📌 React Hooks

🚀 1. React Hooks란?

📖 개념

- **Hooks는 React 16.8에서 도입된 기능으로, 함수형 컴포넌트에서도 상태 관리 및 라이프사이클 기능을 사용할 수 있도록 해줍니다.**
- 기존의 **클래스형 컴포넌트에서 사용하던 `this.state`, `this.setState`, `componentDidMount`, `componentDidUpdate` 등의 기능을 함수형 컴포넌트에서 활용 가능**
- **재사용성 증가, 코드 간결화, 성능 최적화 등의 이점을 제공하며, 함수형 컴포넌트의 유일한 상태 관리 및 로직 구현 방식으로 자리 잡고 있음**

✅ React Hooks의 주요 특징

1. **클래스형 컴포넌트 없이도 상태 관리 가능** → `useState`, `useReducer`
2. **컴포넌트 라이프사이클을 함수형 컴포넌트에서 다룰 수 있음** → `useEffect`
3. **참조 및 DOM 요소를 다룰 수 있음** → `useRef`
4. **성능 최적화 가능** → `useMemo`, `useCallback`
5. **전역 상태 관리를 보다 간편하게 구현** → `useContext`
6. **Redux 없이도 상태 공유 가능** → `useReducer`, `useContext` 조합
7. **커스텀 훅(Custom Hooks)으로 로직을 재사용 가능** → 반복되는 로직을 모듈화하여 코드 중복 방지

💡 2. 주요 React Hooks

📌 1) `useState` - 상태 관리

컴포넌트의 상태를 저장하고 변경할 수 있는 훅

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

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>증가</button>
```

```
    </div>
  );
}
```

✅ `useState`는 배열 비구조화 할당을 사용하며, 상태 변경 시 렌더링이 자동으로 수행됨 ✅ 여러 개의 상태를 각각 관리할 수 있으며, 상태가 객체일 경우 `spread` 연산자를 활용하여 변경해야 함

```
const [user, setUser] = useState({ name: "Alice", age: 25 });

setUser({ ...user, age: 30 }); // 기존 값 유지하며 age 변경
```

📌 2) `useEffect` - 사이드 이펙트 관리

컴포넌트가 렌더링될 때 실행되는 함수 (API 요청, 이벤트 리스너 등록 등)

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

function Timer() {
  const [time, setTime] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => setTime(prev => prev + 1), 1000);
    return () => clearInterval(interval);
  }, []); // 빈 배열 = 컴포넌트가 처음 마운트될 때만 실행

  return <p>경과 시간: {time}초</p>;
}
```

✅ `useEffect`는 마운트, 업데이트, 언마운트 시점을 제어 가능하며, 의존성 배열을 활용해 특정 조건에서만 실행 가능

✅ **클린업 함수**를 반환하여 메모리 누수를 방지할 수 있음

```
useEffect(() => {
  const handleResize = () => {
    console.log(window.innerWidth);
  };
  window.addEventListener("resize", handleResize);
  return () => {
    window.removeEventListener("resize", handleResize);
  };
}, []);
```

📌 3) `useRef` - DOM 요소 접근 및 값 유지

렌더링과 무관한 값을 저장하거나 DOM 요소에 접근할 때 사용

```
import { useRef, useEffect } from "react";

function InputFocus() {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} placeholder="자동 포커스" />;
}
```

✅ `useRef`는 **리렌더링이 발생하지 않는 값**을 저장하는 데 유용하며, 변수값을 유지할 수 있음

📌 4) `useReducer` - 복잡한 상태 관리

Redux와 유사한 방식으로 상태를 업데이트하는 훅

```
import { useReducer } from "react";

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
      <button onClick={() => dispatch({ type: "decrement" })}>-</button>
    </div>
  );
}
```

✅ `useReducer`는 `useState`보다 복잡한 상태 로직을 다룰 때 유용하며, `dispatch`를 사용해 명확한 액션을 전달 가능

📌 5) `useContext` - 전역 상태 관리

컴포넌트 간 전역 데이터를 공유할 때 사용 (Redux 대체 가능)

```
import { createContext, useContext } from "react";

const ThemeContext = createContext("light");

function ThemeDisplay() {
  const theme = useContext(ThemeContext);
  return <p>현재 테마: {theme}</p>;
}
```

✅ `useContext`를 사용하면 **props drilling 없이 전역 상태를 관리할 수 있음**

📌 6) `useCallback` - 함수 메모이제이션

함수가 불필요하게 생성되지 않도록 메모이제이션

```
import { useCallback } from "react";

const handleClick = useCallback(() => {
  console.log("버튼 클릭");
}, []);
```

✅ `useCallback`을 사용하면 **함수 재생성을 방지하여 성능 최적화 가능**

📌 7) `useMemo` - 연산 최적화

비싼 계산 비용이 드는 연산을 캐싱하여 성능 최적화

```
import { useMemo } from "react";

const squaredValue = useMemo(() => expensiveCalculation(num), [num]);
```

✅ `useMemo`를 사용하면 **불필요한 연산을 방지하여 성능 최적화 가능**
