Here's a structured format for your React Hooks overview notes:

---

## React Hooks Overview

### Introduction

- \*\*Purpose\*\*: Hooks enable function components to manage state and access other React features, reducing the need for class components.

### Key Rules for Using Hooks

1. \*\*Function Components Only\*\*: Hooks must be used inside React function components, not in regular JavaScript functions or class components.

2. \*\*Top-Level Calls\*\*: Hooks should be called at the top level of the component, not inside loops, conditions, or nested functions.

3. \*\*No Conditional Usage\*\*: Hooks must be called in the same order on every render. Avoid calling them conditionally.

---

### React `useState` Hook

1. \*\*Importing useState\*\*

```javascript

import React, { useState } from 'react';

```

2. \*\*Basic Syntax\*\*

```javascript

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

```

- `state`: Current state value.

- `setState`: Function to update the state.

- `initialValue`: Initial state value (can be any data type).

3. \*\*Example: Counter\*\*

```javascript

import React, { useState } from 'react';

function Counter() {

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

const increment = () => setCount(count + 1);

const decrement = () => setCount(count - 1);

return (

<div>

<p>Current Count: {count}</p>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

export default Counter;

```

4. \*\*Handling State Updates\*\*

- State updates are asynchronous.

- Use `setState` to update state, never modify state directly.

5. \*\*Updating State Based on Previous State\*\*

```javascript

setState(prevState => prevState + 1);

```

6. \*\*Lazy Initialization\*\*

```javascript

const [state, setState] = useState(() => {

// Expensive computation

return computedValue;

});

```

7. \*\*Multiple useState Calls\*\*

```javascript

function Form() {

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

}

```

8. \*\*Using useState with Objects and Arrays\*\*

- \*\*Objects\*\*: Manually merge properties using the spread operator:

```javascript

const [user, setUser] = useState({ name: '', age: '' });

setUser(prevUser => ({ ...prevUser, age: 30 }));

```

- \*\*Arrays\*\*: Handle array updates manually:

```javascript

const [items, setItems] = useState([]);

setItems([...items, newItem]); // Adding an item

```

9. \*\*Common Pitfalls\*\*

- \*\*Re-rendering Issues\*\*: Avoid storing derived data in state.

- \*\*Infinite Loops\*\*: Be cautious with side effects that depend on state.

10. \*\*Summary\*\*

- `useState` provides state management for functional components, returning the current state and a function to update it.

---

### React `useEffect` Hook

1. \*\*Basic Usage\*\*

```javascript

useEffect(() => {

// Effect code here

return () => {

// Cleanup code here (optional)

};

}, [dependencies]);

```

- \*\*Effect Function\*\*: Runs after the component renders.

- \*\*Cleanup Function\*\*: Optionally returned to clean up resources.

2. \*\*Example\*\*

```javascript

import React, { useEffect } from 'react';

function ExampleComponent() {

useEffect(() => {

document.title = 'Hello, World!';

}, []); // Runs once on mount

return <div>Hello, World!</div>;

}

```

3. \*\*How It Works\*\*

- \*\*Effect Function\*\*: Runs after the component renders.

- \*\*Cleanup Function\*\*: Optionally returned to clean up resources.

4. \*\*Dependency Array\*\*

- \*\*No Dependency Array\*\*: Effect runs after every render.

- \*\*Empty Array (`[]`)\*\*: Effect runs once after initial render.

- \*\*Specific Dependencies\*\*: Effect runs when specified dependencies change.

5. \*\*Example with Dependencies\*\*

```javascript

useEffect(() => {

console.log('This runs when `count` changes');

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

```

6. \*\*Common Use Cases\*\*

- \*\*Fetching Data\*\*:

```javascript

useEffect(() => {

async function fetchData() {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log(data);

}

fetchData();

}, []);

```

- \*\*Subscribing to Events\*\*:

```javascript

useEffect(() => {

const handleResize = () => console.log('Window resized');

window.addEventListener('resize', handleResize);

return () => window.removeEventListener('resize', handleResize);

}, []);

```

- \*\*Updating Document Title\*\*:

```javascript

useEffect(() => {

document.title = `You clicked ${count} times`;

}, [count]); // Runs whenever `count` changes

```

7. \*\*Best Practices\*\*

- \*\*Avoid Unnecessary Dependencies\*\*: Include only necessary dependencies.

- \*\*Handle Cleanup Properly\*\*: Always clean up side effects.

- \*\*Use Multiple useEffect Hooks\*\*: Separate concerns into different hooks.

8. \*\*Advanced Concepts\*\*

- \*\*Conditionally Firing Effects\*\*: Use conditional logic inside the effect function if needed.

- \*\*Performance Optimization\*\*: Use `React.memo`, `useCallback`, and `useMemo` to optimize performance.

- \*\*React Strict Mode\*\*: Effects might run multiple times in development mode for debugging.

9. \*\*Common Pitfalls\*\*

- \*\*Infinite Loops\*\*: Avoid effects that cause infinite loops by correctly managing dependencies.

- \*\*Stale Closures\*\*: Ensure variables used in effects are included in the dependency array.

- \*\*Async Functions\*\*: Define async functions inside the effect and call them.

10. \*\*Troubleshooting\*\*

- \*\*Debugging\*\*: Use console logs and React DevTools to debug effects.

- \*\*Reactivity Issues\*\*: Ensure accurate dependency tracking.

11. \*\*Summary\*\*

- `useEffect` manages side effects in functional components and replaces lifecycle methods from class components.

---

### React `useMemo` Hook

1. \*\*Purpose\*\*

- `useMemo` helps optimize performance by memoizing the result of expensive computations and preventing unnecessary recalculations.

2. \*\*Syntax\*\*

```javascript

const memoizedValue = useMemo(() => {

// Expensive computation

return computedValue;

}, [dependencies]);

```

- `memoizedValue`: The cached result of the computation.

- The computation only runs when dependencies change.

3. \*\*Example\*\*

```javascript

import React, { useMemo } from 'react';

function ExpensiveComponent({ value }) {

const computedValue = useMemo(() => {

// Expensive calculation

return value \* 2;

}, [value]);

return <div>{computedValue}</div>;

}

```

4. \*\*When to Use\*\*

- Use `useMemo` to optimize performance when dealing with expensive computations or functions that depend on props or state.

5. \*\*Common Pitfalls\*\*

- \*\*Overuse\*\*: Avoid using `useMemo` for simple computations or small components.

---

### React `useCallback` Hook

1. \*\*Purpose\*\*

- `useCallback` memoizes callback functions to prevent unnecessary re-renders of child components that rely on those callbacks.

2. \*\*Syntax\*\*

```javascript

const memoizedCallback = useCallback(() => {

// Callback function

}, [dependencies]);

```

- `memoizedCallback`: The memoized function that only changes when dependencies change.

3. \*\*Example\*\*

```javascript

import React, { useCallback } from 'react';

function ParentComponent() {

const handleClick = useCallback(() => {

console.log('Button clicked');

}, []); // Empty dependency array

return <ChildComponent onClick={handleClick} />;

}

```

4. \*\*When to Use\*\*

- Use `useCallback` when passing callbacks to child components to prevent unnecessary re-renders.

5. \*\*Common Pitfalls\*\*

- \*\*Overuse\*\*: Like `useMemo`, overusing `useCallback` can lead to unnecessary complexity. Use it where it provides a clear benefit in preventing re-renders.

---

### React `useContext` Hook

1. \*\*Purpose\*\*

- `useContext` provides a way to access values from a React context within a function component without needing to wrap the component in a `Consumer`.

2. \*\*Syntax\*\*

```javascript

const value = useContext(MyContext);

```

- `value`: The current context value for `MyContext`.

3. \*\*Example\*\*

```javascript

import React, { useContext } from 'react';

import { MyContext } from './MyContextProvider';

function MyComponent() {

const value = useContext(MyContext);

return <div>{value}</div>;

}

```

4. \*\*When to Use\*\*

- Use `useContext` when you need to access a context value in a function component without adding extra wrapper code with `Consumer`.

5. \*\*Common Pitfalls\*\*

- Ensure the component using `use

Context` is within a matching `Context.Provider` in the component tree.

---

### React `useReducer` Hook

1. \*\*Purpose\*\*

- `useReducer` is an alternative to `useState` for managing more complex state logic. It is particularly useful when the state depends on multiple sub-values or when the next state depends on the previous one.

2. \*\*Syntax\*\*

```javascript

const [state, dispatch] = useReducer(reducer, initialState);

```

- `state`: The current state value.

- `dispatch`: Function to dispatch actions to update the state.

- `reducer`: A function that determines state updates based on actions.

- `initialState`: The initial state value.

3. \*\*Example\*\*

```javascript

import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

throw new Error();

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<div>

<p>Count: {state.count}</p>

<button onClick={() => dispatch({ type: 'increment' })}>+</button>

<button onClick={() => dispatch({ type: 'decrement' })}>-</button>

</div>

);

}

```

4. \*\*When to Use\*\*

- Use `useReducer` when you have complex state logic that involves multiple state sub-values or when different actions affect the state in distinct ways.

5. \*\*Common Pitfalls\*\*

- Avoid using `useReducer` for simple state management where `useState` would be more straightforward.

---

### React `useRef` Hook

1. \*\*Purpose\*\*

- `useRef` creates a mutable reference object that persists across renders. It can be used to access DOM elements directly or to persist values between renders without causing a re-render.

2. \*\*Syntax\*\*

```javascript

const refContainer = useRef(initialValue);

```

- `refContainer`: A reference object with a `.current` property that can hold a value or a DOM element.

3. \*\*Example\*\*

```javascript

import React, { useRef } from 'react';

function TextInputWithFocusButton() {

const inputEl = useRef(null);

const onButtonClick = () => {

inputEl.current.focus();

};

return (

<>

<input ref={inputEl} type="text" />

<button onClick={onButtonClick}>Focus the input</button>

</>

);

}

```

4. \*\*When to Use\*\*

- Use `useRef` when you need to access or modify a DOM element directly or store a mutable value that doesn’t need to trigger a re-render.

5. \*\*Common Pitfalls\*\*

- Avoid using `useRef` to store state that should cause re-renders. For that, use `useState` or `useReducer`.

---

### React `useImperativeHandle` Hook

1. \*\*Purpose\*\*

- `useImperativeHandle` customizes the instance value that is exposed to parent components when using `ref`. It’s typically used with `forwardRef`.

2. \*\*Syntax\*\*

```javascript

useImperativeHandle(ref, createHandle, [dependencies]);

```

3. \*\*Example\*\*

```javascript

import React, { useImperativeHandle, useRef, forwardRef } from 'react';

const FancyInput = forwardRef((props, ref) => {

const inputRef = useRef();

useImperativeHandle(ref, () => ({

focus: () => {

inputRef.current.focus();

}

}));

return <input ref={inputRef} />;

});

function Parent() {

const ref = useRef();

return (

<>

<FancyInput ref={ref} />

<button onClick={() => ref.current.focus()}>Focus the input</button>

</>

);

}

```

4. \*\*When to Use\*\*

- Use `useImperativeHandle` to expose imperative methods from a child component to a parent component.

5. \*\*Common Pitfalls\*\*

- Don’t overuse `useImperativeHandle`; it should only be used when necessary, as it can lead to less declarative code.

---

### React `useLayoutEffect` Hook

1. \*\*Purpose\*\*

- `useLayoutEffect` is similar to `useEffect` but fires synchronously after all DOM mutations. Use it when you need to make DOM changes before the browser paints.

2. \*\*Syntax\*\*

```javascript

useLayoutEffect(() => {

// DOM changes here

}, [dependencies]);

```

3. \*\*Example\*\*

```javascript

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

function LayoutEffectExample() {

const divRef = useRef();

useLayoutEffect(() => {

divRef.current.style.color = 'red';

}, []);

return <div ref={divRef}>This text will be red.</div>;

}

```

4. \*\*When to Use\*\*

- Use `useLayoutEffect` when you need to synchronously manipulate the DOM before the browser has a chance to paint.

5. \*\*Common Pitfalls\*\*

- Avoid overusing `useLayoutEffect` as it can lead to performance issues by blocking the painting process.

---

These hooks, along with `useState`, `useEffect`, `useMemo`, and `useCallback`, provide a comprehensive toolkit for managing state, side effects, and performance in React functional components.

---