React part2

Q23. What is Virtual DOM?

Answer: Virtual DOM is a lightweight JavaScript copy of the real DOM kept in memory. React uses it to make updates faster.

How it works:

1. React keeps a virtual copy of UI in memory
2. When something changes, React creates new Virtual DOM
3. React compares new with old (Diffing)
4. React updates only changed parts in real DOM

Real-Life Example: Like editing a Word document - you only reprint the pages that changed, not the entire document!

Q24. What is Real DOM vs Virtual DOM?

Real DOM	Virtual DOM
Updates are slow	Updates are fast
Directly updates HTML	Updates JSX
DOM manipulation expensive	DOM manipulation easy
Memory wastage	No memory wastage

Why Virtual DOM is faster: Real DOM updates involve repainting and reflow (slow). Virtual DOM batches updates and minimizes changes.

Q25. How does React update the UI?

Answer: React follows 4 steps:

1. **Trigger** - State or props change
2. **Render** - React creates new Virtual DOM
3. **Reconciliation** - Compare new vs old

4. **Commit** - Update only changed parts

Example:

javascript

```
function Counter(){
const[count, setCount]=useState(0);
// When button clicks:
// 1. State changes (count becomes 1)
// 2. New Virtual DOM created
// 3. React compares old vs new
// 4. Updates only the number
return(
     <div>
    <p>Count:{count}</p>
     <button onClick={()=>setCount(count +1)}>Increase</button>
</div>);}
```

Q26. What is Reconciliation?

Answer: Reconciliation is React's process to update DOM efficiently by comparing two Virtual DOM trees.

How it works:

- **Different Types:** Destroy old, create new
- **Same Types:** Keep node, update attributes only
- **Lists:** Use keys to match items

Example:

javascript

```
// Old Virtual DOM
<div className="old"><span>Hello</span></div>

// New Virtual DOM
<div className="new"><span>Hello</span></div>

// React only updates className, not entire div
```

Q27. What is Diffing Algorithm?

Answer: Diffing is how React compares two Virtual DOM trees.

Rules:

1. Different Types - Replace completely:

javascript

```
// Before
<div>Content</div>

// After
<span>Content</span>// Result: div destroyed, span created
```

2. Same Type - Update attributes only:

javascript

```
// Before
<div className="old">Content</div>

// After
<div className="new">Content</div>// Result: Only className updated
```

3. Lists with Keys:

javascript

```
// React knows which items changed
<li key="1">Item1</li>
<li key="2">Item2</li>
```

Performance: O(n) time - very fast!

Q28. What are Keys in React?

Answer: Keys are special attributes for list elements. They help React identify which items changed.

Why important:

- Optimize rendering
- Maintain component state
- Improve performance

❌ **Bad Example:**

javascript

```
const items =['Apple','Banana'];// Bad - using index
items.map((item, index)=>(

 <li key={index}>{item}</li>

))
```

✅ **Good Example:**

javascript

```
const items =[
{id:1,name:'Apple'},
{id:2,name:'Banana'}
];
// Good - using unique id
items.map(item=>(
   <li key={item.id}>{item.name}</li>
))
```

Q29. Why not use index as key?

Answer: Using index causes problems when list items are reordered, added, or removed.

Problem Example:

javascript

```
// Initial list
[{index:0,name:'Apple'},// key = 0
{index:1,name:'Banana'},// key = 1
```

```
    {index:2,name:'Orange'} // key = 2

    ]
    // After removing Banana
    [{index:0,name:'Apple'},// key = 0 (same)
    {index:1,name:'Orange'}// key = 1 (was 2!)
     ]// React thinks Orange is Banana!
```

When index is OK:

- Static list (never changes)
- Never reordered
- No unique IDs available

Q30. What is React Fiber?

Answer: React Fiber is the new reconciliation engine in React 16+.

Goals:

- Split work into chunks
- Pause and resume work
- Assign priority to updates
- Abort unnecessary work

Simple Explanation: Before Fiber, React blocked browser while updating. Now React can pause to handle user input - smoother apps!

🔄 Section 5: Component Lifecycle

Q31. What is Component Lifecycle?

Answer: Lifecycle = phases a component goes through from creation to removal.

Three Phases:

1. Mounting (Component created)

- constructor()
- render()
- componentDidMount()

2. Updating (Component re-renders)

- render()
- componentDidUpdate()

3. Unmounting (Component removed)

- componentWillUnmount()

Note: In functional components, use **useEffect** for all lifecycle!

Q32. How to implement lifecycle in Functional Components?

Answer: Use useEffect Hook.

1. componentDidMount (runs once):

javascript

```
useEffect(()=>{

console.log('Component mounted');// Fetch data, add listeners

},[]);// Empty array = run once
```

2. componentDidUpdate (runs on updates):

javascript

```
useEffect(()=>{

console.log('Count changed');// Runs when count changes

},[count]);// Run when count changes
```

3. componentWillUnmount (cleanup):

javascript

```
useEffect(()=>{
console.log('Mounted');
return()=>{

console.log('Will unmount');// Cleanup: remove listeners

};},[]);
```

Complete Example:

javascript

```
import{ useState, useEffect }from'react';
function UserProfile({ userId })
{
const[user, setUser]=useState(null);

useEffect(()=>{
// Mount: Fetch user
fetch(`/api/users/${userId}`)
.then(res=> res.json())
.then(data=>setUser(data));// Unmount: Cleanup

return()=>{
console.log('Cleaning up');
};},[userId]);// Re-run when userId changes

return <div>{user?.name}</div>;
}
```

Q33. componentDidMount vs useEffect?

componentDidMount	useEffect
Class components only	Functional components only
Runs once after first render	Can run on every render or specific updates
Separate method	Single Hook for all lifecycle
No cleanup in same method	Cleanup in return function

Example:

javascript

```
// Class Component
class MyComponent extends React.Component{
componentDidMount(){
console.log('Mounted');}}// Functional Component (same thing)
```

```
function MyComponent(){
useEffect(()=>{console.log('Mounted');
},[]);}
```

⚡ Section 6: Events in React

Q34. How to handle events in React?

Answer: Use camelCase naming (onClick, onChange) and pass functions.

Differences from HTML:

- React: `onClick` (camelCase)
- HTML: `onclick` (lowercase)
- Pass function, not string

Example:

javascript

```
// HTML way ❌
<button onclick="handleClick()">Click</button>

// React way ✅

<button onClick={handleClick}>Click</button>
```

Complete Example:

javascript

```
function Button(){
const handleClick=(event)=>{

event.preventDefault();// Stop default

console.log('Clicked!');
console.log(event.target);// Get element
};
return(
<button onClick={handleClick}>ClickMe</button>
);}
```

Q35. What is Synthetic Event?

Answer: Synthetic Event is React's cross-browser wrapper around native browser events.

Benefits:

- Works same in all browsers
- Better performance
- Same API everywhere

Common Properties:

javascript

```
function handleEvent(event){

 event.type;// Event type (click, change)

 event.target;// Element that triggered

event.currentTarget;// Element with handler

 event.preventDefault();// Prevent default

event.stopPropagation();// Stop bubbling}
```

Q36. How to pass parameters to event handlers?

Answer: Three methods:

Method 1: Arrow Function (inline):

javascript

```
function ItemList(){
const handleClick=(id)=>{
console.log('Item ID:', id);
};

return(
<button onClick={()=>handleClick(123)}>Click</button>
```

```
8    );}
```

Method 2: bind():

javascript

```
1    <button onClick={handleClick.bind(this,123)}>Click</button>
```

Method 3: Currying (recommended):

javascript

```
1    function ItemList(){
2    // Returns a function
3    const handleClick=(id)=>(event)=>{
4
5    console.log('Item ID:', id);
6    };
7    return(
8    <button onClick={handleClick(123)}>Click</button>
9    );}
```

Q37. What is Event Bubbling?

Answer: Event on child "bubbles up" to parents.

Example:

javascript

```
1    function App(){
2    const handleParent=()=>{
3    console.log('Parent clicked');
4    };
5
6    const handleChild=(e)=>{
7    console.log('Child clicked');// e.stopPropagation();
8     // Stop bubbling
9    };
10
11   return(
```

```
<div onClick={handleParent}>Parent
<button onClick={handleChild}>Child</button>
</div>
);}
// Click button output:
// "Child clicked"
// "Parent clicked"
```

Stop Bubbling:

javascript

```
const handleChild=(e)=>{

e.stopPropagation();// Stop!

console.log('Only child');};
```

📝 Section 7: Forms and Controlled Components

Q38. What are Controlled Components?

Answer: Form elements whose values are controlled by React state.

Characteristics:

- Value controlled by state
- Changes handled by event handlers
- State change = re-render

Example:

javascript

```
import{ useState }from'react';
function Form(){

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

const handleSubmit=(e)=>{
 e.preventDefault();
alert(`Name: ${name}`);};

return(
```

```
<form onSubmit={handleSubmit}>
<input  type="text" value={name}// Controlled by state
 onChange={(e)=>setName(e.target.value)}// Update state
/>
<button>Submit</button>
</form>
);
}
```

Q39. What are Uncontrolled Components?

Answer: Form elements that store their own state in DOM. Use refs to access values.

Example:

javascript

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

function Form(){
const nameRef =useRef();// Create ref

const handleSubmit=(e)=>{
e.preventDefault();
alert(`Name: ${nameRef.current.value}`);// Get value via ref
};

return(
<form onSubmit={handleSubmit}>
<input  type="text" ref={nameRef}// Attach ref
 defaultValue="Sid"// Default value
/>
<button>Submit</button>
</form>);
}
```

Q40. Controlled vs Uncontrolled

Feature / Aspect	**Controlled Component**	**Uncontrolled Component**
Who controls the value	React (through state)	DOM (browser)

How value is accessed	Using `useState` (state variable)	Using `useRef` (direct DOM reference)
Value prop	Uses `value` prop to set input value	Uses `defaultValue` prop for initial value
Data flow	One-way data binding (React → UI)	Data lives in DOM until accessed
Example	`<input value={name} onChange={handleChange} />`	`<input defaultValue="John" ref={inputRef} />`
Form validation	Easier — validation can happen in React state	Harder — need to read DOM value manually
Code length	More code but more control	Less code, less control
Performance	Slightly slower (due to re-renders)	Slightly faster (no re-renders)
React-way or HTML-way	✅ More React way	More traditional HTML way
Recommended for	Forms needing validation, live updates	Simple or small forms

When to use:

- **Controlled:** Complex forms, validation
- **Uncontrolled:** Simple forms, file uploads

Q41. How to handle multiple inputs?

Answer: Use single onChange with computed property names.

Example:

javascript

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

function Form(){
// Single state object for all inputs

const[formData, setFormData]=useState(
{name:'',
email:'',
age:''});

const handleChange=(e)=>{
```

```
const{ name, value }= e.target;

setFormData(prevData=>({...prevData,// Keep other values

[name]: value // Update only this field
}));};
return(
<form>
<input  name="name" // Must match state key
 value={formData.name} onChange={handleChange}// Same handler for all
 placeholder="Name"/>

<input  name="email" value={formData.email} onChange={handleChange} placeholder="Email"/>

<input  name="age" value={formData.age} onChange={handleChange} placeholder="Age"/>

</form>);
}
```

Q42. How to handle form validation?

Answer: Manual validation is most common.

Example:

javascript

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

function Form(){
const[email, setEmail]=useState('');
const[error, setError]=useState('');

const handleSubmit=(e)=>{
 e.preventDefault();// Validation checks
if(!email){
setError('Email is required');

return;
}
if(!email.includes('@')){
setError('Invalid email');
return;
}
// Valid - clear error and submit
setError('');

console.log('Form submitted:', email);
};
return(
```

```
<form onSubmit={handleSubmit}>

<input  type="email" value={email} onChange={(e)=>setEmail(e.target.value)} placeholder="
{
/* Show error if exists */
}{
error &&<p style={{color:'red'}}>
{error}</p>
}
<button type="submit">Submit</button>

</form>
);}
```

HTML5 Validation:

javascript

```
<input  type="email" required // Must fill

minLength={5}// Min 5 chars

maxLength={50}// Max 50 chars
/>
```

Q43. onChange vs onInput?

onChange	onInput
React's normalized event	Native DOM event
Works for all inputs	Only text inputs
✅ Recommended	Rarely used
Fires on value change	Fires on input

Best Practice: Always use `onChange` in React.

Q44. How to handle file upload?

Answer: Use uncontrolled component (ref) because file values cannot be set programmatically.

Example:

javascript

```javascript
import{ useState }from'react';
function FileUpload(){
const[file, setFile]=useState(null);
const[preview, setPreview]=useState('');

const handleFileChange=(e)=>{
const selectedFile = e.target.files[0];// Get first file
setFile(selectedFile);// Create image preview

if(selectedFile && selectedFile.type.startsWith('image/'))
{const reader =newFileReader();

reader.onload end=()=>{
setPreview(reader.result);// Set preview
};
reader.readAsDataURL(selectedFile);}};

const handleSubmit=(e)=>{

e.preventDefault();// Create form data
const formData =newFormData();
 formData.append('file', file);// Send to server

fetch('/upload',{method:'POST',body: formData })
.then(res=> res.json())
.then(data=>console.log('Success:', data))
.catch(err=>console.error('Error:', err));};

return(

<form onSubmit={handleSubmit}>
<input  type="file" onChange={handleFileChange} accept="image/*"// Only images
/>
{/* Show preview */
}
{preview &&<img src={preview} alt="Preview" width="200"/>}

<button type="submit">Upload</button>
</form>
);}
```