**Javascript**

1. **What Will be output of this and why**

const myObject = {

regularExample: function () {

console.log("REGULAR: ", this);

},

arrowExample: () => {

console.log("ARROW: ", this);

},

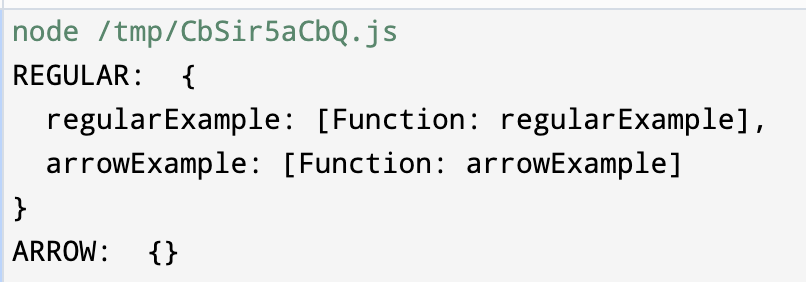
};

myObject.regularExample();

myObject.arrowExample();

It wll show output on browser and shell(nodejs) differently  
  
In browser it will print global object



And in node js it will print   
  
  


1. **Can we create object of arrow function explain.  
   Answer :** No we can not create because **,** As you have already seen, an arrow function does not have the prototype object; hence it cannot be used to create an object or, in other words, cannot be used as a constructor. An arrow function: Does not have the prototype property. Also does not have the internal [[Constructor]] property.

1. **How to use Functions as Constructors explain normal with arrow diff..  
   Answer :**

**function RegularFuncBird(name, color) {**

**this.name = name;**

**this.species = color;**

**console.log(this);**

**}**

**const ArrowFuncBird = (name, color) => {**

**this.name = name;**

**this.color = color;**

**console.log(this);**

**};**

**new RegularFuncBird("Parrot", "blue");**

**new ArrowFuncBird("Parrot", "blue");**

****

1. **Difference between forEach() & map()  
   Answer :   
     
   forEach() : -** just loop through the elements. It's throws away return values

and always returns undefined.The result of this method does not give us an output .

**map() :-** loop through the elements allocates memory and stores return values by

iterating main array.

Now what will be the output of this below code.

var numbers = [2, 3, 5, 7];

var forEachNum = numbers.forEach(function (number) {

return number;

});

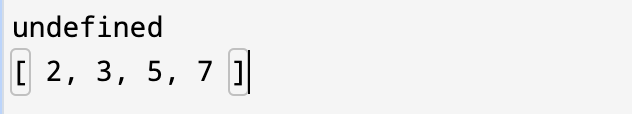
console.log(forEachNum);

var mapNum = numbers.map(function (number) {

return number;

});

console.log(mapNum);

.

1. **Different between find, filter and map  
   Answer:**

**Filter**

filter() method creates an array filled with all array elements that

pass a test (provided as a function).

**var ages = [32, 33, 16, 40];**

**function checkAdult(age) {**

**return age >= 18;**

**}**

**const data = ages.filter(checkAdult);**

**console.log(data);**

**Output : [ 32, 33, 40 ]**

**Find :**

The find() method returns the value of the first element in an array that

pass a test (provided as a function).

var ages = [10, 33, 16, 40];

function checkAdult(age) {

return age >= 18;

}

const data2 = ages.find(checkAdult);

console.log(data2);

**Output : 33̦**

1. **Explain reduce, Every & Some function  
   Answer   
   every() :**  returns **true** if all elements in an array pass a test else it will return **false**.  
   **reduce() :**  method reduces the array to a single value.

const array1 = [1, 2, 3, 4];

const initialValue = 5;

const sumWithInitial = array1.reduce((accumulator, currentValue) => {

return accumulator + currentValue;

}, initialValue);

console.log(sumWithInitial);

**Some() :** This method checks if any of the elements in an array pass a test

(provided as a function).

1. **Explain some Object Methods  
   Answer :**

**Object.entries() :**

**let object = {**

**name: "Rohan",**

**email: "Rohan@gmail.com",**

**company: "Google",**

**};**

**const entries = Object.entries(object);**

**console.log(entries);**

Output   
  
  
  
  
**Object.freeze :** The Object.freeze() static method freezes an object.

Freezing an object prevents extensions and makes existing properties

non-writable and non-configurable. A frozen object can no longer be changed:

new properties cannot be added, existing properties cannot be removed, their

enumerability, configurability, writability, or value cannot be changed, and the

object's prototype cannot be re-assigned. freeze() returns the same object

that was passed in.  
  
**Object.fromEntries** :

let object = [

["name", "Rohan"],

["email", "Rohan@gmail.com"],

["company", "Google"],

];

const entries = Object.fromEntries(object);

console.log(entries);

Output:

{

name: "Rohan",

email: "Rohan@gmail.com",

company: "Google",

}

**Object.hasOwn()** : The Object.hasOwn() static method returns true

if the specified object has the indicated property as its own property. If the

property is inherited, or does not exist, the method returns false.

1. **What will ‘5’ + 3 and ‘5’ — 3 return?**

**Answer :**‘5’ + 3 returns the string ‘53’. This is because when you try to

use the “+” operator with a string and a number, JS performs type

coercion, converting the number to a string and concatenating it with the

original string.

‘5’ — 3 returns the number 2. Interestingly, the “—” operator does not

perform string concatenation. Instead, it coerces both operands to

numbers and performs the substraction.

1. **Write a program for below input and output produce  
   Input :** Welcome to this Javascript Guide!  
   **Output :** emocleW ot siht tpircsavaJ !ediuG **Answer :**

**var string = "Welcome to this Javascript Guide!";**

**var reverseEntireSentence = reverseBySeparator(string, "");**

**var reverseEachWord = reverseBySeparator(reverseEntireSentence, " ");**

**console.log(reverseEachWord);**

**function reverseBySeparator(string, separator) {**

**return string.split(separator).reverse().join(separator);**

**}**

**Another Way**

**const reverseWordsInSent = (inpStr) => {**

**return inpStr.split(' ').reduce((str, currStr) => {**

**console.log(currStr);**

**const revStr = currStr.split('').reverse().join('')**

**str.push(revStr)**

**return str**

**}, []).join(' ')**

**}**

**console.log(reverseWordsInSent("Welcome to this Javascript Guide!"))**

1. **What will be the following code output ?**

**Answer:**

**(function () {**

**var a = (b = 5);**

**})();**

**console.log(b);**

The code above will output 5 even though it seems as if the variable was declared within a function and can't be accessed outside of it.

This is because var a = b = 5; is interpreted the following way:

var a = b;

b = 5;

But b is not declared anywhere in the function with var so it is set equal to 5 in the global scope.

1. **Explain the output of following code   
   Answer**

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

**setTimeout(() => console.log(i), 0);**

**}**

**Output : 4 4 4 4**

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

**setTimeout(() => console.log(i), 0);**

**}**

**Output : 0 1 2 3**

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

**setTimeout(**

**(d) => {**

**console.log(d);**

**},0,i**

**);**

**}**

**Output : 0 1 2 3**

**function printValue(val) {**

**setTimeout(() => {**

**console.log(val);**

**}, 0);**

**}**

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

**printValue(i);**

**}**

**Output : 0 1 2 3**

1. **Explain Shallow and deepcopy also explain the following code.  
   Answer :  
   Shallow copying-**  creates a new object with

references to the same memory locations as the original object, while

**Deep Copy** creates a new object with new memory locations for all of its

properties and nested objects or arrays.

**Program 1**

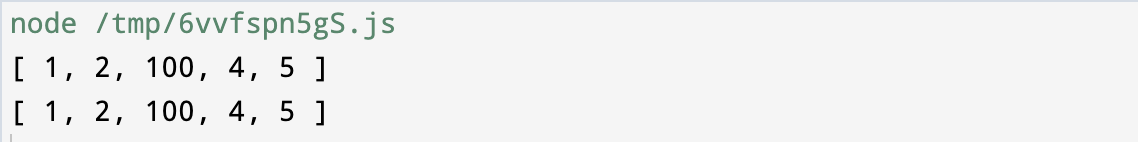
**let arr1 = [1, 2, 3, 4, 5];**

**let arr2 = arr1;**

**arr2[2] = 100;**

**console.log(arr1);**

**console.log(arr2);**

**  
  
  
Program 2**

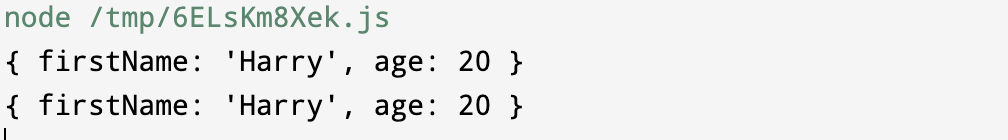
**let obj1 = { firstName: "Harry", age: 25 };**

**let obj2 = obj1;**

**obj2.age = 20;**

**console.log(obj1);**

**console.log(obj2);**

**  
  
  
Program 3**

**let arr1 = [1, 2, 3, 4, 5];**

**let arr2 = [...arr1];**

**arr2[2] = 100;**

**console.log(arr1);**

**console.log(arr2);**

**  
  
Program 4**

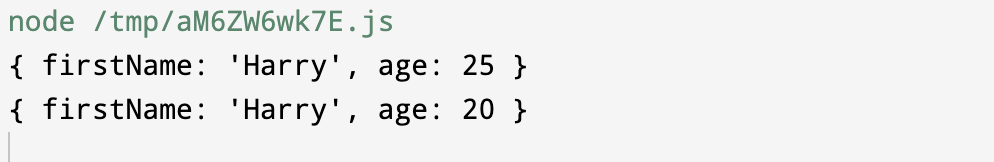
**let obj1 = { firstName: "Harry", age: 25 };**

**let obj2 = { ...obj1 };**

**obj2.age = 20;**

**console.log(obj1);**

**console.log(obj2);**

****

1. **How We can also achieve deep copy Explain all method  
   Answer :**- Using the **spread operator** method. [...] {...}   
   - Using **JSON.parse(JSON.stringify())**- Using **Object.assign()**- Using Lodash’s **\_.cloneDeep()**
2. **Explain Prototyping in js  
   Answer:**JavaScript is a prototype based language, so, whenever we

create a function using JavaScript, JavaScript engine adds a prototype

property inside a function, Prototype property is basically an object (also

known as Prototype object), where we can attach methods and properties

in a prototype object, which enables all the other objects to inherit these

methods and properties.

function Person(name, job, yearOfBirth) {

this.name = name;

this.job = job;

this.yearOfBirth = yearOfBirth;

}

Person.prototype.calculateAge = function () {

console.log("The current age is:" + (2019 - this.yearOfBirth));

};

const user1 = new Person("Rohan", "Software", 1995);

console.log(user1.calculateAge());

We can also create prototype by using **\_\_proto\_\_**

Like that

const a = {

name: "Rohan",

language: "MERN",

};

console.log(a);

const p = {

run: () => {

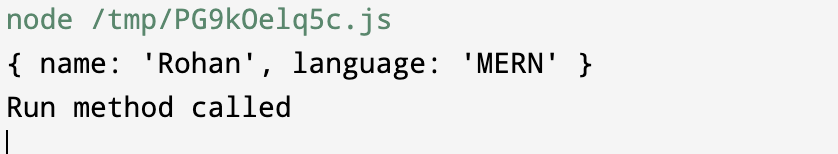
alert("Run method called");

},

};

a.\_\_proto\_\_ = p;

a.run();

  
  
  
  
  
We can also do nested level prototypes and also define some properties later assigned.

const a = {

name: "Rohan",

language: "MERN",

run: () => {

alert("run inside a");

},

};

console.log(a);

const p = {

run: () => {

alert("Run method called");

},

};

a.\_\_proto\_\_ = p;

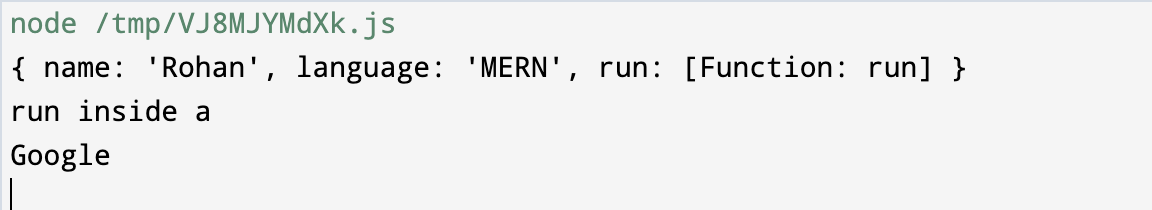
p.\_\_proto\_\_ = {

name2: "Google",

};

a.run();

console.log(a.name2);



1. **Explain Closure in js and explain output of following program  
   Answer:  
   A closure** is the combination of a function bundled together

(enclosed) with references to its surrounding state (the lexical

environment).

In other words, a closure gives you access to an outer

function's scope from an inner function. In JavaScript, closures are

created every time a function is created, at function creation time.

**Lexical Scope :** The lexical scope is the scope of a variable or function

based on where it is defined in the source code. The scope is determined

by the placement of variables and functions in the code, and it remains

the same throughout the execution of the program.

**function init() {**

**var name = "Mozilla";**

**function displayName() {**

**console.log(name);**

**}**

**name = "Harry";**

**return displayName;**

**}**

**let c = init();**

**c();**

**Output : Harry**

**function returnFunc() {**

**const x = () => {**

**let a = 1;**

**console.log(a);**

**const y = () => {**

**console.log(a);**

**const z = () => {**

**let a = 10;**

**console.log(a);**

**};**

**z();**

**};**

**a = 999;**

**y();**

**};**

**return x;**

**}**

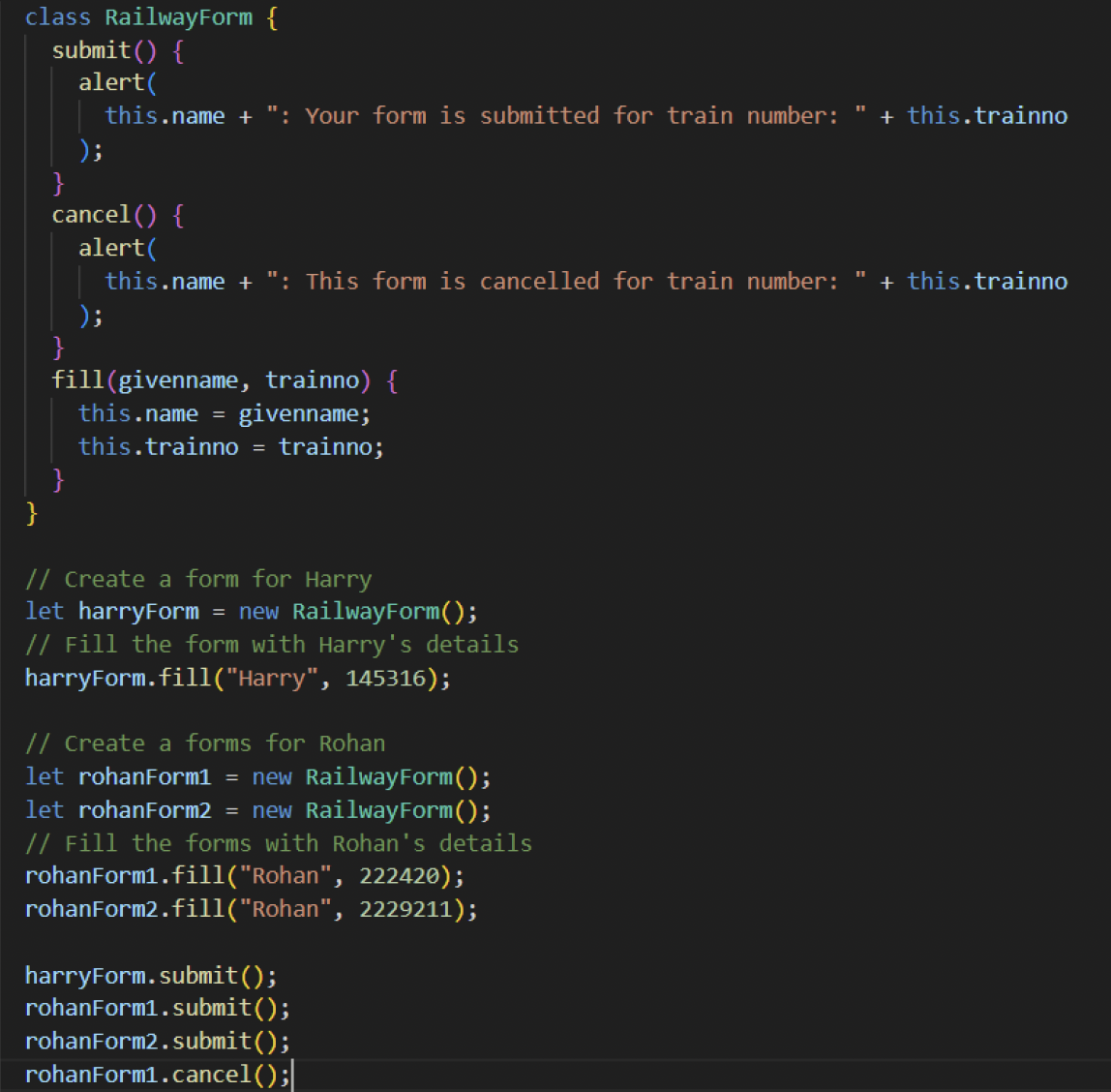
**let a = returnFunc();**

**a();**

**Output :   
  
1**

**999**

**10**

1. **Define a class and create an object and call its property>  
   Answer :   
     
   **
2. **Explain Es6 Features.**
3. **Explain Typescript ( Please refer to my Created React Interview Doc for typescript question).  
   Please find => “ Explain TypeScript in React js ” question no 65**
4. **JavaScript engines have two places to store data**

**Answer :   
Stack:** It is a data structure used to store static data. Static data refers to data whose size is known by the engine during compile time. In JavaScript, static data includes primitive values like strings, numbers, boolean, null, and undefined. References that point to objects and

functions are also included. A fixed amount of memory is allocated for static data. This process is known as static memory allocation.

**Heap:** It is used to store objects and functions in JavaScript. The engine doesn’t allocate a fixed amount of memory. Instead, it allocates more space as required.

1. **What is Callback Queue.**

**Answer:**

**Callback Queue:** After the timer gets expired, the callback function is put inside the Callback Queue, and the Event Loop checks if the Call Stack is empty and if empty, pushes the callback function from Callback Queue toCall Stack and the callback function gets removed from the Callback Queue.

1. **Javascript Call Stack**

**Answer:**

**The call stack** is a crucial concept in JavaScript's runtime environment, representing the mechanism by which the JavaScript engine keeps track of function calls in a program. It operates as a Last In, First Out (LIFO) data structure, meaning that the last function called is the first one to be resolved.   
If any function called all the variable of that function used are save to memory (heap & stack memory)After function is executed successfully then its variable is removed from memory.

1. **Explain Event bubbling, Event Propagation, Event Capturing, Event Delegation.**
2. **Explain Call Apply & Bind.  
   Answer :**

**function sayHello() {**

**return "Hello" + this.name;**

**}**

**var obj = { name: "Sandy" };**

**sayHello.call(obj);**

**// Output "Hello Sandy"**

**var person = {**

**age: 23,**

**getAge: function (city, country) {**

**return this.age + " " + city + " " + country;**

**},**

**};**

**var person2 = { age: 54 };**

**console.log(person.getAge.call(person2, "Udaipur", "Bharat"));**

**// Output "54 Udaipur Bharat"**

**var person = {**

**age: 23,**

**getAge: function (city, country) {**

**return this.age + " " +city + " " +country;**

**},**

**};**

**var person2 = { age: 54 };**

**console.log(person.getAge.apply(person2, ["Udaipur",'Bharat']));**

**// Output "54 Udaipur Bharat"**

1. **Explain Following operator   
   Answer   
     
   The logical AND assignment (&&=)** operator only evaluates the right operand and assigns to the left if the left operand is truthy.

**let a = 1;**

**let b = 0;**

**a &&= 2;**

**console.log(a);**

**b &&= 2;**

**console.log(b);**

**// Expected output: 2  
// Expected output: 0**

**The logical OR assignment (||=)** operator only evaluates the right operand and assigns to the left if the left operand is falsy.

**const a = { duration: 50, title: "" };**

**a.duration ||= 10;**

**console.log(a.duration);**

**a.title ||= "title is empty.";**

**console.log(a.title);**

**// Expected output: 50  
// Expected output: "title is empty"**

**The nullish coalescing assignment (??=)** operator, also known as the logical nullish assignment operator, only evaluates the right operand and assigns to the left if the left operand is nullish (null or undefined).

**const a = { duration: 50 };**

**a.speed ??= 25;**

**console.log(a.speed);**

**a.duration ??= 10;**

**console.log(a.duration);**

**// Expected output: 25**

**// Expected output: 50**

1. **Write This Program  
   Input : [1, 2, 5, 4, [7, 8, 9, [8, 9, 8]]]  
   Output : [ 1, 2, 5, 4, 7, 8, 9, 8, 9, 8 ]**

**Answer :-**

**const a = [1, 2, 5, 4, [7, 8, 9, [8, 9, 8]]];**

**const aa = (data) => {**

**let newData = [];**

**if (Array.isArray(data)) {**

**data.map((item, index) => {**

**newData.push(...aa(item));**

**});**

**} else {**

**newData.push(data);**

**}**

**return newData;**

**};  
 console.log(aa(a))**

1. **Explain Following React Concept  
   Answer :**

* **useLayoutEffect=>** page load hone se phle chlta h useEffect se phle (use for dom update, work before screen updated, runs after every render,works syncrounsly)

**useLayoutEffect** : runs synchronously after a render but before the screen is updated(pained).

**useEffect**: runs asynchronously and after a render is painted to the screen.

**useLayoutEffect** will run first then **useEffect** will run

useEffect(() => {

    console.log('First useEffect......');

  }, []);

  useEffect(() => {

    console.log('Second useEffect......');

  }, []);

  useLayoutEffect(() => {

    console.log('Third useLayoutEffect....');

  }, [])

Output:   
Third useLayoutEffect....

First useEffect......

Second useEffect......

* **forwardRef:**
* **lazyLoading;**
* **SuspenseList:**  SuspenseList is a **React concurrent feature** that controls how multiple Suspense components are revealed on the screen. It ensures a smoother loading experience by **coordinating the order** in which the content appears.

| **Feature** | **Description** |
| --- | --- |
| **SuspenseList** | Controls rendering order of multiple Suspense components |
| **forwards** | Reveals components from first to last |
| **backwards** | Reveals components from last to first |
| **together** | Reveals all components at once |

📌 **SuspenseList is still experimental and requires React 18+ with concurrent features enabled.**

<SuspenseList revealOrder="together" tail=”collapsed/hidden”>

<Suspense fallback={<div>Loading Profile...</div>}>

<Profile />

</Suspense>

<Suspense fallback={<div>Loading Posts...</div>}>

<Posts />

</Suspense>

<Suspense fallback={<div>Loading Comments...</div>}>

<Comments />

</Suspense>

</SuspenseList>

* **forin forof**
* **react.memo=>** use for pure(complete) component ko hi memoize krta h , works on primitive data types only props me pass krne pr, second parameter me function pass ker skte h usme we can compare nextprop and prev prop
* **useMemo =>** use for values ko memoize krta expects 2 parameters first function and second dependancy array(similar to useEffect) for big calculations
* **useCallback =>** use for memoize the function first function and second dependancy array(similar to useEffect)
* **useReducer =>** const [state,dispatch] = useReducer(reducer,initialData)=> reducer ek function jo task perform krta h state and actions k basis pr and ek initial data leta h dispatch me 2 parameters le skte h type jo action type hota h and another one payload to send the data
* **useState =>**
* **useEffect :**

**1️ useEffect Without a Dependency Array**

useEffect(() => {

console.log("Effect runs on every render");

});

**🔹 Behavior:**

* Runs **on every render** (including re-renders).
* Can cause performance issues if not handled properly.

**🔹 When to Use?**

✅ When you need to run an effect on **every render**, like logging updates.

**2️ useEffect With an Empty Dependency Array []**

useEffect(() => {

console.log("Effect runs only on mount");

}, []);

**🔹 Behavior:**

* Runs **only once**, when the component mounts.
* **Does NOT re-run** on re-renders.

**🔹 When to Use?**

✅ Fetching initial data (API calls).  
✅ Setting up event listeners.  
✅ Subscribing to WebSocket connections.

**3️ useEffect With Dependencies [dependency]**

useEffect(() => {

console.log(`Effect runs when count changes: ${count}`);

}, [count]);

**🔹 Behavior:**

* Runs **on mount** and whenever count changes.
* Avoids unnecessary re-renders by only running when dependencies change.

**🔹 When to Use?**

✅ Fetching data when a prop/state changes.  
✅ Updating the DOM when a variable changes.

**4️ useEffect With Cleanup Function**

useEffect(() => {

console.log("Effect runs on mount");

return () => {

console.log("Cleanup runs before unmounting");

};

}, []);

**🔹 Behavior:**

* return () => {} runs **before the component unmounts**.
* Useful for **cleaning up event listeners, intervals, or subscriptions**.

**🔹 When to Use?**

✅ Clearing timeouts/intervals.  
✅ Unsubscribing from API/WebSockets.

**Example: Cleaning Up an Interval**

useEffect(() => {

const interval = setInterval(() => {

console.log("Interval running...");

}, 1000);

return () => {

clearInterval(interval);

console.log("Interval cleared on unmount");

};

}, []);

* **useRef:**
* **useContext** 
  + Context API: A way to pass data through the component tree without having to pass props down manually at every level.
  + **createContext:** Creates a Context object.
  + **Provider:** A component that provides the context value to its children.
  + **useContext (Consumer):** A hook that allows you to consume a context.
* **useImparitiveHandle**
* **useTransition,**
* **useDeferredValue,**
* **useId,**
* **useSyncExternalStore**
* **HOC in React**
* **Portals**
* **Profiler**

1. **What will be the output of this>  
   Answer**

**let x = 'Hello' && false;**

**let y = 1 && 'Yes' && true;**

**let z = true && 'Hello' && 10;**

**let x1 = '' || false;**

**let y1 = 0 || 'Yes' || true;**

**let z1 = true || 'Hello' || 10;**

**let x2 = !false;**

**let y2 = !('Hello')**

**Output   
  
false**

**true**

**10**

**false**

**Yes**

**true**

**True *// x is equal to true***

**false *// y is equal to false ('Hello' is truthy)***

1. **Write program for find percentage of given object  
   Input : { '1': 0, '2': 1, '3': 1, '4': 1, '5': 1 };  
   Output : { '1': '0%', '2': '25%', '3': '25%', '4': '25%', '5': '25%' }**

**const getPercentages = (ratingCounts) => {**

**const total = Object.values(ratingCounts).reduce((sum, count) => sum +count, 0);**

**const percentages = {};**

**for (const [rating, count] of Object.entries(ratingCounts)) {**

**const percentage = total === 0 ? 0 : ((count / total) \* 100).toFixed(0);**

**percentages[rating] = `${percentage}%`;**

**}**

**return percentages;**

**};**

**const ratingPercentages = getPercentages(ratingCounts);**

**console.log(ratingPercentages);**