In React, `useState` is a hook that allows you to add state to functional components. It provides a way to manage and update values over the lifecycle of a component.

Here's an explanation with examples of how `useState` works in different situations:

1. Basic Counter Example

This example shows how to use `useState` to create a simple counter that increments and decrements when buttons are clicked.

jsx
Copy code
```
import React, { useState } from 'react';

function Counter() {
  // Declare a state variable 'count' with the initial value of 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}

export default Counter;
```

2. Toggle Button Example

Here’s how to use `useState` to toggle between two states (true/false).

jsx
Copy code
```
import React, { useState } from 'react';

function Toggle() {
  // Declare a state variable 'isOn' with the initial value of 'false'
  const [isOn, setIsOn] = useState(false);

  return (
    <div>
      <button onClick={() => setIsOn(!isOn)}>
        {isOn ? 'ON' : 'OFF'}
      </button>
    </div>
  );
}

export default Toggle;
```

3. Managing Multiple State Variables

You can declare multiple `useState` hooks for managing different pieces of state in a component.

jsx
Copy code

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

function UserInfo() {
  // Declare multiple state variables
  const [name, setName] = useState('');
  const [age, setAge] = useState('');

  return (
    <div>
      <input
        type="text"
        placeholder="Enter name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <input
        type="number"
        placeholder="Enter age"
        value={age}
        onChange={(e) => setAge(e.target.value)}
      />
      <p>
        Name: {name}, Age: {age}
      </p>
    </div>
  );
}

export default UserInfo;
```

4. Using Objects in `useState`

Instead of managing multiple state variables, you can also manage the state as an object.

jsx
Copy code
```
import React, { useState } from 'react';

function Profile() {
  const [profile, setProfile] = useState({ name: '', age: '' });

  return (
    <div>
      <input
        type="text"
        placeholder="Enter name"
        value={profile.name}
        onChange={(e) => setProfile({ ...profile, name: e.target.value
})}
      />
      <input
        type="number"
        placeholder="Enter age"
        value={profile.age}
        onChange={(e) => setProfile({ ...profile, age: e.target.value
})}
      />
      <p>
```

```
        Name: {profile.name}, Age: {profile.age}
      </p>
    </div>
  );
}

export default Profile;
```

5. Using Arrays in `useState`

You can also manage arrays in `useState`. Here's an example of how to add new items to an array.

jsx
Copy code
```jsx
import React, { useState } from 'react';

function ItemList() {
  const [items, setItems] = useState([]);

  const addItem = () => {
    setItems([...items, `Item ${items.length + 1}`]);
  };

  return (
    <div>
      <button onClick={addItem}>Add Item</button>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default ItemList;
```

In React, `useEffect` is a hook that allows you to perform side effects in function components. Side effects are actions like data fetching, setting up subscriptions, or manually changing the DOM. The `useEffect` hook is called after the render, and by default, it runs after every render cycle (both the first and subsequent renders). You can also control when it runs by passing dependencies.

Syntax:

```
useEffect(() => {
  // Side effect code here
  return () => {
    // Cleanup code (optional) - runs when component unmounts or
before running the effect again
  };
}, [dependencies]);
```

-

Example 1: Basic usage without dependencies (runs on every render)

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```
export default App;
```

- In this example, every time the component renders (e.g., when you click the button), `useEffect` updates the document's title.

Example 2: Using `useEffect` with dependencies (runs only when `count` changes)

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]); // Only run the effect when `count` changes

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default App;
```

- Here, the effect only runs when the value of `count` changes, improving performance by preventing unnecessary re-renders.

Cleanup with `useEffect`:

If your effect sets up a subscription or timer, you can return a cleanup function that runs when the component is unmounted or the effect is about to run again.

javascript

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Timer running');
  }, 1000);

  // Cleanup function to clear the timer
  return () => clearInterval(timer);
}, []); // Empty array means it only runs once (on mount)
```

4o

In React, `useRef` is a hook that allows you to persist values across renders without causing re-renders. It is primarily used for accessing and manipulating DOM elements directly, but it can also store mutable values similar to instance variables in class components.

Unlike state variables (which trigger re-renders when updated), updates to `useRef` do not cause the component to re-render.

Syntax:

```
const refContainer = useRef(initialValue);
```

- `useRef` returns a **mutable object** with a `current` property, which you can update.
- The initial value is only assigned once, and updates to `current` won't trigger a re-render.

Common use cases:

- Accessing DOM elements (e.g., focusing an input field).
- Storing mutable values that don't cause re-renders (e.g., timers, previous values).

Example 1: Accessing a DOM element

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

```
function App() {
  const inputRef = useRef(null);

  const focusInput = () => {
    // Access the input DOM node and call focus on it
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Click the button
to focus" />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}

export default App;
```

- In this example, `useRef` is used to store a reference to the `input` element. The `focusInput` function can directly call `inputRef.current.focus()` to focus on the input field.

Example 2: Storing a mutable value that persists across renders

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

function App() {
  const [count, setCount] = useState(0);
  const renderCount = useRef(0); // useRef to store render count

  useEffect(() => {
    renderCount.current += 1;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <p>Component rendered {renderCount.current} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}

export default App;
```

- Here, `useRef` is used to store the number of renders of the component (`renderCount`). This value persists across renders, but updating it does not cause the component to re-render.