

// Usage
const someHtml = "<p>This is <strong>dynamic</strong> HTML content.</p>";

<DynamicHtml htmlString={someHtml} />
```

Important Notes:

- React escapes HTML by default to prevent XSS attacks.
- `dangerouslySetInnerHTML` **bypasses React's escaping**, so **only use it with trusted content**.
- Always sanitize user-generated HTML before rendering with `dangerouslySetInnerHTML`.

---

11. How to add data into useState array in functional component in react ?

To add data into a `useState` array in a React functional component, you update the state by creating a new array with the existing items plus the new item.

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

function MyComponent() {
  const [items, setItems] = useState([]);

  const addItem = (newItem) => {
    setItems(prevItems => [...prevItems, newItem]);
  };

  return (
    <><button onClick={() => addItem("New item")}>Add Item</button>
      <ul>
        {items.map((item, idx) => (
          <li key={idx}>{item}</li>
        ))}
      </ul>
    </>
  );
}
```

Explanation:

- Use the updater function form of `setItems` to get the latest state.
- Create a new array with the spread operator `[...prevItems, newItem]` to add the new item.
- This ensures immutability and triggers a re-render.

12. Change focus/enable/disable textbox in child component based on parent component button click ?

To **change focus** or **enable/disable a textbox in a child component** based on a button click in the parent component in React, you can use **props** and **refs**.

Here's a minimal example showing both:

a. Parent controls enable/disable and focus on child's textbox

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

function Child({ disabled, shouldFocus }) {
  const inputRef = useRef(null);

  useEffect(() => {
    if (shouldFocus && inputRef.current) {
      inputRef.current.focus();
    }
  }, [shouldFocus]);

  return (
    <inputref={inputRef}
      type="text"
      disabled={disabled}
      placeholder={disabled ? "Disabled" : "Enabled"}
    />
  );
}

function Parent() {
  const [disabled, setDisabled] = useState(true);
  const [focusRequest, setFocusRequest] = useState(false);
```

```
  const toggleDisabled = () => {
    setDisabled(prev => !prev);
  };

  const focusInput = () => {
    setFocusRequest(true);
    // reset the focus request after focusing to allow future focus
    setTimeout(() => setFocusRequest(false), 0);
  };

  return (
    <><button onClick={toggleDisabled}>
        {disabled ? "Enable" : "Disable"} Textbox
      </button>
      <button onClick={focusInput} disabled={disabled}>
        Focus Textbox
      </button>
      <Child disabled={disabled} shouldFocus={focusRequest} />
    </>
  );
}

export default Parent;
```

How it works:

- `disabled` state in Parent controls whether the child input is enabled or disabled.
- `shouldFocus` prop signals the child to focus the textbox.
- The child uses `useEffect` to focus when `shouldFocus` changes to `true`.
- Parent resets `focusRequest` immediately so subsequent focus events can trigger again.

13. How to display dropdown value selected by user in another textbox ?

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

function DropdownToTextbox() {
  const [selectedValue, setSelectedValue] = useState("");

  const handleChange = (e) => {
    setSelectedValue(e.target.value);
  };

  return (
    <div>
      <select value={selectedValue} onChange={handleChange}>
        <option value="">Select an option</option>
        <option value="Apple">Apple</option>
        <option value="Banana">Banana</option>
        <option value="Orange">Orange</option>
      </select>

      <input type="text" value={selectedValue} readOnly />
    </div>
  );
}

export default DropdownToTextbox;
```

Explanation:

- The `select` element controls the `selectedValue` state.
- When the user selects a value, `handleChange` updates the state.

- The textbox's `value` is set to the current `selectedValue`, showing the selected dropdown option.
- `readOnly` is used on the textbox so the user cannot edit it manually.

---

14. How to display number of characters remaining functionality for textarea using react useRef?

React example showing how to display the number of characters remaining for a `<textarea>` using `useRef`:

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

function TextareaWithCharCount() {
  const maxLength = 100;
  const textareaRef = useRef(null);
  const [remaining, setRemaining] = useState(maxLength);

  const handleChange = () => {
    const length = textareaRef.current.value.length;
    setRemaining(maxLength - length);
  };

  return (
    <div>
      <textarearef={textareaRef}
        maxLength={maxLength}
        onChange={handleChange}
        rows={4}
        cols={40}
      />
      <div>{remaining} characters remaining</div>
```

```
    </div>
  );
}

export default TextareaWithCharCount;
```

How it works:

- `textareaRef` points to the textarea DOM node.
- On each `onChange` event, read `textareaRef.current.value.length` to get current length.
- Calculate remaining chars (`maxLength - length`) and update state.
- Display remaining count below the textarea.

---

15. Create a search textbox filter in reactjs

Here's a simple ReactJS example of a search textbox filter that filters a list of items as you type:

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

const SearchFilter = () => {
  const items = [
    "Apple",
    "Banana",
    "Orange",
    "Mango",
    "Pineapple",
    "Grapes",
    "Strawberry",
  ];

  const [searchTerm, setSearchTerm] = useState("");

  // Filter items based on search term (case insensitive)
  const filteredItems = items.filter(item =>
```

```
    item.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <inputtype="text"
        placeholder="Search fruits..."
        value={searchTerm}
        onChange={e => setSearchTerm(e.target.value)}
      />
      <ul>
        {filteredItems.length > 0 ? (
          filteredItems.map((item, idx) => <li key={idx}>{item}</li>)
        ) : (
          <li>No results found</li>
        )}
      </ul>
    </div>
  );
};

export default SearchFilter;
```

How it works:

- You type in the input box.
- `searchTerm` state updates on each keystroke.
- The list `items` is filtered to include only those matching the `searchTerm`.
- The filtered list displays below the input.

---

16. Display radio button data selected by user in another textbox

Here's a simple React example showing how to display the selected radio button value inside a textbox:

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

function RadioToTextbox() {
  const [selectedOption, setSelectedOption] = useState("");

  const handleChange = (e) => {
    setSelectedOption(e.target.value);
  };

  return (
    <div>
      <h3>Select an option:</h3>
      <label>
        <inputtype="radio"
          value="Option 1"
          checked={selectedOption === "Option 1"}
          onChange={handleChange}
        />
        Option 1
      </label>
      <br />
      <label>
        <inputtype="radio"
          value="Option 2"
          checked={selectedOption === "Option 2"}
          onChange={handleChange}
        />
        Option 2
      </label>
```

```
      <br />
      <br />

      <label>
        Selected option:
        <input type="text" value={selectedOption} readOnly />
      </label>
    </div>
  );
}

export default RadioToTextbox;
```

17. Create an error boundary component in react ?

```
import React from 'react';

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

  static getDerivedStateFromError(/* error */) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    if (this.props.onError) {
      this.props.onError(error, info);
    } else {
```

```
    console.error(error, info);
  }
 }

 render() {
  if (this.state.hasError) {
   return this.props.fallback || <div>Something went wrong.</div>;
  }
  return this.props.children;
 }
}

export default ErrorBoundary;
```

Usage example:

```
import ErrorBoundary from './ErrorBoundary';
import MyComponent from './MyComponent';

function App() {
 return (
  <ErrorBoundaryfallback={<div>Oops—an error occurred.</div>}
   onError={(error, info) => {
    // send to logging service
    console.log('Logging error:', error, info);
   }}
  >
   <MyComponent />
  </ErrorBoundary>
 );
}
```

- ErrorBoundary must be a class component.

- It resets to fallback UI when a child throws.
- You can pass a `fallback` prop (any JSX) and an `onError` callback.
- Wrap any tree under it to catch runtime errors in rendering/lifecycle.

---

18. Create a counter component using useState ?

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
      <button onClick={() => setCount(count - 1)} disabled={count === 0}>
        Decrement
      </button>
      <button onClick={() => setCount(0)}>
        Reset
      </button>
    </div>
  );
}

export default Counter;
```

19. Create a counter component using useReducer ?

1. Counter Component Implementation

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

// Reducer functionfunction counterReducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'RESET':
      return { count: 0 };
    case 'SET_VALUE':
      return { count: action.payload };
    default:
      return state;
  }
}

// Initial stateconst initialState = { count: 0 };

function Counter() {
  const [state, dispatch] = useReducer(counterReducer, initialState);

  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1>Counter: {state.count}</h1>

      <div>
        <button onClick={() => dispatch({ type: 'DECREMENT' })}>
```

```
        Decrement -
      </button>

      <button
        onClick={() => dispatch({ type: 'RESET' })}style={{ margin: '0 10px' }}>
        Reset
      </button>

      <button onClick={() => dispatch({ type: 'INCREMENT' })}>
        Increment +
      </button>
    </div>

    <div style={{ marginTop: '20px' }}>
      <input
        type="number"value={state.count}onChange={(e) =>
          dispatch({
            type: 'SET_VALUE',
            payload: Number(e.target.value) || 0
          })
        }style={{ padding: '5px', width: '60px' }}/>
    </div>
  </div>);
}

export default Counter;
```

2. Key Features

1. **Reducer Function**:
   - Handles state transitions based on action types
   - Pure function that returns new state
   - Handles increment, decrement, reset, and direct value setting
2. **Action Types**:

- `INCREMENT`: Increases count by 1
- `DECREMENT`: Decreases count by 1
- `RESET`: Returns count to 0
- `SET_VALUE`: Sets count to a specific value from input

3. **useReducer Hook**:
   - `const [state, dispatch] = useReducer(counterReducer, initialState);`
   - Provides current state and dispatch function to send actions
4. **UI Elements**:
   - Buttons for increment/decrement/reset
   - Input field for direct value modification
   - Real-time display of current count

3. How to Use:

1. Create a new file `Counter.js` with the above code
2. Import and use it in your main App component:

```
// App.jsimport Counter from './Counter';

function App() {
  return (
    <div className="App">
      <Counter />
    </div>);
}
```

---

20. How to call a method on every rerender of a component ?

To call a method **on every re-render** of a React component, you use the `useEffect` hook **without a dependency array**.

✅ Syntax:

```
useEffect(() => {
  // This runs after every render (initial + re-renders)
  yourMethod();
});
```

🔧 Example:

```
import React, { useState, useEffect } from 'react';

function ReRenderExample() {
  const [count, setCount] = useState(0);

  const logMessage = () => {
    console.log('Component rendered. Current count:', count);
  };

  useEffect(() => {
    logMessage();
  }); // ⚠️ No dependency array → runs after every render

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

```
export default ReRenderExample;
```

📌 Explanation:

- `useEffect()` without a dependency array runs **after every render**.
- Each time the component state or props change, it re-renders → triggering the `useEffect`.

⚠️ Caution:

Avoid expensive logic inside this `useEffect` since it runs frequently. Use dependencies wisely for performance.

---

21. Create a pure component in reactjs ?

🔷 What is a Pure Component?

A **Pure Component** in React is a component that **doesn't re-render** unless its `props` or `state` **actually change**.

It implements `shouldComponentUpdate()` with a **shallow comparison** of `props` and `state` out of the box.

🔷 When to use?

Use it when:

- You want to **optimize performance**.
- You know the component re-renders unnecessarily.
- Props/state are **immutable** and don't change deeply.

✅ Example Using `React.PureComponent`

```
import React from 'react';
```

```
class Counter extends React.PureComponent {
  render() {
    console.log('Counter rendered');
    return <h1>Count: {this.props.count}</h1>;
  }
}

export default Counter;
```

✅ Usage:

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

function App() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    // Updating with the same value won't re-render <Counter />
    setCount(prev => prev);
  };

  return (
    <div>
      <Counter count={count} />
      <button onClick={handleClick}>Update Count</button>
    </div>
  );
}

export default App;
```

✅ Functional Component Alternative

In **functional components**, you use `React.memo()` for the same purpose:

```
import React from 'react';

const Counter = React.memo(({ count }) => {
  console.log('Counter rendered');
  return <h1>Count: {count}</h1>;
});

export default Counter;
```

---

22. Create a controlled and uncontrolled component in react ?

In React, **controlled** and **uncontrolled** components refer to how form data is handled inside the component. Let's explore both with code examples and clear explanations.

✅ 1. Controlled Component

A **controlled component** is one where **React state controls the input value**. You use `useState` or any state management to store and update the input value.

Example:

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

function ControlledInput() {
  const [name, setName] = useState('');

  const handleChange = (e) => {
    setName(e.target.value);
```

```
  };

  return (
    <div>
      <h3>Controlled Component</h3>
      <input type="text" value={name} onChange={handleChange} />
      <p>Entered Name: {name}</p>
    </div>
  );
}
```

✅ Key Points:

- Input value is stored in React state.
- `onChange` updates the state.
- React is always in control of the input's value.

❌ 2. Uncontrolled Component

An **uncontrolled component** is one where **the DOM controls the input value**, and you access it using a `ref`.

🔧 Example:

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

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

  const handleSubmit = () => {
    alert('Entered Name: ' + inputRef.current.value);
  };

  return (
```

```
    <div>
      <h3>Uncontrolled Component</h3>
      <input type="text" ref={inputRef} />
      <button onClick={handleSubmit}>Show Value</button>
    </div>
  );
}
```

✅ Key Points:

- No React state used to track input value.
- You directly interact with the DOM via `ref`.
- Useful for simple scenarios or performance-sensitive operations.

📌 Summary

Feature	Controlled Component	Uncontrolled Component
Input value managed by	React state	DOM (via `ref`)
Use case	Complex forms, validation	Simple inputs, quick forms
Access value	`useState` + `value` prop	`useRef().current.value`
React control	Full control	Minimal control

---

23. Create a custom hook using reactjs ?

A **custom hook** in React is a JavaScript function whose name starts with `use` and **lets you reuse logic across components**. Instead of repeating code (e.g., `useState`, `useEffect` setups) in multiple components, you can extract that logic into a custom hook.

Use Case: `useWindowWidth` (Get the current window width)

Step 1: Create the Custom Hook

```
// useWindowWidth.js
import { useState, useEffect } from 'react';

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);

    window.addEventListener('resize', handleResize);

    // Clean up on unmount
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  return width;
}

export default useWindowWidth;
```

🔧 Step 2: Use it in a Component

```
import React from 'react';
import useWindowWidth from './useWindowWidth';

function WindowWidthComponent() {
  const width = useWindowWidth();

  return (
    <div>
      <h2>Custom Hook Example</h2>
```

```
      <p>Current window width: {width}px</p>
    </div>
  );
}
```

✅ Explanation

- `useWindowWidth` is a **custom hook** that:
  - Uses `useState` to track the window width.
  - Uses `useEffect` to listen to the resize event.
  - Cleans up the event listener on unmount.
- This hook can now be **reused in any component** without duplicating logic.

💡 When to Use Custom Hooks

- Fetching data (`useFetch`)
- Managing timers (`useTimeout`, `useInterval`)
- Reusable state behavior (e.g., form handlers, toggle logic)
- Subscribing to global events (scroll, resize, visibility)

---

24. Give an emaple of optimization using useMemo ?

✅ Example of Optimization using `useMemo` in React

Let's say you have a component that performs a heavy calculation every time it renders. This is inefficient **if the inputs to the calculation haven't changed**.

Code Example

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

```
function ExpensiveComponent() {
  const [count, setCount] = useState(0);
  const [input, setInput] = useState('');

  // Heavy calculation
  const expensiveValue = useMemo(() => {
    console.log('Calculating expensive value...');
    let total = 0;
    for (let i = 0; i < 100000000; i++) {
      total += i;
    }
    return total;
  }, []); // Only runs once, on initial render

  return (
    <div>
      <h2>Expensive Value: {expensiveValue}</h2>
      <inputtype="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Type something"
      />
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
    </div>
  );
}

export default ExpensiveComponent;
```

🔍 Explanation

- `useMemo(() => {/* calculation */}, [dependencies])` caches the result of a computation.
- In the example, the expensive loop is **only run once** because the dependency array is empty `[]`.

- Without `useMemo`, this loop would execute **on every re-render**, even if the result doesn't need to change.

💡 When to Use `useMemo`

Use it to **optimize performance** when:

- You have a **heavy computation**.
- The computation depends on certain variables.
- You want to **avoid recalculating** when those variables haven't changed.

---

25. How to Share data between components using context api ?

The **Context API** allows you to create a global state or shared values that can be accessed by any component in the tree, **without passing props manually** at every level.

✅ Use Case Example

We want to share a `user` object between `App`, `Navbar`, and `UserProfile`.

Step-by-Step Implementation

1. Create a Context

```
// UserContext.js
import { createContext } from 'react';

export const UserContext = createContext(null);
```

---

2. Create a Provider Component

```
// UserProvider.js
import React, { useState } from 'react';
import { UserContext } from './UserContext';

const UserProvider = ({ children }) => {
  const [user, setUser] = useState({ name: "Saikrishna", email: "sai@email.co
m" });

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

export default UserProvider;
```

3. Wrap your app with the Provider

```
// App.js
import React from 'react';
import UserProvider from './UserProvider';
import Navbar from './Navbar';
import UserProfile from './UserProfile';

function App() {
  return (
    <UserProvider>
      <Navbar />
      <UserProfile />
    </UserProvider>
  );
}
```

```
export default App;
```

4. Consume the Context in Any Component

```
// Navbar.js
import React, { useContext } from 'react';
import { UserContext } from './UserContext';

const Navbar = () => {
  const { user } = useContext(UserContext);
  return <div>Welcome, {user.name}!</div>;
};

export default Navbar;
```

```
// UserProfile.js
import React, { useContext } from 'react';
import { UserContext } from './UserContext';

const UserProfile = () => {
  const { user, setUser } = useContext(UserContext);

  return (
    <div>
      <h2>User Profile</h2>
      <p>Email: {user.email}</p>
      <button onClick={() => setUser({ ...user, name: 'Krishna' })}>
        Change Name
      </button>
    </div>
```

```
  );
};

export default UserProfile;
```

Summary:

Create a context using `createContext()`

Wrap your app with a `Provider`

Use `useContext()` to consume the shared data in any component

---

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

In React, the `useEffect` hook in **functional components** replaces several **lifecycle methods** from **class components**. Here's a clear breakdown:

🔄 Class Component Lifecycle Methods vs useEffect

Class Component Lifecycle Method	Equivalent with `useEffect`	Purpose
`componentDidMount()`	`useEffect(() => { ... }, [])`	Runs **once after initial render** (mount)
`componentDidUpdate()`	`useEffect(() => { ... }, [dependency])`	Runs **after re-render** when **dependency changes**
`componentWillUnmount()`	`useEffect(() => { return () => { ... } }, [])`	Runs cleanup code **on unmount**
`componentDidMount` + `componentDidUpdate`	`useEffect(() => { ... })` (no dependency array)	Runs **after every render** (not recommended often)

✅ Examples for Better Understanding

1. `componentDidMount`

```
// Class
componentDidMount() {
  console.log('Component mounted');
}

// Functional
useEffect(() => {
  console.log('Component mounted');
}, []);
```

2. componentDidUpdate

```
// Class
componentDidUpdate(prevProps, prevState) {
  if (prevProps.count !== this.props.count) {
    console.log('Count updated');
  }
}

// Functional
useEffect(() => {
  console.log('Count updated');
}, [count]);
```

3. componentWillUnmount

```
// Class
componentWillUnmount() {
  console.log('Cleanup before unmount');
```

```
}

// Functional
useEffect(() => {
  return () => {
    console.log('Cleanup before unmount');
  };
}, []);
```

💡 Summary

The `useEffect` hook in functional components handles:

- Mounting
- Updating
- Unmounting

All in one unified API — depending on how you use the **dependency array** and **cleanup function**.

---

27. Create a popup using portal ?

✅ 1. Create the `Popup` component using portal

```
// Popup.js
import React from 'react';
import ReactDOM from 'react-dom';

const Popup = ({ children, onClose }) => {
  return ReactDOM.createPortal(
    <div style={overlayStyle}>
      <div style={popupStyle}>
        <button onClick={onClose} style={closeBtn}>X</button>
```

```
        {children}
      </div>
    </div>,
    document.getElementById('popup-root')
  );
};

const overlayStyle = {
  position: 'fixed',
  top: 0, left: 0, right: 0, bottom: 0,
  backgroundColor: 'rgba(0,0,0,0.5)',
  display: 'flex', alignItems: 'center', justifyContent: 'center',
};

const popupStyle = {
  background: '#fff',
  padding: '20px',
  borderRadius: '10px',
  minWidth: '300px',
  position: 'relative'
};

const closeBtn = {
  position: 'absolute',
  top: '10px',
  right: '10px',
  border: 'none',
  background: 'transparent',
  fontSize: '18px',
  cursor: 'pointer'
};

export default Popup;
```

✅ 2. Use the `Popup` in your App

```
// App.js
import React, { useState } from 'react';
import Popup from './Popup';

const App = () => {
  const [showPopup, setShowPopup] = useState(false);

  return (
    <div>
      <h1>React Portal Popup</h1>
      <button onClick={() => setShowPopup(true)}>Open Popup</button>

      {showPopup && (
        <Popup onClose={() => setShowPopup(false)}>
          <h2>This is a Portal Popup!</h2>
          <p>You can close it by clicking the X button.</p>
        </Popup>
      )}
    </div>
  );
};

export default App;
```

✅ 3. Add a `div` in `public/index.html` for the portal

```
<!-- index.html -->
<body>
  <div id="root"></div>
  <div id="popup-root"></div> <!-- required for ReactDOM.createPortal -->
</body>
```

28. How to call a method immediately after state is updated or after component is rerendered in reactjs

✅ 1. Call a method immediately after state is updated

Use `useEffect` and pass the specific state variable as a dependency.

```
jsx
CopyEdit
import React, { useState, useEffect } from 'react';

function ExampleComponent() {
  const [count, setCount] = useState(0);

  // Runs after 'count' is updated and the component is re-rendered
  useEffect(() => {
    console.log('Count updated:', count);
    yourMethod();
  }, [count]); // 👈 Dependency

  const yourMethod = () => {
    console.log("Method called after state update");
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(prev => prev + 1)}>Increment</button>
    </div>
  );
```

```
}
```

✅ 2. Call a method after *every* re-render

Pass **no dependency array** to `useEffect`.

```
jsx
CopyEdit
useEffect(() => {
  console.log("Runs after every render");
  yourMethod();
});
```

⚠️ Note:

- React state updates are **asynchronous**, so calling a function **immediately after** `setState` inside the same block **won't reflect the updated state**.
- Always rely on `useEffect` for post-update logic.

---

29. How to Force a component to rerender with out using useState in react ?

To force a React component to re-render **without using** `useState`, you can use the `useReducer` or `useRef` + `forceUpdate` trick.

✅ Option 1: `useReducer` (Cleanest Way)

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

const forceUpdateReducer = (x) => x + 1;
```

```
function MyComponent() {
  const [, forceUpdate] = useReducer(forceUpdateReducer, 0);

  return (
    <div>
      <p>Component content</p>
      <button onClick={forceUpdate}>Force Rerender</button>
    </div>
  );
}
```

✅ Option 2: Using `useRef` + Manual `forceUpdate`

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

function MyComponent() {
  const renderCount = useRef(0);
  const [, forceRerender] = React.useReducer((x) => x + 1, 0);

  const handleClick = () => {
    renderCount.current++;
    forceRerender();
  };

  return (
    <div>
      <p>Render Count: {renderCount.current}</p>
      <button onClick={handleClick}>Force Rerender</button>
    </div>
  );
}
```

✅ Option 3: Using `this.forceUpdate()` in Class Components

```
class MyComponent extends React.Component {
  force = () => {
    this.forceUpdate();
  };

  render() {
    return (
      <div>
        <p>Class Component</p>
        <button onClick={this.force}>Force Rerender</button>
      </div>
    );
  }
}
```

⚠️ Avoid `forceUpdate` tricks in production unless necessary.

Prefer `useState` / `useReducer` for state-driven re-renders. Let me know your use case and I can recommend the cleanest solution.

---

30. How to rerender a component on value change in react ?

To re-render a React component when a value changes, the most common and recommended way is to use `useState` or `useEffect` **with props/state**.

✅ 1. Using `useState` (most common)

The component re-renders **automatically** when state changes.

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

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count is: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Every time `setCount` is called, React re-renders the component with the updated state.

✅ 2. Using `useEffect` to respond to value changes (props or state)

```
import React, { useEffect, useState } from 'react';

function Watcher({ value }) {
  useEffect(() => {
    console.log('Value changed:', value);
    // This will run every time `value` changes
  }, [value]);

  return <div>Value is: {value}</div>;
}
```

If `value` is updated in a parent component, `Watcher` will re-render automatically.

✅ 3. Using props — re-render happens automatically if props change

```
function Parent() {
```

```
  const [name, setName] = useState("John");

  return (
    <><Child name={name} />
      <button onClick={() => setName("Doe")}>Change Name</button>
    </>
  );
}

function Child({ name }) {
  return <div>Hello, {name}</div>;
}
```

When `setName` updates the `name`, `Child` component re-renders with the new prop.

🧠 Summary:

Trigger	Causes Re-render?			
`useState`	✅ Yes			
`useReducer`	✅ Yes			
Changing props	✅ Yes			
`useRef`	❌ No			
`useEffect`	⚠️ Used to react to changes, not trigger re-render itself			

31. Give an example of optimization using usecallbacks in react ?

✅ Scenario

Suppose you have a **parent component** that renders a **child component**. The child receives a function as a prop. If the parent re-renders, the function is re-created, causing the child to re-render unnecessarily.

`useCallback` helps **memoize** the function so that its reference doesn't change unless its dependencies change.

🔧 Example

```
import React, { useState, useCallback } from 'react';

const Child = React.memo(({ handleClick }) => {
  console.log('Child rendered');
  return <button onClick={handleClick}>Click Me</button>;
});

const Parent = () => {
  const [count, setCount] = useState(0);
  const [otherState, setOtherState] = useState(false);

  // Without useCallback, this function will be recreated on every render
  const handleClick = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <Child handleClick={handleClick} />
      <button onClick={() => setOtherState(!otherState)}>Toggle Other State</b
utton>
    </div>
  );
};
```

```
export default Parent;
```

🧠 Explanation

- **Without** `useCallback`: The `handleClick` function gets re-created every time `Parent` renders. So `Child` sees a new `handleClick` prop and re-renders.
- **With** `useCallback`: The function is memoized, so `Child` won't re-render unless the function actually changes (which only happens if its dependencies change).

✅ Benefit

Using `useCallback` avoids **unnecessary re-renders of child components**, especially when:

- The child is wrapped in `React.memo()`.
- The function doesn't need to change on every render.
- Performance matters due to many children or heavy renders.

---

32. How to call a method when component is rendered for the first time in react ?

To **call a method when a component is rendered for the first time** in React (i.e., on component mount), you should use the `useEffect` hook with an **empty dependency array** `[]`.

✅ Syntax:

```
useEffect(() => {
  // code here runs only once when the component mounts
}, []);
```

✅ Example:

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

function MyComponent() {
  useEffect(() => {
    console.log('Component mounted');
    fetchData(); // method call
  }, []); // empty array ensures it runs only once

  const fetchData = () => {
    // Simulate API call
    console.log('Fetching data...');
  };

  return <div>Hello World</div>;
}
```

✅ Explanation:

- `useEffect` runs **after the component is mounted**.
- Passing `[]` as the second argument means it **won't re-run** unless the component is unmounted and re-mounted.
- This is equivalent to `componentDidMount()` in class components.

✅ Class Component Equivalent:

```
class MyComponent extends React.Component {
  componentDidMount() {
    console.log('Component mounted');
    this.fetchData();
  }

  fetchData = () => {
```

```
    console.log('Fetching data...');
  };

  render() {
    return <div>Hello World</div>;
  }
}
```

---

33. How to change styles based on condition in react ?

✅ 1. Using Inline Styles with Ternary Operator

You can directly apply conditional styles using a ternary operator:

```
function StatusBox({ isActive }) {
  return (
    <divstyle={{
        backgroundColor: isActive ? 'green' : 'gray',
        color: 'white',
        padding: '10px',
      }}
    >
      {isActive ? 'Active' : 'Inactive'}
    </div>
  );
}
```

✅ 2. Conditional Class Names

You can apply different CSS classes conditionally:

```
function Button({ isPrimary }) {
  return (
    <button className={isPrimary ? 'btn-primary' : 'btn-secondary'}>
      Click Me
    </button>
  );
}
```

CSS:

```
.btn-primary {
  background-color: blue;
  color: white;
}

.btn-secondary {
  background-color: gray;
  color: black;
}
```

---

34. How do you access the dom element ?

In React, you can **access DOM elements** using **Refs**.

✅ Method: Using `useRef` (Functional Components)

`useRef` allows you to directly access a DOM node created by JSX.

Example:

```
import React, { useRef, useEffect } from "react";

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

  useEffect(() => {
    // Access the DOM element and focus it
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} placeholder="Focus on mount" />;
}
```

✅ Method: `createRef` (Class Components)

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myDivRef = React.createRef();
  }

  componentDidMount() {
    // Access DOM element
    this.myDivRef.current.style.backgroundColor = 'yellow';
  }

  render() {
    return <div ref={this.myDivRef}>Hello</div>;
  }
}
```

🔎 When to use it?

- To focus an input field
- To measure DOM dimensions
- To scroll an element into view
- To interact with third-party libraries (charts, modals, etc.)

⚠️ Best Practice

Avoid excessive use of DOM access in React. Prefer **React state and props** for controlling behavior and appearance.

---

35. Perform type checking using prop-types in reactjs ?

In ReactJS, you can use the `prop-types` package to perform **type checking** on component props. It helps ensure that components are used with the correct data types and provides warnings in the console during development if types don't match.

✅ Steps to Use `prop-types`:

1. **Install `prop-types`:**

```
npm install prop-types
```

1. **Import and use it in your component:**

🔷 Example: Functional Component

```
import React from 'react';
import PropTypes from 'prop-types';
```

```
const UserProfile = ({ name, age, isAdmin }) => {
  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
      {isAdmin && <p>Admin Access</p>}
    </div>
  );
};

// Type checking with PropTypes
UserProfile.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number,
  isAdmin: PropTypes.bool
};

export default UserProfile;
```

🔷 Example: Class Component

```
import React from 'react';
import PropTypes from 'prop-types';

class Product extends React.Component {
  render() {
    return (
      <div>
        <h3>{this.props.title}</h3>
        <p>Price: ${this.props.price}</p>
      </div>
    );
  }
}
```

```
Product.propTypes = {
  title: PropTypes.string.isRequired,
  price: PropTypes.number.isRequired
};

export default Product;
```

🔶 Common Prop Types

Type	Usage
`PropTypes.string`	String
`PropTypes.number`	Number
`PropTypes.bool`	Boolean
`PropTypes.array`	Array
`PropTypes.object`	Object
`PropTypes.func`	Function
`PropTypes.node`	Anything that can be rendered
`PropTypes.element`	React element
`PropTypes.symbol`	ES6 Symbol

🔶 Advanced Usage

```
PropTypes.arrayOf(PropTypes.string)
PropTypes.objectOf(PropTypes.number)
PropTypes.oneOf(['small', 'medium', 'large'])
PropTypes.oneOfType([PropTypes.string, PropTypes.number])
PropTypes.shape({
  id: PropTypes.number.isRequired,
  name: PropTypes.string
})
```

36. How to Implement a Theme Switcher Using Context API

Step 1: Create Theme Context

We'll create a context that holds the current theme and a function to toggle it.

```
// ThemeContext.js
import React, { createContext, useState, useContext } from 'react';

const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  // Theme state: 'light' or 'dark'
  const [theme, setTheme] = useState('light');

  // Toggle between light and dark
  const toggleTheme = () => {
    setTheme(prev => (prev === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

// Custom hook for easier usage
export const useTheme = () => useContext(ThemeContext);
```

Step 2: Wrap Your App with ThemeProvider

Wrap your main app component with the `ThemeProvider` so the theme context is available throughout the app.

```
// index.js or App.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { ThemeProvider } from './ThemeContext';

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

Step 3: Create ThemeSwitcher Component

This component will read the current theme and toggle it on button click.

```
// ThemeSwitcher.js
import React from 'react';
import { useTheme } from './ThemeContext';

const ThemeSwitcher = () => {
  const { theme, toggleTheme } = useTheme();

  return (
    <button onClick={toggleTheme}>
      Switch to {theme === 'light' ? 'Dark' : 'Light'} Mode
    </button>
  );
};
```

```
export default ThemeSwitcher;
```

Step 4: Use Theme in Your App

You can now consume the theme value anywhere to apply styles conditionally.

```
// App.js
import React from 'react';
import { useTheme } from './ThemeContext';
import ThemeSwitcher from './ThemeSwitcher';

const App = () => {
  const { theme } = useTheme();

  const appStyle = {
    height: '100vh',
    display: 'flex',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: theme === 'light' ? '#fff' : '#222',
    color: theme === 'light' ? '#222' : '#fff',
    flexDirection: 'column',
    gap: '20px',
  };

  return (
    <div style={appStyle}>
      <h1>{theme.charAt(0).toUpperCase() + theme.slice(1)} Mode</h1>
      <ThemeSwitcher />
    </div>
  );
};

export default App;
```

Explanation

- **Context API** creates a global state (`ThemeContext`) that holds the current theme and the toggle function.
- **ThemeProvider** wraps the app and provides access to theme state everywhere.
- **useTheme** custom hook simplifies accessing context values.
- **ThemeSwitcher** component reads current theme and toggles it.
- **App** component uses the theme value to apply styles dynamically.

This keeps theme state centralized and allows any component in the tree to read or update the theme easily.

---

37. How to update Document Title on Mount in a React Component

To update the **document title** when a React component mounts, you typically use the `useEffect` hook in a functional component.

Why use `useEffect`?

- React components render many times during their lifecycle.
- We want to update the document title **only once when the component mounts** (or when specific values change).
- `useEffect` with an empty dependency array runs only once after the first render (mount).

Example: Update document title on mount

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

function MyComponent() {
```

```
  useEffect(() => {
    document.title = "My New Page Title";
  }, []);  // empty dependency array means run once on mount

  return <div>Hello, this is my component!</div>;
}

export default MyComponent;
```

Explanation:

- `useEffect(() => { ... }, [])` runs the callback function once after the component is mounted.
- Inside the callback, we set `document.title` to the desired string.
- The empty array `[]` ensures this effect does **not** run on subsequent re-renders.

In class components

If you use class components, you do this in `componentDidMount`:

```
class MyComponent extends React.Component {
  componentDidMount() {
    document.title = "My New Page Title";
  }

  render() {
    return <div>Hello from class component</div>;
  }
}
```

Summary:

- Use `useEffect` with empty dependencies in functional components.
- Use `componentDidMount` in class components.

- This updates the browser tab's title when the component mounts.

If you want to update the title dynamically based on props or state, add those dependencies in the array. But for just updating once on mount, keep it empty.

---

38. How to create a Higher-Order Component (HOC) to Log Props in reactjs ?

✅ What is a Higher-Order Component (HOC) in React?

A **Higher-Order Component (HOC)** is a **function** that takes a **component as input** and returns a **new component** with extended or enhanced functionality.

Example use case: You want to log props every time a component renders – without modifying the component directly. A HOC is the perfect solution.

✅ How to Create a HOC to Log Props in React

Here's a step-by-step explanation with code:

🔧 Step 1: Create the HOC

```
// withPropsLogger.js
import React, { useEffect } from 'react';

const withPropsLogger = (WrappedComponent) => {
  return function PropsLoggerWrapper(props) {
    useEffect(() => {
      console.log(`[Props Logger] ${WrappedComponent.name} props:`, props);
    }, [props]); // log every time props change

    return <WrappedComponent {...props} />;
  };
};
```

```
export default withPropsLogger;
```

✅ What this does:

- Takes a `WrappedComponent` as input.
- Logs the props to the console whenever they change.
- Renders the `WrappedComponent` with all received props.

🔧 Step 2: Use the HOC

```
// MyComponent.js
import React from 'react';

const MyComponent = ({ name }) => {
  return <div>Hello, {name}</div>;
};

export default MyComponent;
```

```
// App.js
import React from 'react';
import MyComponent from './MyComponent';
import withPropsLogger from './withPropsLogger';

const MyComponentWithLogging = withPropsLogger(MyComponent);

function App() {
  return <MyComponentWithLogging name="Saikrishna" />;
}

export default App;
```

39. How will you Manage Environment-Specific Configurations in Reactjs in your project ?

1. Using `.env` Files with Create React App (CRA)

CRA supports environment variables through `.env` files.

- `.env` (default, for all envs)
- `.env.development`
- `.env.production`
- `.env.test`

Important: All variables must start with `REACT_APP_` to be accessible in your React app.

Example Setup

Create these files in your project root:

.env.development

```
REACT_APP_API_URL=https://dev.api.example.com
REACT_APP_FEATURE_FLAG=true
```

.env.production

```
REACT_APP_API_URL=https://api.example.com
REACT_APP_FEATURE_FLAG=false
```

2. Accessing Variables in React Code

```
function App() {
  const apiUrl = process.env.REACT_APP_API_URL;
  const featureFlag = process.env.REACT_APP_FEATURE_FLAG === 'true';
```

```
  return (
    <div>
      <h1>Environment: {process.env.NODE_ENV}</h1>
      <p>API URL: {apiUrl}</p>
      <p>Feature Enabled: {featureFlag ? 'Yes' : 'No'}</p>
    </div>
  );
}

export default App;
```

- `process.env.NODE_ENV` will be `'development'` or `'production'` depending on how you run/build your app.
- `process.env.REACT_APP_API_URL` will be replaced at build time with the matching `.env` file value.

3. Running Your App

- `npm start` will use `.env.development`
- `npm run build` will use `.env.production`

4. Custom Configuration Object (Optional)

If you want more control, create a config file that switches based on `NODE_ENV`:

```
// src/config.js
const devConfig = {
  apiUrl: "https://dev.api.example.com",
  featureFlag: true,
};

const prodConfig = {
  apiUrl: "https://api.example.com",
  featureFlag: false,
};
```

```
const config = process.env.NODE_ENV === "production" ? prodConfig : devC
onfig;

export default config;
```

Use it in your app:

```
import config from './config';

function App() {
  return (
    <div>
      <h1>API URL: {config.apiUrl}</h1>
      <p>Feature Enabled: {config.featureFlag ? 'Yes' : 'No'}</p>
    </div>
  );
}
```

Summary

- Use `.env` files with variables prefixed by `REACT_APP_`.
- Access them in your React code via `process.env.REACT_APP_*`.
- CRA automatically picks the right `.env` based on environment.
- For more flexibility, create your own config file and export based on `NODE_ENV`.

---

40. How to create a Custom Hook for Data Fetching with Loading and Error States

Creating a custom React hook for data fetching with loading and error states is a great way to encapsulate and reuse fetch logic cleanly across your app.

Why a Custom Hook for Data Fetching?

- Reuse the same fetch logic in multiple components
- Keep components cleaner and focused on UI
- Handle loading, error, and data states in one place

How to Create It — Step by Step

1. Define the Hook

Create a function called `useFetch` which takes a URL (or any fetch parameter).

It manages:

- `data` — the fetched result
- `loading` — boolean while fetching
- `error` — error message if fetch fails

2. Use `useState` for state management and `useEffect` to trigger fetch

Example Code

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    if (!url) return;

    setLoading(true);
    setError(null);

    fetch(url)
```

```
      .then((response) => {
        if (!response.ok) {
          throw new Error("Network response was not ok");
        }
        return response.json();
      })
      .then((json) => {
        setData(json);
        setLoading(false);
      })
      .catch((err) => {
        setError(err.message || "Something went wrong");
        setLoading(false);
      });
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

3. Using the Hook in a Component

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

function UserList() {
  const { data, loading, error } = useFetch("https://jsonplaceholder.typicode.com/users");

  if (loading) return <p>Loading users...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <ul>
```

```
      {data.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

How it Works

- When the component mounts or the URL changes, `useEffect` triggers the fetch.
- While fetching, `loading` is `true`.
- If the fetch succeeds, `data` is set and `loading` is `false`.
- If it fails, `error` is set and `loading` is `false`.
- The component using the hook can respond to these states accordingly.

---

41. Create a component to upload the file in reactjs ?

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

function FileUpload() {
  const [file, setFile] = useState(null);
  const [status, setStatus] = useState("");

  const onFileChange = (e) => {
    setFile(e.target.files[0]);
    setStatus("");
```

```
};

const onUpload = async () => {
 if (!file) {
  setStatus("Please select a file first");
  return;
 }

 const formData = new FormData();
 formData.append("file", file);

 setStatus("Uploading...");

 try {
  const response = await fetch("https://your-api-endpoint.com/upload", {
   method: "POST",
   body: formData,
  });

  if (!response.ok) {
   throw new Error(`Upload failed with status ${response.status}`);
  }

  const result = await response.json();
  setStatus(`Upload successful: ${result.message || file.name}`);
 } catch (error) {
  setStatus(`Upload error: ${error.message}`);
 }
};

return (
 <div>
  <input type="file" onChange={onFileChange} />
  <button onClick={onUpload}>Upload</button>
  {status && <p>{status}</p>}
 </div>
```

```
  );
}

export default FileUpload;
```

Replace `"https://your-api-endpoint.com/upload"` with your actual upload URL.

This sends the selected file as multipart/form-data via POST to your API.

---

42. How to cancel the api call when the timeout occurs using abort controller ?

What is AbortController?

- `AbortController` is a browser API that lets you abort (cancel) a fetch request.
- You create an instance of `AbortController`.
- Pass its `signal` property to the fetch request.
- Call `abort()` on the controller to cancel the fetch.

How to use AbortController with timeout in React?

You combine `AbortController` with `setTimeout` to cancel the fetch if it takes too long.

Example code and explanation

```
import React, { useEffect, useState } from 'react';

function FetchWithTimeout() {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
```

```
  const controller = new AbortController();  // Create AbortController
  const signal = controller.signal;

  const timeoutId = setTimeout(() => {
    controller.abort(); // Cancel the fetch after timeout
  }, 5000); // 5000ms = 5 seconds timeout

  setLoading(true);
  fetch('https://jsonplaceholder.typicode.com/posts/1', { signal })
    .then((response) => {
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      return response.json();
    })
    .then((json) => {
      setData(json);
      setError(null);
    })
    .catch((err) => {
      if (err.name === 'AbortError') {
        setError('Fetch request was aborted due to timeout');
      } else {
        setError(err.message);
      }
    })
    .finally(() => {
      setLoading(false);
      clearTimeout(timeoutId); // Clear the timeout if fetch finished before time
out
    });

  // Cleanup if component unmounts before fetch completes
  return () => {
    clearTimeout(timeoutId);
    controller.abort();
```

```
    };
  }, []);

  return (
    <div>
      {loading && <p>Loading...</p>}
      {error && <p style={{ color: 'red' }}>Error: {error}</p>}
      {data && (
        <div>
          <h3>{data.title}</h3>
          <p>{data.body}</p>
        </div>
      )}
    </div>
  );
}

export default FetchWithTimeout;
```

Explanation:

- We create an `AbortController` instance and get its `signal`.
- Pass the `signal` to `fetch` options.
- We set a `setTimeout` to call `controller.abort()` after 5 seconds.
- If the fetch completes before timeout, `clearTimeout` cancels the timeout.
- If timeout happens first, fetch is aborted, and `.catch` handles the `AbortError`.
- The cleanup function cancels the fetch and clears the timeout if the component unmounts early.

Summary

- Use `AbortController` to cancel fetch requests.
- Combine with `setTimeout` to implement timeout logic.
- Handle `AbortError` to detect when fetch is canceled.

- Always clear your timeouts to avoid leaks.

15 Angular Scenario based interview questions

1. How do you send the data from parent component to child component in angular ?

Ans. In Angular, you pass data from a parent component to a child component using `@Input()` **properties**. Here's a clear, step-by-step explanation:

a. Define an `@Input()` Property in the Child Component

- Use the `@Input()` decorator on a property in the child component to receive data from the parent.

```
// child.component.ts
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <h2>Child Component</h2>
    <p>Message from parent: {{ parentMessage }}</p>
    <p>User: {{ userData.name }} ({{ userData.age }})</p>
  `})
export class ChildComponent {
  @Input() parentMessage!: string;// Receives string
  @Input() userData!: { name: string, age: number
```

```
};// Receives object
}
```

b. Pass Data from Parent Component's Template

- Bind the parent's data to the child's `@Input()` properties using **property binding** (`[propertyName]`).

```
// parent.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <h1>Parent Component</h1>
    <app-child
      [parentMessage]="message"
      [userData]="user"
    ></app-child>
  `})
export class ParentComponent {
  message = "Hello from Parent!";// Data to send
  user = { name: 'Alice', age: 30 };// Object data}
```

---

2. How to call parent component method from child component in angular ?

In Angular's component-based architecture, parent-child communication is elegantly handled through `@Output()` decorators and `EventEmitter`. This maintains Angular's unidirectional data flow while allowing children to "notify" parents about events. Here's the beautiful flow:

1. **Child Component**: Declares an `@Output()` event emitter (like raising a flag).

2. **Parent Component**: Listens for this event and triggers its own method.
3. **Data Flow**: Child emits events with optional data payloads → Parent catches and responds.

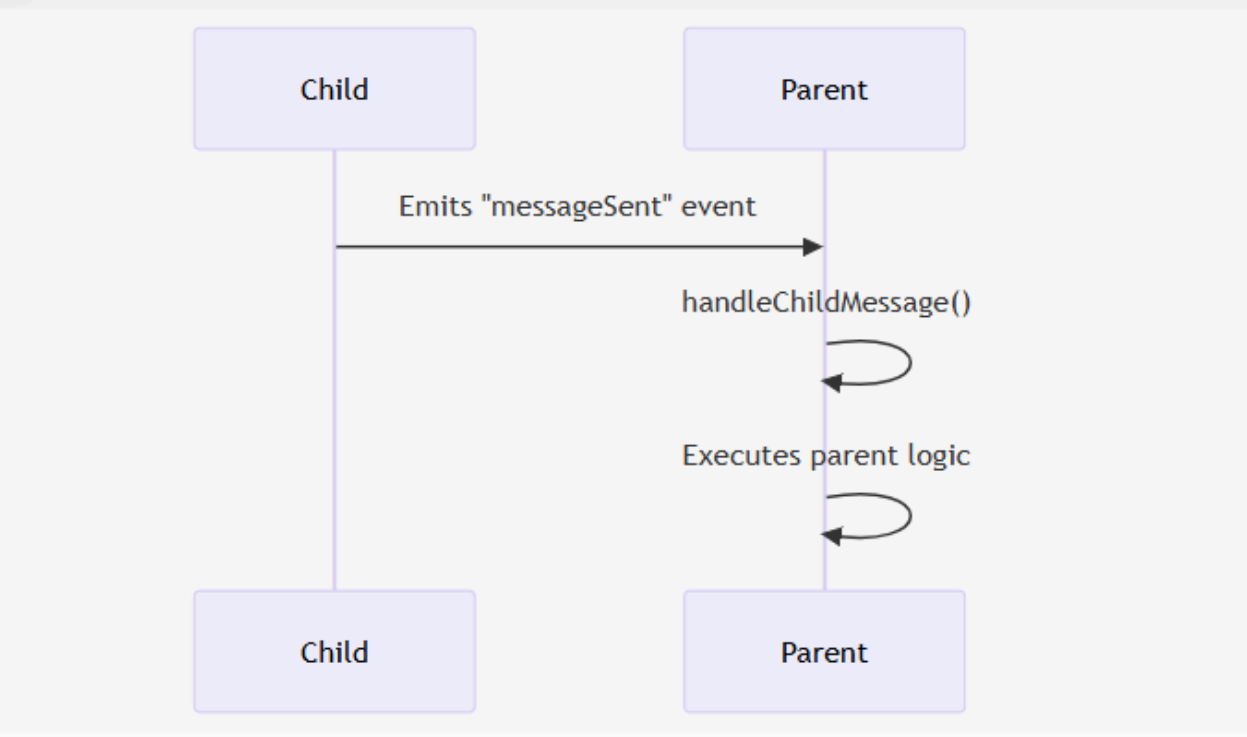
Step 1: Child Component

```
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
 selector: 'app-child',
 template: `
  <div class="child">
   <h2>🌼 Child Component</h2>
   <button (click)="sendMessageToParent()">
   Send Message
   </button>
  </div>
 `})
export class ChildComponent {
// 1. Declare output event emitter
@Output() messageSent = new EventEmitter<string>();

 sendMessageToParent() {
  const message = "Hello from child!";
// 2. Emit event with data payload
this.messageSent.emit(message);
 }
}
```

Step 2: Parent Component

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-parent',
```

```
  template: `
    <div class="parent">
      <h1>🌳 Parent Component</h1>
      <p>Child's message: {{ childMessage }}</p>

      <!-- 3. Bind to child's event -->
      <app-child (messageSent)="handleChildMessage($event)">
      </app-child>
    </div>
  `})
export class ParentComponent {
  childMessage = '';

// 4. Handle event from child
handleChildMessage(message: string) {
    this.childMessage = message;
    console.log('Parent received:', message);
// Call any parent method here!
}
}
```



3. How to bind array/array of objects to dropdown in angular ?

Binding an array or array of objects to a dropdown (`<select>`) in Angular is straightforward. Here's how you can do it clearly:

Step 1: Define Array or Array of Objects in Component

```
// component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-dropdown',
  templateUrl: './dropdown.component.html',
})
export class DropdownComponent {
```

```
  // Simple array
  fruits = ['Apple', 'Banana', 'Orange'];

  // Array of objects
  countries = [
    { id: 1, name: 'India' },
    { id: 2, name: 'USA' },
    { id: 3, name: 'Canada' },
  ];

  selectedFruit: string = '';
  selectedCountryId: number | null = null;
}
```

Step 2: Template Binding (`ngFor`)

In your HTML template, use the `*ngFor` directive to loop over the array:

For a simple array

```
<select [(ngModel)]="selectedFruit">
  <option *ngFor="let fruit of fruits" [value]="fruit">
    {{ fruit }}
  </option>
</select>

Selected Fruit: {{ selectedFruit }}
```

For an array of objects

```
<select [(ngModel)]="selectedCountryId">
```

```
  <option *ngFor="let country of countries" [value]="country.id">
    {{ country.name }}
  </option>
</select>

Selected Country ID: {{ selectedCountryId }}
```

Step 3: Import FormsModule

To enable two-way binding (`[(ngModel)]`), import the `FormsModule` in your module:

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [DropdownComponent],
  bootstrap: [DropdownComponent],
})
export class AppModule {}
```

Explanation

- `ngFor` loops through the array.
- `[value]` sets the value attribute for the option.
- `[(ngModel)]` enables two-way binding, allowing your selection to update the bound variable.

Output Example:

- Selecting a fruit like "Banana" would set `selectedFruit` to `"Banana"`.

- Selecting a country like "India" would set `selectedCountryId` to `1`.

---

4. Create a lazy loaded component in angular ?

✅ Step 1: Create a component

```
ng generate component lazy-component
```

✅ Step 2: Create a Module for Lazy Loading

Create a new module to host the component:

```
ng generate module lazy --routing
```

This creates two files:

- `lazy.module.ts`
- `lazy-routing.module.ts`

✅ Step 3: Add the Component to the Lazy Module

lazy.module.ts:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { LazyComponentComponent } from '../lazy-component/lazy-compone
nt.component';
import { LazyRoutingModule } from './lazy-routing.module';

@NgModule({
  declarations: [LazyComponentComponent],
```

```
  imports: [
    CommonModule,
    LazyRoutingModule
  ]
})
export class LazyModule { }
```

✅ Step 4: Define Route in Lazy Routing Module

lazy-routing.module.ts:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LazyComponentComponent } from '../lazy-component/lazy-compone
nt.component';

const routes: Routes = [
  { path: '', component: LazyComponentComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class LazyRoutingModule { }
```

✅ Step 5: Configure Lazy Loading in the Main Routing Module

Update your app's main routing (`app-routing.module.ts`):

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
```

```
  {
    path: 'lazy',
    loadChildren: () => import('./lazy/lazy.module').then(m => m.LazyModule)
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
```

✅ Step 6: Use the Lazy Loaded Route

In your HTML, use a router link:

```
<a routerLink="/lazy">Go to Lazy Loaded Component</a>
<router-outlet></router-outlet>
```

Explanation:

- **Lazy loading** means the `LazyModule` and its components won't load initially, only loading when the user navigates to the `/lazy` route.
- This approach optimizes performance by reducing the initial load time.

---

5. How to display data entered by the user in another textbox ?

Step-by-step approach:

1. Component HTML:

```
<input type="text" [(ngModel)]="userInput" placeholder="Enter something" />

<!-- Second textbox displays entered data -->
<input type="text" [value]="userInput" readonly /
```

2. Component TypeScript:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-input-display',
  templateUrl: './input-display.component.html'
})
export class InputDisplayComponent {
  userInput: string = '';
}
```

Explanation:

- `[(ngModel)]` provides two-way binding, automatically syncing input changes with the component's `userInput` property.
- The second input uses property binding (`[value]`) to reflect the current value of `userInput`.

Remember:

- Ensure you import `FormsModule` in your module:

```
import { FormsModule } from '@angular/forms';

@NgModule({
```

```
  imports: [FormsModule]
})
export class YourModule { }
```

This approach automatically updates the second textbox as you type in the first one.

---

6. How to conditionally render an element or text in angular ?

In Angular, you can conditionally render an element or text using built-in directives like `*ngIf`:

Using `ngIf`:

HTML:

```
<div *ngIf="showElement">
  This element is conditionally rendered.
</div>
```

Component:

```
export class ExampleComponent {
  showElement: boolean = true; // or false
}
```

Conditional text or alternate rendering:

```
<div>
  {{ showElement ? 'Element is visible!' : 'Element is hidden!' }}
</di
```

Using else block with `ngIf`:

HTML:

```
<div *ngIf="isLoggedIn; else loggedOut">
  Welcome back, user!
</div>

<ng-template #loggedOut>
  Please log in.
</ng-template
```

Component:

```
export class ExampleComponent {
  isLoggedIn: boolean = false; // toggle based on logic
}
```

These simple patterns cover most conditional rendering scenarios in Angular.

---

7. How to perform debouncing in angular ?

Method 1: Using RxJS (Recommended)

Import necessary modules:

```
typescript
CopyEdit
import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';
```

```
import { debounceTime, distinctUntilChanged } from 'rxjs/operators';
```

Component Implementation:

```
//component.ts
@Component({
  selector: 'app-debounce',
  templateUrl: './debounce.component.html',
})
export class DebounceComponent implements OnInit {

  searchControl = new FormControl('');

  ngOnInit(): void {
    this.searchControl.valueChanges.pipe(
      debounceTime(500),            // waits for 500ms pause
      distinctUntilChanged()        // triggers only if value changes
    )
    .subscribe(value => {
      this.onSearch(value);
    });
  }

  onSearch(value: string): void {
    console.log('Debounced Search:', value);
    // Perform search or API call here
  }
}
```

HTML:

```
```

```
<input type="text" [formControl]="searchControl" placeholder="Search..." />
```

Explanation:

- `debounceTime(500)`: waits 500 milliseconds after typing stops before emitting the event.
- `distinctUntilChanged()`: ensures the subscribed event triggers only if the value has changed.

Method 2: Manual Debounce with JavaScript (less common):

If RxJS isn't desired, you could create a custom debounce function manually.

component.ts

```
debounceTimer: any;

onKeyup(value: string): void {
  clearTimeout(this.debounceTimer);
  this.debounceTimer = setTimeout(() => {
    this.onSearch(value);
  }, 500); // waits 500ms
}

onSearch(value: string): void {
  console.log('Search:', value);
}
```

HTML

```
<input type="text" (keyup)="onKeyup($event.target.value)" placeholder="Sea
rch..." />
```

8. Create a component to fetch data from api in angular ?

Step-by-step Implementation

a. Import `HttpClientModule` in your App Module (`app.module.ts`):

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    HttpClientModule
  ]
})
export class AppModule { }
```

b. Create the Service (`data.service.ts`):

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class DataService {

  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) {}

  fetchData(): Observable<any[]> {
```

```
    return this.http.get<any[]>(this.apiUrl);
  }
}
```

c. Create the Component (`fetch-data.component.ts`):

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-fetch-data',
  templateUrl: './fetch-data.component.html'
})
export class FetchDataComponent implements OnInit {

  data: any[] = [];
  loading: boolean = true;
  error: string = '';

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.fetchData().subscribe({
      next: (response) => {
        this.data = response;
        this.loading = false;
      },
      error: (err) => {
        this.error = err.message || 'Error fetching data';
        this.loading = false;
      }
    });
  }
```

```
}
```

d. Create the HTML Template (`fetch-data.component.html`):

```
<div *ngIf="loading">Loading data...</div>
<div *ngIf="error" class="error">{{ error }}</div>

<ul *ngIf="!loading && !error">
  <li *ngFor="let item of data">
    <strong>{{ item.title }}</strong>
    <p>{{ item.body }}</p>
  </li>
</ul>
```

Explanation:

- **Service (** `HttpClient` **)** is the recommended way to fetch API data.
- **Loading** and **error handling** are managed clearly.
- Component fetches data on initialization using Angular's lifecycle hook `ngOnInit`.

Now your Angular component cleanly fetches and displays data from an API.

---

9. Given two dropdowns, select 2nd dropdown options based on value selected in one dropdown in angular ? (Load states based on country selected)

a. Sample Data Setup

```
// country-state.data.ts
export const countries = [
  { id: 1, name: 'India' },
  { id: 2, name: 'USA' }
];

export const states = [
  { id: 1, name: 'Telangana', countryId: 1 },
  { id: 2, name: 'Karnataka', countryId: 1 },
  { id: 3, name: 'California', countryId: 2 },
  { id: 4, name: 'Texas', countryId: 2 }
];
```

b. Component TypeScript

```
import { Component } from '@angular/core';
import { countries, states } from './country-state.data';

@Component({
  selector: 'app-country-state-dropdown',
  templateUrl: './country-state-dropdown.component.html'
})
export class CountryStateDropdownComponent {
  countries = countries;
  allStates = states;

  selectedCountryId: number = 0;
  filteredStates: any[] = [];

  onCountryChange(countryId: string) {
    const id = +countryId;
```

```
    this.filteredStates = this.allStates.filter(s => s.countryId === id);
    this.selectedCountryId = id;
  }
}
```

c. Component HTML

```
<!-- Country Dropdown -->
<select (change)="onCountryChange($event.target.value)">
  <option value="">Select Country</option>
  <option *ngFor="let country of countries" [value]="country.id">{{ country.name }}</option>
</select>

<!-- State Dropdown -->
<select [disabled]="filteredStates.length === 0">
  <option value="">Select State</option>
  <option *ngFor="let state of filteredStates" [value]="state.id">{{ state.name }}</option>
</select>
```

Optional: Use `ngModel` for two-way binding (if needed)

Add `FormsModule` to `AppModule` imports if using `[(ngModel)]`:

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule]
})
```

10. How to display dynamic html data in angular ?

To display dynamic HTML data in Angular safely (i.e., HTML content coming from a variable), you should use Angular's `[innerHTML]` binding along with **DomSanitizer** to avoid security issues like XSS.

Suppose you have some HTML content as a string in your component:

```
// component.ts
export class MyComponent {
  htmlContent: string = "<p style='color:red;'>This is dynamic HTML content</
p>";
}
```

In the template:

```
<!-- component.html -->
<div [innerHTML]="htmlContent"></div>
```

If the HTML comes from external source (safe approach):

Angular sanitizes HTML automatically when using `[innerHTML]`, but sometimes you may want to trust some HTML explicitly (e.g., you know it's safe). Use `DomSanitizer` like this:

```
import { Component } from '@angular/core';
import { DomSanitizer, SafeHtml } from '@angular/platform-browser';

@Component({
  selector: 'app-my-component',
  templateUrl: './my.component.html'
})
```

```
export class MyComponent {
  htmlContent: SafeHtml;

  constructor(private sanitizer: DomSanitizer) {
    const dirtyHtml = "<p style='color:red;'>Dynamic HTML content with <b>b
old</b> text</p>";
    this.htmlContent = this.sanitizer.bypassSecurityTrustHtml(dirtyHtml);
  }
}
```

Template:

```
<div [innerHTML]="htmlContent"></div>
```

Summary:

- Use `[innerHTML]` binding to render HTML from string.
- If the content is dynamic and from an external source, sanitize or trust it via `DomSanitizer`.
- Avoid direct string interpolation with `{{ }}` because Angular will escape HTML as plain text.

---

11. How to update an array property in an Angular component and reflect the changes in the template?

To update an array property in an Angular component and have those changes reflected in the template, follow these key points:

Basic steps:

1. **Update the array property in the component (e.g., add, remove, modify items).**
2. **Angular's change detection will automatically update the view if you mutate the array properly or replace it.**
3. **Use Angular template syntax like `ngFor` to display the array.**

Example

Component (`example.component.ts`):

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
})
export class ExampleComponent {
  items: string[] = ['Apple', 'Banana', 'Cherry'];

  addItem() {
    this.items.push('Date');
    // Alternatively, to trigger change detection safely:
    // this.items = [...this.items, 'Date'];
  }

  removeItem(index: number) {
    this.items.splice(index, 1);
  }

  updateItem(index: number, newValue: string) {
    this.items[index] = newValue;
    // If needed, reassign to new array reference for change detection:
    // this.items = [...this.items];
  }
```

```
}
```

Template (`example.component.html`):

```
<ul>
  <li *ngFor="let item of items; let i = index">
    {{ item }}
    <button (click)="removeItem(i)">Remove</button>
  </li>
</ul>

<button (click)="addItem()">Add Item</button>
```

- **Mutating the array** (e.g., `.push()`, `.splice()`) usually works fine because Angular's default change detection detects these changes in most cases.
- However, in some cases (especially with `OnPush` change detection strategy or immutability preferences), **replace the array reference** like this:

  ```
  this.items = [...this.items, 'NewItem'];
  ```

  This triggers Angular to detect a new array and update the template accordingly.
- Always **bind your template with** `ngFor` or other directives that react to data changes.

---

12. Change focus/enable/disable textbox in child component based on parent component button click ?

To control focus, enable, or disable a textbox inside a **child component** from a **parent component button click** in Angular, follow these steps:

1. **Pass a property from parent to child** via `@Input()` to control enable/disable or focus state.
2. **Use `@ViewChild` in child** to get the textbox reference for focusing.
3. **On input change, update textbox state or focus inside the child component.**

a. Child Component (child.component.ts & html)

```
// child.component.ts
import { Component, Input, OnChanges, SimpleChanges, ViewChild, ElementRef } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <input #inputBox type="text" [disabled]="isDisabled" />
  `
})
export class ChildComponent implements OnChanges {
  @Input() isDisabled: boolean = false;
  @Input() shouldFocus: boolean = false;

  @ViewChild('inputBox') inputBox!: ElementRef<HTMLInputElement>;

  ngOnChanges(changes: SimpleChanges) {
    if (changes['shouldFocus'] && this.shouldFocus) {
      // Focus input when shouldFocus becomes true
      setTimeout(() => this.inputBox.nativeElement.focus(), 0);
    }
  }
}
```

b. Parent Component (parent.component.ts & html)

```
// parent.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <button (click)="toggleTextbox()">Toggle Enable/Disable</button>
    <button (click)="focusTextbox()">Focus Textbox</button>

    <app-child
      [isDisabled]="isTextboxDisabled"
      [shouldFocus]="shouldFocusTextbox">
    </app-child>
  `
})
export class ParentComponent {
  isTextboxDisabled = false;
  shouldFocusTextbox = false;

  toggleTextbox() {
    this.isTextboxDisabled = !this.isTextboxDisabled;
  }

  focusTextbox() {
    this.shouldFocusTextbox = true;
    // Reset after focus to allow future focus toggles
    setTimeout(() => this.shouldFocusTextbox = false, 100);
  }
}
```

Explanation:

- The **parent toggles** `isTextboxDisabled` to enable/disable the textbox in the child.

- The **parent sets** `shouldFocusTextbox` **to true** to trigger focus in the child input.
- The child detects changes in inputs using `ngOnChanges` and **focuses the input programmatically**.
- Resetting `shouldFocusTextbox` to false lets you focus again on subsequent clicks.

---

13. How to display dropdown value selected by user in another textbox ?

To display the dropdown value selected by the user in another textbox in Angular, you can:

1. Bind the dropdown selection to a component property using `[(ngModel)]` or reactive forms.
2. Bind the textbox's value to the same property (or a derived one).
3. When user selects an option, the textbox updates automatically.

Simple Example Using `ngModel`

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  selectedValue: string = '';
  options = ['Apple', 'Banana', 'Cherry'];
}
```

app.component.html

```
<select [(ngModel)]="selectedValue">
  <option *ngFor="let option of options" [value]="option">{{ option }}</option
>
</select>

<br />

<input type="text" [value]="selectedValue" readonly />
```

Explanation:

- `[(ngModel)]="selectedValue"` binds the dropdown selected value two-way.
- The textbox has `[value]="selectedValue"` which updates automatically when selection changes.
- Setting `readonly` prevents user editing the textbox directly.

---

14. How do you implement a character countdown functionality for a `<textarea>` in Angular? Explain how you can use a template reference variable or `@ViewChild` to track the number of characters entered and display the remaining characters dynamically.

Here's how to implement a **character countdown** for a `<textarea>` in Angular, using both **template reference variables** and **@ViewChild** approaches to track characters entered and display remaining characters dynamically.

a. Using Template Reference Variable

Concept:

- Use a template reference variable (e.g., `#myTextarea`) on `<textarea>`.
- Bind to `(input)` event to track the current length.

- Display remaining characters based on max limit.

Example

component.ts

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-char-count',
  templateUrl: './char-count.component.html',
})
export class CharCountComponent {
  maxChars = 100;
  currentLength = 0;

  onInput(event: Event) {
    const input = event.target as HTMLTextAreaElement;
    this.currentLength = input.value.length;
  }
}
```

component.html

```html
<textarea
  #myTextarea
  (input)="onInput($event)"
  [attr.maxLength]="maxChars"
></textarea>

<p>{{ maxChars - currentLength }} characters remaining</p>
```

b. Using @ViewChild to access the textarea DOM element

Concept:

- Use `@ViewChild` to get the textarea element reference in the component.
- Listen to input events or trigger updates manually.
- Update remaining characters in the component and bind to the template.

Example

component.ts

```
import { Component, ViewChild, ElementRef, AfterViewInit } from '@angular/c
ore';

@Component({
  selector: 'app-char-count',
  templateUrl: './char-count.component.html',
})
export class CharCountComponent implements AfterViewInit {
  @ViewChild('myTextarea') myTextarea!: ElementRef<HTMLTextAreaElement
>;

  maxChars = 100;
  currentLength = 0;

  ngAfterViewInit() {
    this.myTextarea.nativeElement.addEventListener('input', () => {
      this.currentLength = this.myTextarea.nativeElement.value.length;
    });
  }
}
```

component.html

```
<textarea #myTextarea [attr.maxLength]="maxChars"></textarea>

<p>{{ maxChars - currentLength }} characters remaining</p>
```

Summary:

Approach	How it works	Use when
Template Reference Variable	Bind `(input)` event and update length directly	Simple and declarative, preferred way
@ViewChild	Access DOM element, listen manually in TS	When direct DOM manipulation needed

---

15. Create a search textbox filter in angular

Here's a simple example to create a search textbox filter in Angular:

a. Setup Angular Component

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
```

```
  searchText: string = '';
  items: string[] = [
    'Apple',
    'Banana',
    'Orange',
    'Grapes',
    'Mango',
    'Pineapple',
    'Strawberry',
  ];

  get filteredItems(): string[] {
    if (!this.searchText) {
      return this.items;
    }
    const lowerSearch = this.searchText.toLowerCase();
    return this.items.filter((item) =>
      item.toLowerCase().includes(lowerSearch)
    );
  }
}
```

b. Template with Search Input and Filtered List

app.component.html

```
<inputtype="text"
  placeholder="Search items"
  [(ngModel)]="searchText"
/>

<ul>
  <li *ngFor="let item of filteredItems">{{ item }}</li>
```

```
</ul>
```

c. Add FormsModule to your App Module

Make sure `FormsModule` is imported in your app module to use `[(ngModel)]`:

app.module.ts

```
typescript
Copy
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';  // import FormsModule here

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

How it works:

- The user types in the textbox (`searchText` bound via ngModel).
- The getter `filteredItems` filters the original list based on the search string.
- The filtered list is displayed dynamically.

---

16. Display radio button data selected by user in another textbox

Here is a simple Angular example showing how to display the selected radio button value inside a textbox:

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  selectedOption: string = '';

  onSelectionChange(value: string) {
    this.selectedOption = value;
  }
}
```

```
<!-- app.component.html -->
<form>
  <label>
    <inputtype="radio"
      name="options"
      value="Option 1"
      (change)="onSelectionChange('Option 1')"
      [checked]="selectedOption === 'Option 1'"
    />
    Option 1
  </label>
```

```
  <label>
    <inputtype="radio"
      name="options"
      value="Option 2"
      (change)="onSelectionChange('Option 2')"
      [checked]="selectedOption === 'Option 2'"
    />
    Option 2
  </label>
  <label>
    <inputtype="radio"
      name="options"
      value="Option 3"
      (change)="onSelectionChange('Option 3')"
      [checked]="selectedOption === 'Option 3'"
    />
    Option 3
  </label>

  <br /><br />
  <input type="text" [value]="selectedOption" readonly />
</form>
```

How it works:

- When a radio button is selected, the `onSelectionChange` method is called with the value of the selected option.
- The component property `selectedOption` stores the current selection.
- The textbox uses Angular's property binding `[value]="selectedOption"` to display the selected radio button value dynamically.

---

17. Create an error boundary component in angular ?

Angular doesn't have a built-in concept called "Error Boundary" like React, but you can achieve similar behavior using Angular's `ErrorHandler` class combined with a component to show fallback UI on error.

Minimal example to create an error boundary component in Angular:

Step 1: Create a global error handler

Create a custom error handler service by extending Angular's `ErrorHandler`:

```
// error-boundary-handler.service.ts
import { ErrorHandler, Injectable, Injector } from '@angular/core';
import { ErrorBoundaryService } from './error-boundary.service';

@Injectable()
export class ErrorBoundaryHandler implements ErrorHandler {
  constructor(private injector: Injector) {}

  handleError(error: any) {
    const errorBoundaryService = this.injector.get(ErrorBoundaryService);
    errorBoundaryService.setError(error);

    // Log the error or send to server
    console.error('Caught by ErrorBoundaryHandler:', error);
  }
}
```

Step 2: Create an ErrorBoundaryService to share error state

```
// error-boundary.service.ts
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
```

```
@Injectable({ providedIn: 'root' })
export class ErrorBoundaryService {
  private errorSubject = new BehaviorSubject<any>(null);
  error$ = this.errorSubject.asObservable();

  setError(error: any) {
    this.errorSubject.next(error);
  }

  clearError() {
    this.errorSubject.next(null);
  }
}
```

Step 3: Create an ErrorBoundaryComponent to show fallback UI

```
// error-boundary.component.ts
import { Component } from '@angular/core';
import { ErrorBoundaryService } from './error-boundary.service';

@Component({
  selector: 'app-error-boundary',
  template: `
    <ng-container *ngIf="error$ | async as error; else content">
      <div class="error-fallback">
        <h2>Something went wrong!</h2>
        <pre>{{ error?.message || error }}</pre>
        <button (click)="clearError()">Try Again</button>
      </div>
    </ng-container>

    <ng-template #content>
      <ng-content></ng-content>
    </ng-template>
```

```
  `,
  styles: [`
    .error-fallback {
      border: 1px solid red;
      padding: 20px;
      background: #ffe6e6;
      color: #900;
    }
  `]
})
export class ErrorBoundaryComponent {
  error$ = this.errorBoundaryService.error$;

  constructor(private errorBoundaryService: ErrorBoundaryService) {}

  clearError() {
    this.errorBoundaryService.clearError();
  }
}
```

Step 4: Register the custom error handler globally

Add the provider in your `app.module.ts`:

```
import { NgModule, ErrorHandler } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { ErrorBoundaryComponent } from './error-boundary.component';
import { ErrorBoundaryHandler } from './error-boundary-handler.service';

@NgModule({
  declarations: [AppComponent, ErrorBoundaryComponent],
  imports: [BrowserModule],
  providers: [
```

```
    { provide: ErrorHandler, useClass: ErrorBoundaryHandler }
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Step 5: Wrap components inside `<app-error-boundary>`

Use the `ErrorBoundaryComponent` as a wrapper wherever you want to catch and display errors:

```
<!-- app.component.html -->
<app-error-boundary>
  <app-child-component></app-child-component>
</app-error-boundary>
```

If an error occurs anywhere in the child components, the error boundary will show fallback UI.

Summary:

- `ErrorBoundaryHandler` catches uncaught errors globally.
- `ErrorBoundaryService` shares error state reactively.
- `ErrorBoundaryComponent` listens for errors and shows fallback UI.
- Wrap components in `<app-error-boundary>` to isolate error display

---

18. Create a counter component in angular ?

✅ Step 1: Generate Component

First, create a new Angular component via CLI:

```
ng generate component counter
```

This will create:

```
src/app/counter/
├── counter.component.ts
├── counter.component.html
├── counter.component.css
└── counter.component.spec.ts
```

✅ Step 2: Add Logic

🔷 **counter.component.ts**

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css']
})
export class CounterComponent {
  count = 0;

  increment(): void {
    this.count++;
  }

  decrement(): void {
    this.count--;
  }
```

```
  reset(): void {
    this.count = 0;
  }
}
```

✅ Explanation:

- `count` is a class property that stores the current counter value.
- `increment()`, `decrement()`, and `reset()` are methods to change the value.
- These methods are triggered when buttons are clicked.

✅ Step 3: Create Template

🔷 counter.component.html

```
<div class="counter">
  <h2>Counter Value: {{ count }}</h2>

  <div class="buttons">
    <button (click)="decrement()">–</button>
    <button (click)="reset()">Reset</button>
    <button (click)="increment()">+</button>
  </div>
</div>
```

✅ Explanation:

- `{{ count }}`: Angular interpolation to display the counter value.
- `(click)="method()"`: Event binding to trigger corresponding methods on button click.

✅ Step 4: Add Styles (Optional)

🔷 `counter.component.css`

```
.counter {
  text-align: center;
  margin-top: 2rem;
  font-family: Arial, sans-serif;
}

.buttons {
  display: flex;
  justify-content: center;
  gap: 1rem;
  margin-top: 1rem;
}

button {
  font-size: 1.5rem;
  padding: 0.5rem 1rem;
  cursor: pointer;
}
```

✅ Step 5: Use Component in App

In `app.component.html` or any other parent component:

```
<app-counter></app-counter>
```

Make sure it's declared in `app.module.ts` (done automatically when generated with CLI).

✅ Output

A fully working Angular counter that looks like this:

```
less
CopyEdit
Counter Value: 0
[ - ]  [ Reset ]  [ + ]
```

---

19. How to call a method on every rerender of a component ?

In **Angular**, if you want to **call a method on every re-render (change detection cycle)** of a component — the best place to do this is in the `ngDoCheck()` **lifecycle hook**.

✅ What is `ngDoCheck()` ?

`ngDoCheck()` is a lifecycle hook that Angular calls **during every change detection run**, i.e., **whenever the component is checked for changes**, which happens:

- when input properties change,
- on events,
- or when you explicitly trigger change detection.

So it's the Angular equivalent of "every re-render" in React.

✅ Example: Call a Method on Every Rerender

🔷 Step 1: Component File

```
import { Component, DoCheck } from '@angular/core';

@Component({
  selector: 'app-my-component',
```

```
  templateUrl: './my-component.component.html'
})
export class MyComponent implements DoCheck {
  counter = 0;

  constructor() {}

  ngDoCheck(): void {
    this.onEveryRerender(); // Called during every change detection run
  }

  onEveryRerender(): void {
    console.log('Component re-rendered!');
  }

  triggerChange(): void {
    // This method is to trigger a change
    this.counter++;
  }
}
```

🔷 HTML Template

```
<h2>Counter: {{ counter }}</h2>
<button (click)="triggerChange()">Increase Counter</button>
```

✅ Explanation:

- `ngDoCheck()` is triggered on every Angular change detection.
- Inside it, we call `onEveryRerender()` which logs a message or performs any logic.
- Clicking the button updates the counter, which causes a change detection → which calls `ngDoCheck()`.

⚠️ Note:

- Use `ngDoCheck()` **carefully** because it's called **very frequently**.
- Don’t perform **heavy logic** here unless it’s necessary.
