React part1

📚 Section 1: Basic Concepts

Q1. What is React?

Answer:
React is a JavaScript library created by Facebook in 2013 for building user interfaces (UI). It is mainly used for creating single-page applications where content changes dynamically without reloading the page.

Key Points:

- It's a library, not a framework (lighter than Angular)
- Used for building interactive user interfaces
- Makes it easy to create reusable components
- Very popular - used by Facebook, Instagram, Netflix, Airbnb

Simple Example:
Think of React like building with LEGO blocks. Each block (component) can be reused anywhere in your application. You build small pieces and combine them to create a full application.

Q2. What are the main features of React?

Answer:
React has several important features:

1. JSX (JavaScript XML): Allows you to write HTML-like code in JavaScript
2. Components: Reusable pieces of UI code
3. Virtual DOM: Makes updates fast and efficient
4. One-way Data Binding: Data flows in one direction (parent to child)
5. Declarative: You describe what UI should look like, React handles the updates
6. React Hooks: Special functions to use React features

Remember: These features make React fast, easy to understand, and easy to maintain.

Q3. What is JSX?

Answer:
 JSX stands for JavaScript XML. It allows us to write HTML-like code inside JavaScript files. It makes the code easier to read and write.

```
// JSX Example
const element = <h1>Hello, World!</h1>;
 // This looks like HTML but it's actually JSX
// Behind the scenes, it converts to JavaScript
```

Why use JSX?

- More readable than pure JavaScript
- Shows the UI structure clearly
- Prevents injection attacks (safer)
- Can use JavaScript expressions inside curly braces

Important: JSX is not mandatory, but 99% of React developers use it because it's much easier to work with.

Q4. What is a Component in React?

Answer:
 A component is a reusable piece of code that returns HTML (JSX). It's like a JavaScript function that produces part of your webpage.

Two Types of Components:

1. **Functional Components:** Simple functions that return JSX

```
// Functional Component
 function Welcome() {
 return<h1>Hello, Welcome!</h1>;
}
```

3. **Class Components:** ES6 classes that extend React.Component (older way)

```
// Class Component
class Welcome extendsReact.Component {
render() {
    return<h1>Hello, Welcome!</h1>;
  } }
```

Current Practice: Use Functional Components with Hooks (modern and recommended way).

Q5. What is the difference between Functional and Class Components?

Functional Components	Class Components
Simple JavaScript functions	ES6 classes
Less code, easier to read	More code, complex
Use Hooks for state	Use this.state
No 'this' keyword	Must use 'this' keyword
Better performance	Slightly slower
Modern approach (recommended)	Old approach

Tip: If you're starting React now, learn Functional Components with Hooks. Class components are becoming outdated.

Q6. What is the difference between Element and Component?

Answer:

Element: A plain object describing what you want to see on screen

```
// Element
 const element = <h1>Hello!</h1>;
```

Component: A function or class that can accept input and returns an Element

```
// Component
function Welcome() {
return<h1>Hello!</h1>;
 }
```

Simple Difference: Element is what you see. Component is a factory that creates Elements.

🎯 Section 2: Props and State

Q7. What are Props?

Answer:
 Props (short for properties) are used to pass data from parent component to child component. They work like function parameters.

Key Points about Props:

- Props are read-only (cannot be changed by child)
- Passed from parent to child
- Can pass any data: strings, numbers, objects, functions
- Make components reusable

```
// Parent Component
function App() {
 return<Welcomename="John"age={25} />;

}
// Child Component
function Welcome(props) {
 return<h1>Hello {props.name}, you are {props.age} years old</h1>;
 }
```

Remember: Props flow down (parent to child), never up.

Q8. What is State?

Answer:

State is data that changes over time in a component. When state changes, the component re-renders to show the updated data.

Key Points about State:

- State is managed within the component
- State can be changed (mutable)
- When state changes, component re-renders
- Use `useState` Hook in functional components

```
import { useState } from'react';
function Counter() {
const [count, setCount] = useState(0);
return (
<div>
<p>Count: {count}</p>
<button onClick={() => setCount(count + 1)}> Increase </button> </div>
);
 }
```

Q9. What is the difference between Props and State?

Props	State
Passed from parent to child	Managed within component
Read-only (immutable)	Can be changed (mutable)
Cannot be modified by child	Can be modified using setState
Used for communication	Used for dynamic data
Accessed as props.name	Accessed as state.name

Easy Way to Remember:
Props are like function parameters (you receive them).
State is like variables inside function (you manage them).

Q10. Can we change Props?

Answer:
No, we cannot change props in the child component. Props are read-only.

Why? Because React follows "one-way data flow". Data should flow from parent to child only.

If you need to change data:

- Pass the value as props
- Pass a function as props to modify in parent
- Or use state instead of props

```
// ❌ Wrong Way
function Child(props) {
 props.name = "New Name";
 // This will cause error
}

 // ✅ Right Way
function Parent() {
const [name, setName] = useState("John");
return <Child name={name} onChangeName={setName} />;
}
function Child(props) {
return (
<button onClick={() => props.onChangeName("New Name")}> Change Name </button>
 );
}
```

🔄 Section 3: React Hooks

Q11. What are Hooks?

Answer:
Hooks are special functions that let you use React features (like state and lifecycle) in functional components. They were added in React 16.8 (2019).

Popular Hooks:

- `useState` - Manage state
- `useEffect` - Side effects and lifecycle
- `useContext` - Access context
- `useRef` - Access DOM elements

- `useMemo` - Optimize performance
- `useCallback` - Memoize functions
- `useReducer` - Complex state management

Rules of Hooks:

1. Only call Hooks at the top level (not inside loops, conditions, or nested functions)
2. Only call Hooks from React functional components or custom Hooks

---

Q12. What is useState Hook?

Answer:
`useState` is a Hook that lets you add state to functional components. It returns an array with two elements:

1. Current state value
2. Function to update the state

Syntax:

```
const [state, setState] = useState(initialValue);
```

Example:

```
import { useState } from'react';
function Example() {
const [count, setCount] = useState(0);
 const [name, setName] = useState("John");
const [isActive, setIsActive] = useState(true);

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

Important:
❌ `count = count + 1` (Wrong)
✅ `setCount(count + 1)` (Correct)

Q13. What is useEffect Hook?

Answer:
`useEffect` is a Hook for performing side effects in functional components. Side effects are operations that affect things outside the component, like:

- Fetching data from API
- Setting up subscriptions
- Manually changing the DOM
- Timers (`setTimeout`, `setInterval`)

Syntax:

```
useEffect(() => {
 // Code to run after render
return() => {
// Cleanup code (optional)
 };
}, [dependencies]);
```

Example:

```
import { useState, useEffect } from'react';
function Example() {
const [count, setCount] = useState(0);

useEffect(() => {

 document.title = `Count: ${count}`;
}, [count]); // Only re-run when count changes

return (
<button onClick={() => setCount(count + 1)}> Click: {count} </button> );
}
```

Q14. What is the dependency array in useEffect?

Answer:
The dependency array is the second parameter of `useEffect`. It controls when the effect runs.

Three Cases:

```
// 1. No dependency array: Runs after every render
useEffect(() => {
 console.log('Runs after every render');
 });

 // 2. Empty array []: Runs only once (on mount)
useEffect(() => {
console.log('Runs only once when component mounts');
}, []);

 // 3. With dependencies [a, b]: Runs when 'a' or 'b' changes
useEffect(() => {
 console.log('Runs when count changes');
 }, [count]);
```

Common Mistake: Forgetting to add dependencies can cause bugs. React will warn you if you miss a dependency.

Q15. What is useContext Hook?

Answer:
`useContext` lets you access data from React Context without wrapping components in `Context.Consumer`.

Example:

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

// Create context
constThemeContext = createContext('light');

// Parent component
function App() {
return (
 <ThemeContext.Providervalue="dark"> <Toolbar /> </ThemeContext.Provider> );
 }

// Child component
 function Toolbar() {
 const theme = useContext(ThemeContext);

return <div>Current theme: {theme}</div>;
 }
```

Use Case: Perfect for global data like theme, user info, language settings.

Q16. What is useRef Hook?

Answer:
`useRef` returns a mutable object that persists for the lifetime of the component.

1️⃣ Access DOM elements directly:

```
import { useRef } from'react';
 function TextInput() {
 const inputRef = useRef(null);

constfocusInput = () => {
 inputRef.current.focus();
 };
return (
<>
<inputref={inputRef}type="text" />
 <button onClick={focusInput}>Focus Input</button> </> );
 }
```

2️⃣ Store values that don't trigger re-render:

```
function Timer() {
const countRef = useRef(0);

 const increment = () => {
 countRef.current = countRef.current + 1;
 console.log(countRef.current); // Won't re-render
  }; }
```

Key Difference: Changing `useRef` value doesn't cause re-render. Changing `useState` does.

Q17. What is useMemo Hook?

Answer:
`useMemo` memoizes (remembers) the result of an expensive calculation.

```
import { useState, useMemo } from'react';
 function ExpensiveComponent({ numbers })
 {
 const [count, setCount] = useState(0);
 const sum = useMemo(() => {
 console.log('Calculating sum...');

return numbers.reduce((a, b) => a + b, 0);
}, [numbers]);

 return (
<div>
 <p>Sum: {sum}</p> <p>Count: {count}</p>
<button onClick={() => setCount(count + 1)}> Increase Count </button>
 </div> ); }
```

When to use: Only for expensive calculations.

Q18. What is useCallback Hook?

Answer:
`useCallback` returns a memoized version of a callback function.

```
import { useState, useCallback } from'react';
function Parent() {
 const [count, setCount] = useState(0);
const handleClick = useCallback(() =>
{
console.log('Current count:', count);
}, [count]);

 return<Child onClick={handleClick} />;
 }
```

Difference between useMemo and useCallback:

- useMemo: Returns a memoized **value**
- useCallback: Returns a memoized **function**

```
// These are equivalent
useCallback(fn, deps)
useMemo(() => fn, deps)
```

Q19. What is useReducer Hook?

Answer:
`useReducer` is used for complex state logic, similar to Redux reducers.

```
import { useReducer } from'react';
// Reducer function
 function reducer(state, action) {
 switch (action.type) {
 case'increment': return { count: state.count + 1 };
 case'decrement': return { count: state.count - 1 };
 default: return state; }
}

 function Counter() {
 const [state, dispatch] = useReducer(reducer, { count: 0 });

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

Q20. What are Custom Hooks?

Answer:
Custom Hooks are JavaScript functions that start with `"use"` and can call other Hooks.

```
import { useState } from'react';
// Custom Hook
function useInput(initialValue) {
const [value, setValue] = useState(initialValue);
 const handleChange = (e) => { setValue(e.target.value); };

return [value, handleChange];
 }
// Using Custom Hook

function Form() {
```

```
const [name, handleNameChange] = useInput('');

 const [email, handleEmailChange] = useInput('');

return (
<form>
<inputvalue={name}onChange={handleNameChange} />
<inputvalue={email}onChange={handleEmailChange} />
</form> );
}
```

Benefits:

- Reuse logic across components
- Keep components clean and simple
- Easy to test

Q21. What is the difference between useMemo and useCallback?

useMemo	useCallback
Returns memoized **VALUE**	Returns memoized **FUNCTION**
For expensive calculations	For callback functions
`const value = useMemo(() => compute(), [])`	`const fn = useCallback(() => {}, [])`
Caches result of function	Caches function itself

```
// useMemo example
 const sum = useMemo(() => a + b, [a, b]);

// useCallback example

const handleClick = useCallback(() => {
 console.log('clicked');
}, []);
```

Q22. When should we use useEffect cleanup function?

Answer:
Use cleanup function to clean up side effects before component unmounts or before re-running the effect.

Example - Timer Cleanup:

```
import { useEffect, useState } from'react';
 function Timer() {
 const [seconds, setSeconds] = useState(0);
 useEffect(() => {
 const interval = setInterval(() => { setSeconds(s => s + 1); }, 1000);

 return() => {
clearInterval(interval); // Stop timer
 };
 }, []);

 return<div>Seconds: {seconds}</div>;
}
```