Mohammed Fardeen F

STANCH SOLUTIONS PRIVATE LIMITED

JAYANAGAR 4TH T BLOCK,

BENGALURU.

Q1. What is React?

Ans: React is a frontend javascript library used for building fast and interactive user interface.

- It was developed by meta (facebook) in 2011.
- It is used to build user interface based on UI components.

Q2. What is JSX?

Ans: JSX is a Java Script XML, it is a syntax that allows us to write HTML-like code within JavaScript.

- JSX makes it easy to describe what the UI should look like and then render it using React's rendering engine.
- For example:

```
const element = <h1>Hello, world!</h1>;
```

Q3. What are props in React?

Ans: props is a short form of "properties" it is used to pass data from one component to another.

- It passes data from a parent component to a child component.

Q4. How does the Virtual DOM work?

Ans: Virtual DOM is a lightweight copy of the actual DOM (the structure of your webpage).

- When a React component changes (like when you click a button), React creates a new Virtual DOM that represents the updated component.
- It then compares this new Virtual DOM to the previous one to find out what has changed.
- Instead of updating the entire webpage, React only changes the parts that actually need to be updated in the real DOM.

Q5. What are stateless components?

Ans: Stateless components are simple functions in React that display things on the screen.

- Someone gives it instructions (props), and it does exactly that without keeping track of anything.
- Example:

```
const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

- In this example, the Greeting component doesn't remember anything. It just says "Hello" to whatever name it is given!

Q6. What are stateful components?

Ans: A **stateful component** is a component in React that can **remember things** and **change** based on what happens.

- It uses something called **state** to keep track of its own information, like counting clicks or keeping track of form input.
- It's like a box that holds some data (state) and can update that data whenever something happens, like a button click.
- Example:

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

const ClickCounter = () => {

const [count, setCount] = useState(0); // 'count' is the state, and we start at 0

return (

    <div>

    <p>You clicked {count} times</p>

    <button onClick={() => setCount(count + 1)}>Click me</button> {/*
Each click    updates the state */}

</div>

  );

};
```

- When you click the button, the count increases.
- The component **remembers** the count using state and shows the updated number every time you click.

- In simple terms, stateful components **remember** things and change based on what happens

Q7. Why should component names start with a capital letter?

Ans: n React, component names should start with a **capital letter** so that React can tell the difference between a **custom component** and a regular **HTML tag**.

- If your component name starts with a **capital letter** (like MyButton), React knows it's your own custom component.
- If it starts with a **lowercase letter** (like myButton), React thinks it's a normal HTML tag, like or <p>, and this will cause problems.

Example

```
const MyComponent = () => {
  return <h1>Hello!</h1>;
};

<MyComponent />
```

Q8. Why can't you update props in React?

Ans: In React, you **can’t change props** because they are **read-only**. They are passed from the parent to the child component, and the child can only use them, not change them.

- **Props** are like **gifts** given to a child component by the parent. The child can **use** the gift, but it **can’t change** it.
- Example:

```
const Parent = () => {
const [name, setName] = useState('John');

  return <Child name={name} />;
};

const Child = (props) => {
return <h1>Hello, {props.name}</h1>;
```

- Here, Child receives name from the parent but can’t modify it.
- If the **child** wants to change something, the **parent** has to handle that.

Q9. Is it possible to use React without JSX?

Ans: Yes, you can use React without JSX, but it's more complicated. JSX is a special way of writing code that looks like HTML, but React can work without it by using a built-in unction called React.createElement.

- JSX makes it easier to write UI code that looks like HTML.
- Without JSX we can use React.createElement() to build the same thing, but it takes more code and can be harder to read.
- Example:
  ```
  const Hello = () => {
    return React.createElement('h1', null, 'Hello, world!');
  };
  ```

Without JSX, you have to manually create each element, which can get messy for bigger components. That's why most people prefer using JSX—it's simpler and clearer!

Q10. What is JSON and its common operations?

Ans: **JSON (JavaScript Object Notation)** is a simple way to store and share data. It looks like plain text, but it's organized in a way that's easy for computers to understand and for humans to read. It's commonly used to send data between a server and a website (like sending data from an API).

- Example of JSON:

```
{
 "name": "John",
 "age": 30,
 "isStudent": false,
 "courses": ["Math", "Science"]
}
```

- **Common Operations with JSON:**
  1. **Parsing (Converting JSON to an object)**: This means turning a JSON string into something your program can use, like an object in JavaScript.
     - Example in JavaScript:
       ```
       const jsonString = '{"name": "John", "age": 30}';
       const obj = JSON.parse(jsonString);  // Converts JSON to an object
       console.log(obj.name);  // Outputs: John
       ```
  2. **Stringifying (Converting an object to JSON)**: This means turning an object into a JSON string to send it somewhere or save it.
     - Example in JavaScript:
       ```
       const obj = { name: "John", age: 30 };
       const jsonString = JSON.stringify(obj);  // Converts object to JSON string
       ```

```
console.log(jsonString);  // Outputs: '{"name":"John","age":30}'
```

3. **Accessing Data**: You can access data in JSON using keys (like "name" or "age").
   - Example:
     ```
     const person = { name: "John", age: 30 };
     console.log(person.name);  // Outputs: John
     ```

4. **Modifying Data**: You can change JSON data like you do with normal objects or arrays.
   - Example:

     ```
     const person = { name: "John", age: 30 };
     person.age = 31;  // Changes the age
     console.log(person.age);  // Outputs: 31
     ```

Q11. What is the difference between slice and splice?

Ans: **slice()**: Takes a part of an array without changing the original array.

- Returns a new array with the selected elements.
- Example:

  ```
  let fruits = ['apple', 'banana', 'cherry', 'date'];
  let sliced = fruits.slice(1, 3); // Takes part from index 1 to 3 (not including 3)

  console.log(sliced); // ['banana', 'cherry']
  console.log(fruits); // ['apple', 'banana', 'cherry', 'date'] (original array stays the same)
  ```

splice(): Changes the original array by adding or removing elements.

- Returns **the removed elements (if any).**
- Example:

  ```
  let fruits = ['apple', 'banana', 'cherry', 'date'];
  let removed = fruits.splice(1, 2); // Removes 2 items starting from index 1

  console.log(removed); // ['banana', 'cherry']
  console.log(fruits); // ['apple', 'date'] (original array is changed)
  ```

Q12. How do you compare Object and Map?

Ans: Comparing **Object** and **Map** in JavaScript involves looking at their characteristics, usage, and behavior. Here are the key differences:

1. Key Types

- **Object**: Keys are always strings (or Symbols). If you use a number or another type, it gets turned into a string.
- **Map**: Keys can be anything, like strings, numbers, or even objects.

2. Order of Keys

- **Object**: The order of keys isn't guaranteed.
- **Map**: Keeps the order of keys as you add them.

3. Performance

- **Object**: Slower for adding or removing keys, especially if there are many.
- **Map**: Faster for adding and removing key-value pairs.

4. Iteration

- **Object**: You can loop through keys using for...in or Object.keys(), but it's less straightforward.
- **Map**: Has built-in methods to easily loop through keys, values, or both.

5. Size

- **Object**: You have to count keys manually with Object.keys(obj).length.
- **Map**: You can simply use the size property to get the number of key-value pairs.

6. Prototype

- **Object**: Inherits properties from Object.prototype, which can lead to extra properties.
- **Map**: Doesn't have inherited properties, so it's cleaner.

7. Use Cases

- **Object**: Great for simple data structures where keys are strings.
- **Map**: Better when you need different types of keys or when you want to maintain the order of entries.

Example:

Object Example:

```
const person = {
  name: "John",
  age: 30,
};

// Accessing data
console.log(person.name); // John
```

Map Example:

```
const personMap = new Map();
personMap.set("name", "John");
personMap.set(1, "One");

// Accessing data
console.log(personMap.get("name")); // John
```

Q13. What is the difference between == and === operators

Ans: **== (Equality Operator)**

- **Type Coercion**: This operator checks if two values are equal **after changing their types** to match.
- **Example**:
  5 == '5' → **true** (the string '5' becomes a number 5).

=== (Strict Equality Operator)

- **No Type Coercion**: This operator checks if two values are equal **without changing their types**. Both the value and type must match.
- **Example**:
  5 === '5' → **false** (number 5 is not the same as string '5').

Q14. What are lambda expressions or arrow functions?

Ans: Arrow functions (also known as **lambda expressions**) are a shorter way to write functions in JavaScript.

Q15. What is the currying function? Give an example.

Ans: Currying is a technique in programming where a function takes one argument at a time instead of all arguments at once.

- It transforms a function that takes multiple arguments into a series of functions that each take a single argument.
- Example:

```
function add(a) {
  return function(b) {
    return function(c) {
      return a + b + c;
    };
  };
}

// Using the curried function
const add5 = add(5);          // First call, fixing a = 5
const add5And3 = add5(3);     // Second call, fixing b = 3
const result = add5And3(2);   // Third call, fixing c = 2

console.log(result); // Outputs: 10 (5 + 3 + 2)
```

Q16. What is the difference between let and var?

Ans: We use **let** for variables that need to be limited to a specific block of code, like inside loops or if statements.

We use **var** if you need a variable that works throughout a whole function or globally, but it's generally better to use let or const in modern JavaScript.

React & JS Questions Entry Level 2

Q17. What is the Temporal Dead Zone?

Ans: The Temporal Dead Zone is a period when you can't use a variable declared with let or const before you actually declare it in your code. Trying to use it too early results in an error.

- The Temporal Dead Zone is the time between entering a scope and the declaration of a variable where that variable cannot be accessed.
- This behavior helps prevent errors and improves code readability by ensuring variables are declared before use.
- Example:
  console.log(x); // ReferenceError: Cannot access 'x' before initialization let x = 10;
  console.log(y); // ReferenceError: Cannot access 'y' before initialization const y = 20;
- In the first line, trying to log x before it has been declared results in a ReferenceError.
- Similarly, logging y before its declaration also results in a ReferenceError.

Q18. What is an IIFE (Immediately Invoked Function Expression)?

Give an example.

Ans: An IIFE is a function that runs as soon as it's defined. It helps keep your code organized by creating a private space for variables, preventing them from affecting other parts of your program.

- Example:

```
(function() {
  const message = "Hello, World!";
  console.log(message); // Outputs: Hello, World!
})();


// Trying to access 'message' outside the IIFE will result in an error
console.log(message); // ReferenceError: message is not defined
```

- **Function Declaration**: The function is wrapped in parentheses () to indicate that it's an expression.
- **Immediate Invocation**: The () at the end invokes the function immediately.
- **Private Scope**: The variable message is only accessible inside the IIFE, preventing it from interfering with other variables in the global scope.

Q19. What is Hoisting?

Ans: Hoisting is a JavaScript feature that moves variable and function declarations to the top of their scope during the preparation of the code.

This means you can use them before you actually write their declaration, but if you try to use a variable declared with let or const before it's defined, you'll get an error.

Q20. What are closures? Give an example.

Ans: Closures are functions that remember the variables from the place where they were created, even after that place has finished running.

They let you create private variables and keep track of values over time. For example, a counter function can keep counting up without resetting.

- Example:

```
  function makeCounter() {
let count = 0; // A private variable


return function() {
```

```
    count++; // Increment the private variable
    return count; // Return the updated value
  };
}

const counter = makeCounter(); // create a new counter

console.log(counter()); // Outputs: 1 (first call)
console.log(counter()); // Outputs: 2 (second call)
console.log(counter()); // Outputs: 3 (third call)
```

- **Outer Function**: The makeCounter function creates a local variable count and returns an inner function.
- **Inner Function**: The inner function is the closure that has access to the count variable.
- **Private State**: The variable count is private to the makeCounter function. It can only be accessed and modified by the returned inner function.
- **Maintaining State**: Each time you call counter(), it retains the value of count, allowing it to count up each time without resetting.
-

Q21. What are the differences between cookie, local storage, and session storage?

Ans: Cookies, local storage, and session storage are all ways to store data in a web browser, but they have different characteristics and use cases. Here's a breakdown of their differences:

1. Storage Capacity

- **Cookies**: Limited to about 4KB of data. Good for small pieces of information, like user preferences.
- **Local Storage**: Can store around 5-10MB of data (varies by browser). Suitable for larger amounts of data.
- **Session Storage**: Similar to local storage, it can also store about 5-10MB of data.

2. Expiration

- **Cookies**: Can have an expiration date set, after which they are automatically deleted. If no expiration date is set, they expire when the browser is closed.
- **Local Storage**: Data stored here has no expiration date and will persist until it is explicitly deleted by the user or the application.

- **Session Storage**: Data is cleared when the page session ends, which usually happens when the browser or tab is closed.

3. Scope

- **Cookies**: Accessible across all pages of a domain and can be sent to the server with HTTP requests.
- **Local Storage**: Accessible only within the same origin (protocol, hostname, and port). Not sent to the server with requests.
- **Session Storage**: Also accessible only within the same origin but is specific to a single tab or window. Not sent to the server with requests.

4. Data Types

- **Cookies**: Only strings can be stored (but you can stringify objects into strings).
- **Local Storage**: Can store strings, but you can easily convert objects to strings (e.g., using JSON).
- **Session Storage**: Similar to local storage, it can store strings, but you can also store objects by converting them to strings.

5. Usage

- **Cookies**: Commonly used for tracking user sessions, remembering user preferences, and storing small bits of information.
- **Local Storage**: Ideal for storing larger amounts of data that need to persist between sessions, such as user settings or application state.
- **Session Storage**: Useful for storing data that only needs to be available during a single browser session, like temporary form data.

Q22. What is the purpose of double exclamation?

Ans: The double exclamation mark (!!) in JavaScript is a quick way to change any value into true or false. The first ! flips it, and the second ! flips it back, resulting in a boolean representation of the original value.

Q23. What is the difference between null and undefined?

Ans: **null**: Means "no value" and is something you set intentionally. It’s like saying “I want this variable to have no value.”

undefined: Means a variable exists but has not been given a value yet. It’s the default state for variables that haven't been initialized.

Q24. What is eval? Give an example.

Ans: eval() is a JavaScript function that takes a string and runs it as if it were code. It can be useful for evaluating expressions or dynamically creating functions, but it should be used cautiously due to potential security risks and performance issues.

- Example:

```
const x = 10;

const y = 20;


// Using eval to evaluate an expression

const result = eval("x + y"); // Evaluates the expression as if it's JavaScript code

console.log(result); // Outputs: 30
```

Q25. What is the difference between window and document?

Ans: **window**: The whole browser window or tab; it gives you access to things like alerts, the size of the window, and the browser's history.

- **document**: The actual web page content; it lets you change text, create new elements, and interact with everything displayed in the browser.

Q26. What is NaN property?

Ans: NaN means "Not-a-Number" and is used in JavaScript to indicate that a value is not a valid number.

It usually appears when a mathematical operation goes wrong or when you try to convert something that isn’t a number. You can check if a value is NaN using isNaN() or Number.isNaN().

Q27. Write a function that would allow you to do this:

Ans:

```
function multiply(a) {
   return function(b) {
      return a * b;
   };
}
```

// Example usage

const result = multiply(5)(6); // Outputs: 30

console.log(result); // Outputs: 30

Q28. What will the following code output?

Ans The code 0.1 + 0.2 === 0.3 will output false.

Q29. Define ways to empty an array in JavaScript.

Ans In JavaScript, there are several ways to empty an array. Here are some common methods:

1. Setting Length to Zero

2. Using the splice() Method

3. Reassigning to a New Array

4. Using the pop() Method in a Loop

5. Using the shift() Method in a Loop

6. Using the fill() Method

Q30. Write a one-liner function in JavaScript to remove duplicate elements from an

Array.

Ans: const removeDuplicates = arr => [...new Set(arr)];

Q31. What will be the output of the below code:

```
const array = [10, 20, 30, 40];

    const result = array.map((num) => num / 2).filter((num) => num >= 15);

    console.log(result);
```

Ans: The code divides each number in the array by 2, then keeps only the numbers that are greater than or equal to 15. The final output will be [15, 20].

Q32. Find the issue with the below code snippet:

Ans: **setTimeout**: The function schedules a message to be printed after 3 seconds (3000 milliseconds).

- **clearTimeout**: In this case, clearTimeout is mentioned, but it's not invoked correctly. Normally, clearTimeout is used to stop a scheduled timeout, but it requires a reference to the timeout ID, like this: clearTimeout(timeoutID). Since clearTimeout is not properly called, it has no effect.
-

React & JS Questions Entry Level 3

Q33. What will happen if you try to access a variable before it is declared?

Ans: If you access a variable before it is declared:

- **Var** eturns undefined due to hoisting.
- **let and const** throws a ReferenceError because of the Temporal Dead Zone (TDZ).

Q34. Can you explain how the spread operator works in JavaScript?

Ans: The spread operator takes the elements of an array or object and spreads them into individual elements, making it easier to combine, copy, or pass them around.

In Arrays:

- You can expand an array into individual elements.
  const arr1 = [1, 2, 3];
  const arr2 = [...arr1, 4, 5]; // Spreads arr1 into arr2
  console.log(arr2); // Outputs: [1, 2, 3, 4, 5]

In Objects:

- You can copy properties from one object to another or merge objects.

  const obj1 = { a: 1, b: 2 };
  const obj2 = { ...obj1, c: 3 }; // Spreads obj1 into obj2
  console.log(obj2); // Outputs: { a: 1, b: 2, c: 3 }

In Function Calls:

- You can pass array elements as individual arguments to a function.

- const nums = [10, 20, 30];
  console.log(Math.max(...nums)); // Outputs: 30

Q35. What is the output of the following code?

Ans: (arr) is not defined

Q36. How does the useMemo hook optimize performance in React?

Give an example.

Ans: useMemo prevents expensive calculations from running on every render by caching the result until the inputs change, making the component more efficient and faster.

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

  function Example() {
    const [count, setCount] = useState(0);
    const [input, setInput] = useState("");

    // Expensive calculation: just returns count * 2
    const doubleCount = useMemo(() => {
      console.log("Calculating...");
      return count * 2;
    }, [count]); // Only recalculates when `count` changes

    return (
      <div>
        <h1>Double Count: {doubleCount}</h1>
        <button onClick={() => setCount(count + 1)}>Increase Count</button>
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          placeholder="Type something"
        />
      </div>
    );
  }

  export default Example;
  ```
- **useMemo** is used to memoize the result of count * 2. It only recalculates when count changes.
- The input state is unrelated to the calculation, so typing in the input won't trigger the recalculation of doubleCount.

Q37. What will be the output of the following React component?

Ans: The component will render an empty element, with no content inside it.

Q38. How would you describe the useEffect hook in React?

Ans: The useEffect hook let us perform actions like fetching data or setting up timers after a component renders, and we can control when it runs based on the dependencies we provide. we can also clean up resources when the component unmounts or before the effect runs again.

Q39. What is the output of the following code?

Ans: The array is emptied, so the output is an empty array: [].

40. What is the difference between functional and class components in React?

Ans: **Functional Components**: Simpler and now fully capable of managing state and side effects with Hooks. They are preferred for most new development.

- **Class Components**: More traditional approach, still in use but often seen as more complex and verbose.