Understandin g React Lifecycle Hooks

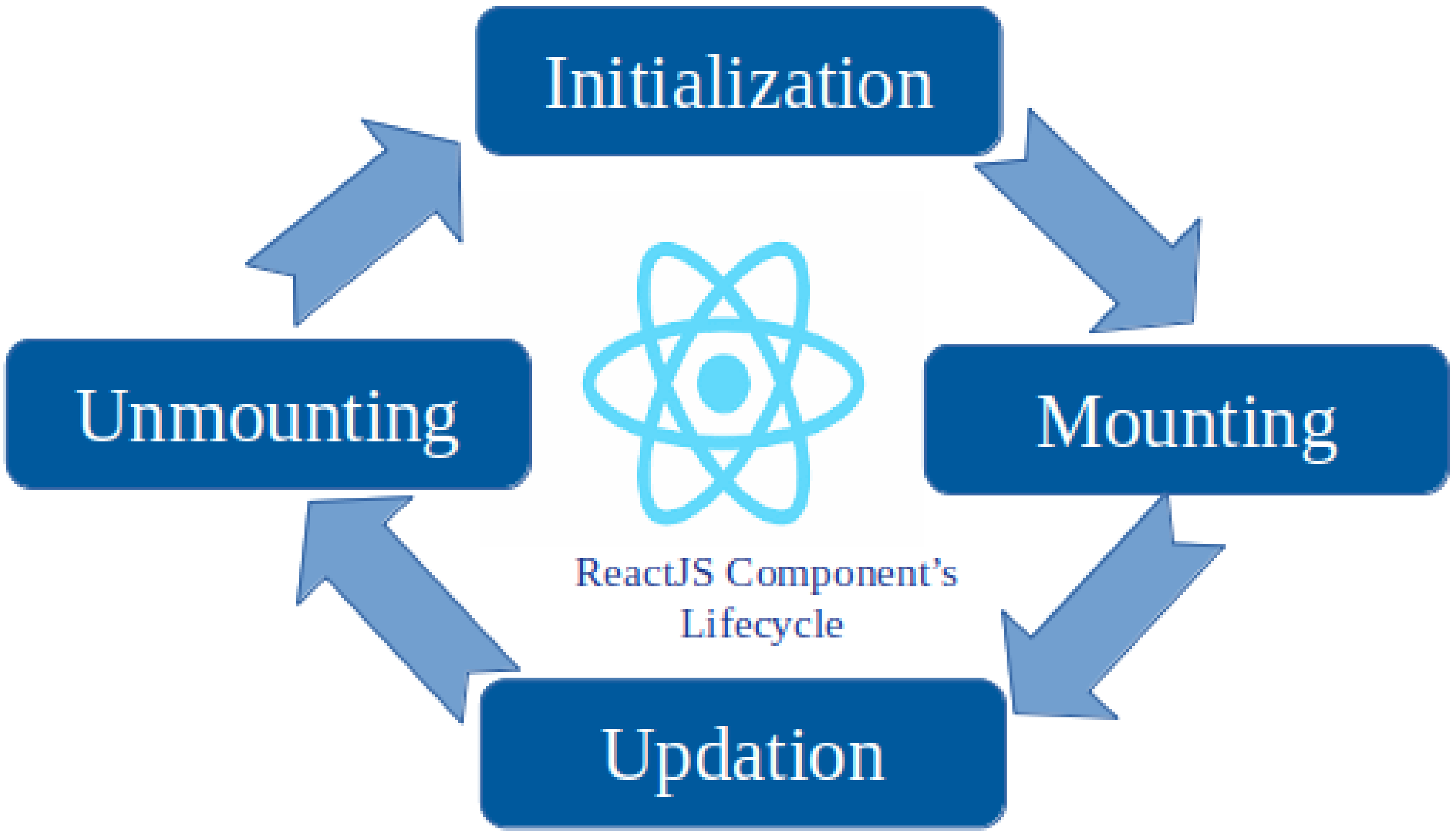
A Deep Dive into Class Components and Functional Components with Hooks

Qinshift Academy

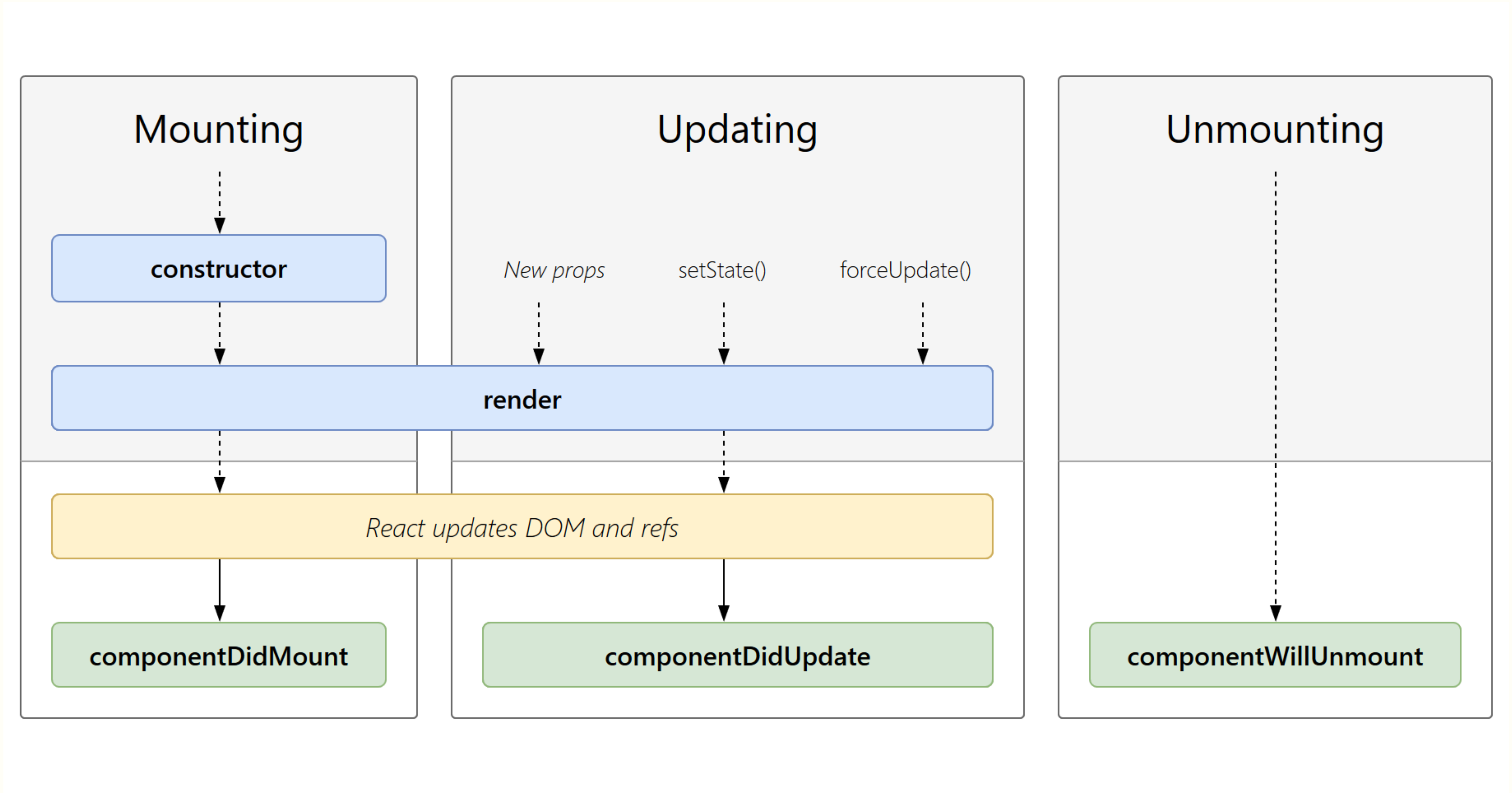
Props vs State

Features	props	state
Can get initial value from parent Component?	Yes	Yes
Can be changed by parent Component?	Yes	No
Can set default values inside Component?	Yes	Yes
Can change inside Component?	No	Yes
Can set initial value for child Components?	Yes	Yes
Can change in child Components?	Yes	No

Component lifecycle



Component lifecycle



Introduction to React Lifecycle

- What are Lifecycle Hooks?
  - Lifecycle hooks are methods that get called at different stages of a component's existence.
  - These hooks provide opportunities to execute code at key moments in the component's lifecycle.
- Lifecycle Stages:
  - Mounting: When the component is being created and inserted into the DOM.
  - Updating: When the component is being re-rendered due to changes in props or state.
  - Unmounting: When the component is being removed from the DOM.

Class Component Lifecycle Hooks

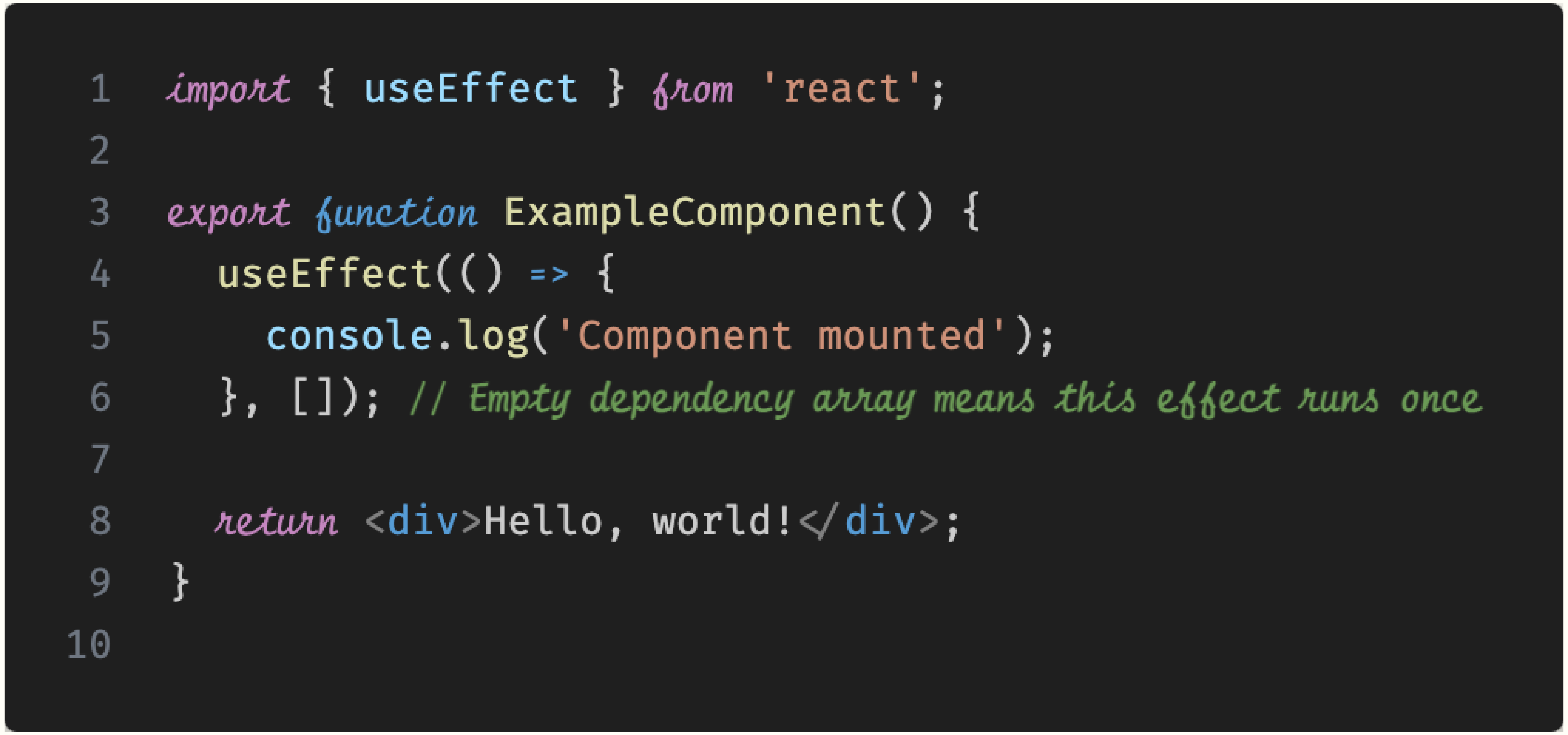
- Mounting:
  - constructor(): Initializes the component's state and binds methods.
  - static getDerivedStateFromProps(): Updates the state based on initial props.
  - render(): Returns the JSX to be rendered in the DOM.
  - componentDidMount(): Invoked immediately after the component is mounted.
- Updating:
  - static getDerivedStateFromProps(): Also called when props change.
  - shouldComponentUpdate(): Determines whether the component should re-render.
  - render(): Called again to update the DOM.
  - getSnapshotBeforeUpdate(): Captures information before the DOM updates.
  - componentDidUpdate(): Invoked immediately after the component updates.
- Unmounting:
  - componentWillUnmount(): Called just before the component is removed from the DOM.

Functional Components with Hooks

- Introduction to Hooks:
  - Hooks are functions that let you "hook into" React state and lifecycle features from function components.
  - Commonly used hooks: useState, useEffect, useCallback, useMemo, useRef.
- useEffect Hook:
  - Combines the lifecycle methods: componentDidMount, componentDidUpdate, and componentWillUnmount.
  - useEffect(callback, dependencies): The callback is run after the render and can optionally clean up.

Examples of Functional Components with Hooks

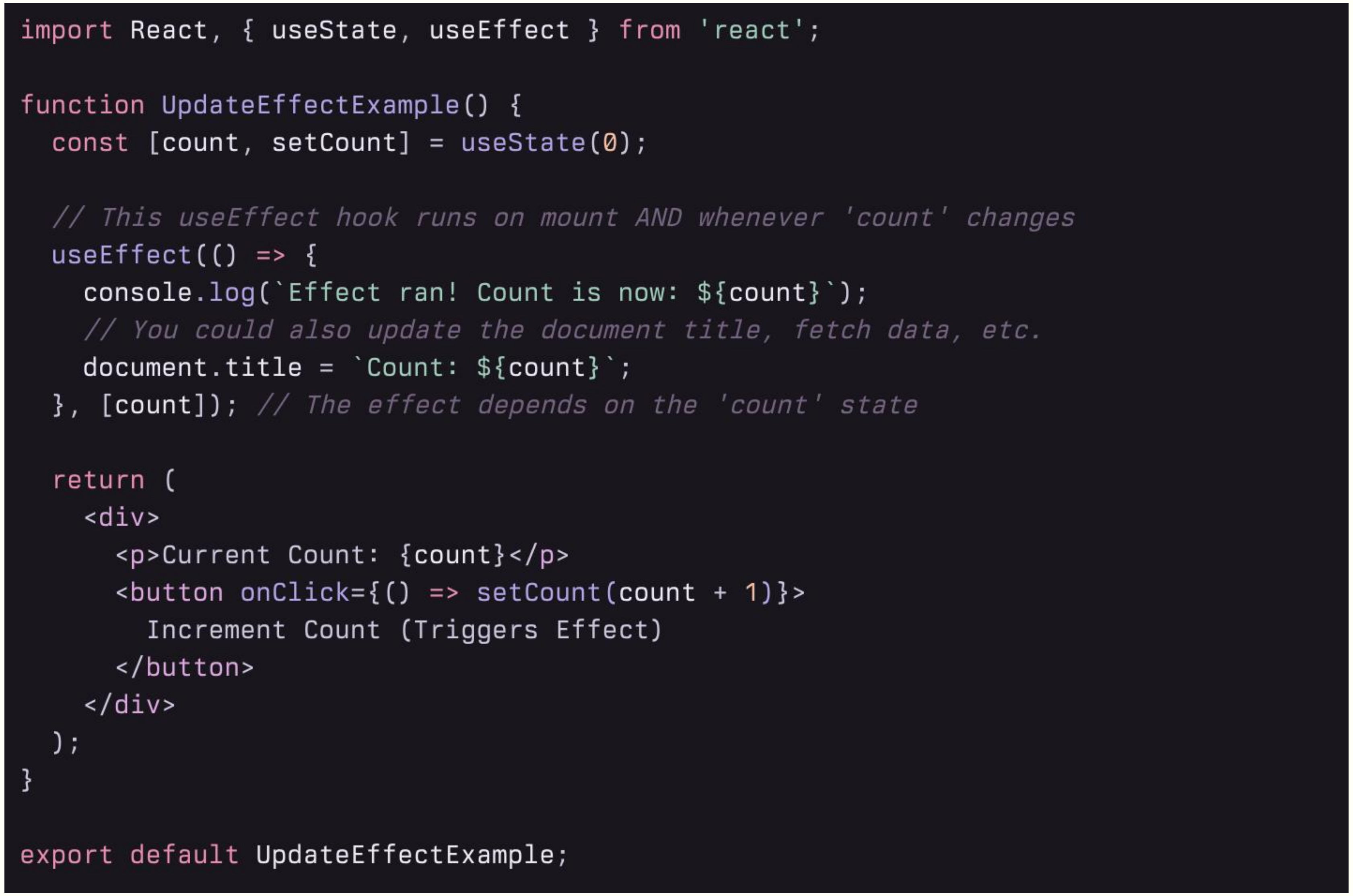
- Mounting with useEffect:



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

export function ExampleComponent() {
  useEffect(() => {
    console.log('Component mounted');
  }, []); // Empty dependency array means this effect runs once

  return <div>Hello, world!</div>;
}

```

Examples of Functional Components with Hooks

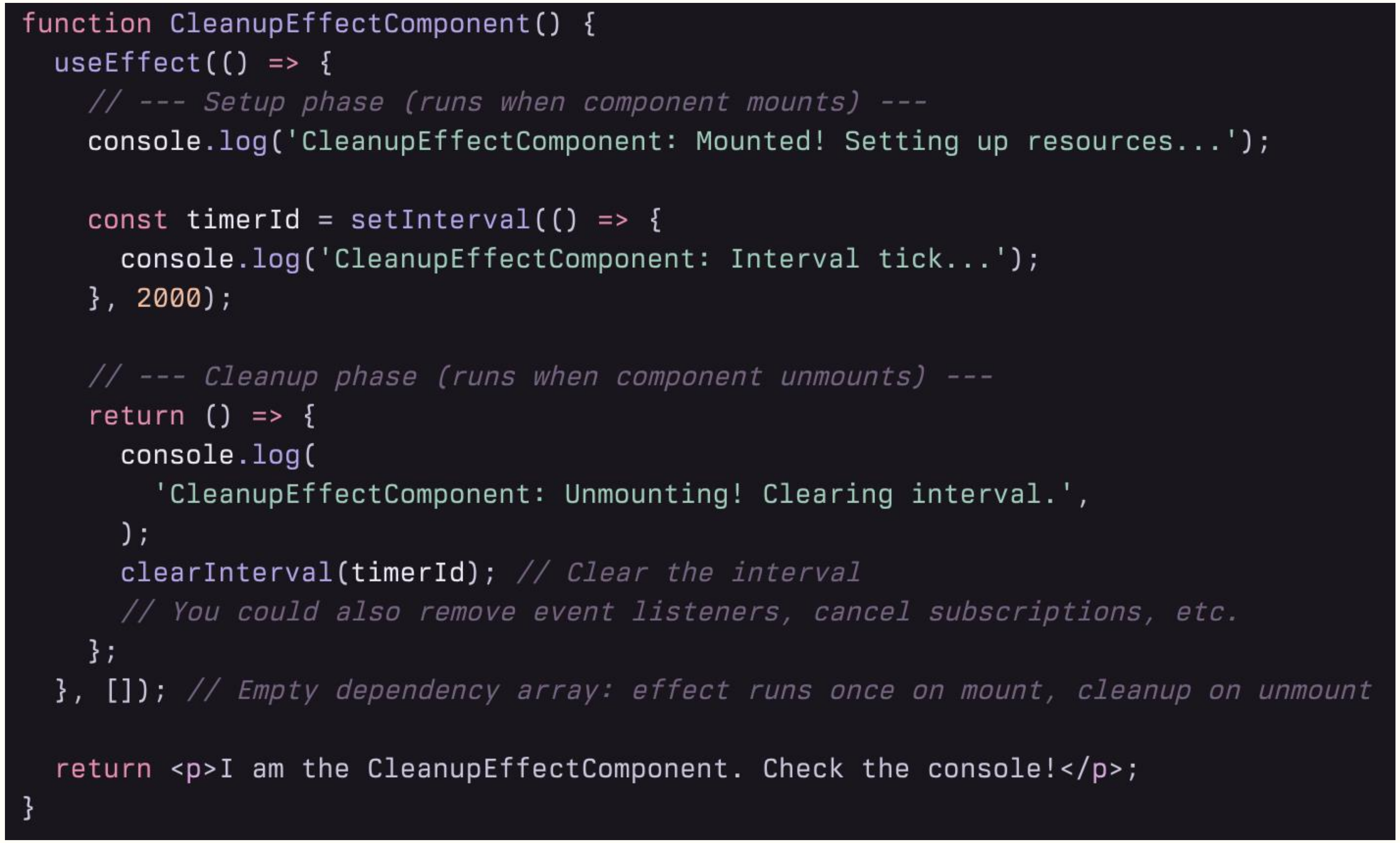
- Updating with useEffect



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

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

  // This useEffect hook runs on mount AND whenever 'count' changes
  useEffect(() => {
    console.log(`Effect ran! Count is now: ${count}`);
    // You could also update the document title, fetch data, etc.
    document.title = `Count: ${count}`;
  }, [count]); // The effect depends on the 'count' state

  return (
    <div>
      <p>Current Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment Count (Triggers Effect)
      </button>
    </div>
  );
}

export default UpdateEffectExample;
```

Examples of Functional Components with Hooks

- Unmounting with useEffect:



```
function CleanupEffectComponent() {
  useEffect(() => {
    // --- Setup phase (runs when component mounts) ---
    console.log('CleanupEffectComponent: Mounted! Setting up resources...');

    const timerId = setInterval(() => {
      console.log('CleanupEffectComponent: Interval tick...');
    }, 2000);

    // --- Cleanup phase (runs when component unmounts) ---
    return () => {
      console.log(
        'CleanupEffectComponent: Unmounting! Clearing interval.',
      );
      clearInterval(timerId); // Clear the interval
      // You could also remove event listeners, cancel subscriptions, etc.
    };
  }, []); // Empty dependency array: effect runs once on mount, cleanup on unmount

  return <p>I am the CleanupEffectComponent. Check the console!</p>;
}
```

Comparing Class and Functional Components

- Class Components:
    - Verbose and often require multiple lifecycle methods.
    - Suitable for scenarios where fine-grained control of lifecycle stages is needed.
- Functional Components:
    - Simpler and more concise.
    - Hooks provide a powerful way to manage state and side effects.
    - Promotes the use of functional programming principles.

Best Practices with Lifecycle Hooks

- Class Components:
  - Minimize stateful logic in constructors.
  - Avoid unnecessary re-renders with shouldComponentUpdate.
  - Clean up resources in componentWillUnmount.
- Functional Components:
  - Use the dependency array in useEffect to control when the effect runs.
  - Clean up subscriptions or timers in the cleanup function of useEffect.
  - Combine multiple useEffect calls for different lifecycle events.

Conclusion

- Lifecycle hooks are essential for managing side effects and state in React.
- Class components offer detailed control with explicit lifecycle methods.
- Functional components with hooks provide a modern, concise way to manage state and effects.

References

- React Documentation: https://reactjs.org/docs/getting-started.html
- React Lifecycle Methods Diagram: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/
- Medium Article on Functional Component Lifecycle: https://medium.com/@addyosmani/a-beginners-guide-to-react-component-lifecycle-5708a33fe45e

The shift begins with you

Trainer name

Trainer

Assistant name

Assistant

trainer@mail.com
assistant@mail.com

Qinshift Academy