3. Hooks and Functional Components

Let’s take a simple example of converting a class-based component into a functional component using React hooks. Suppose we have a counter component:

### Class-based Component:

```jsx
// CounterClass.js
Import React, { Component } from ‘react’;

Class CounterClass extends Component {
  Constructor(props) {
    Super(props);
    This.state = {
      Count: 0,
    };
  }

  componentDidMount() {
    document.title = `Count: ${this.state.count}`;
  }

  componentDidUpdate() {
    document.title = `Count: ${this.state.count}`;
  }

  componentWillUnmount() {
    document.title = ‘React App’;
  }
```

```
 incrementCount = () => {
  this.setState((prevState) => ({
   count: prevState.count + 1,
  }));
 };

 Render() {
  Return (
   <div>
    <p>Count: {this.state.count}</p>
    <button onClick={this.incrementCount}>Increment</button>
   </div>
  );
 }
}

Export default CounterClass;
```

### Functional Component with Hooks:

```jsx
// CounterFunctional.js
Import React, { useState, useEffect } from 'react';

Const CounterFunctional = () => {
 Const [count, setCount] = useState(0);
```

```
  useEffect(() => {
    document.title = `Count: ${count}`;

    // Cleanup function
    Return () => {
      Document.title = ‘React App’;
    };
  }, [count]);

  Const incrementCount = () => {
    setCount((prevCount) => prevCount + 1);
  };

  Return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
};

Export default CounterFunctional;
```

### Advantages of Functional Components with Hooks:

1. **Conciseness and Readability:**

Functional components are more concise and readable than class-based components. Hooks like `useState` eliminate the need for constructors, and the component logic is more straightforward.

2. **No ‘this’ Keyword:**

In functional components, you don’t need to use `this`, which can be confusing in JavaScript. This leads to cleaner code and avoids potential pitfalls related to the behavior of `this` in JavaScript.

3. **Easier to Understand Lifecycle:**

Lifecycle methods in class components can be complex and scattered. With hooks like `useEffect`, you can manage lifecycle-related tasks in a more centralized and linear way, improving the understanding of component behavior.

4. **Better Performance:**

Functional components with hooks can offer better performance optimizations. Hooks allow you to optimize re-renders and avoid unnecessary renders by using `React.memo` and `useCallback` effectively.

5. **Encourages Functional Programming:**

Functional components align with the principles of functional programming, making it easier to reason about the state and behavior of your components. This can lead to more maintainable and scalable code.

6. **Hooks are Reusable:**

Custom hooks enable you to reuse stateful logic across components. This promotes code reuse and modularity.

In summary, functional components with hooks provide a more modern and concise way to handle component logic, making code easier to read, write, and maintain. They also address some of the challenges and complexities associated with class-based components, promoting better performance and a more functional programming paradigm.