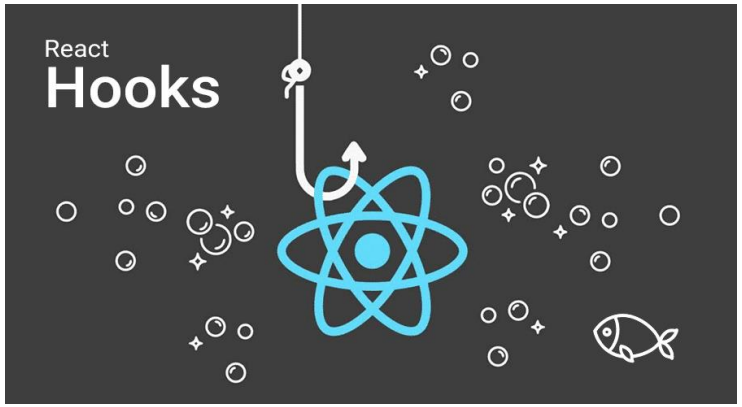
React

Hooks

Hooks

복습) Hooks 란?



- React 의 새로운 기능
- 클래스 컴포넌트에서만 가능했던

state(상태관리) 와 lifecycle(라이프사이클) 이 가능하도록 돕는 기능

클래스형 컴포넌트에서 사용할 수 있는 기능을

함수형 컴포넌트에서도 사용할 수 있도록 **"낚아채는 기능"**

복습) Hook 사용 규칙

codingOn

- 최상위 단계에서만 호출 가능
  - 최상위 컴포넌트 X
  - 반복문, 조건문, 중첩된 함수 내부에서 호출하면 안 되는 것!

- Hook 은 오로지 **React 함수형 컴포넌트 안에서만 호출** 가능하다.
- 커스텀 훅 이름은 "use"로 시작(권장사항)

Hooks 종류

- useState(): 상태 관리를 위한 가장 기본적인 훅
- useRef(): 참조(reference)를 생성하고 관리할 수 있는 훅 (DOM 접근, 변수 보존 등)
- useEffect(): 컴포넌트가 렌더링 될 때마다 특정 작업을 수행하도록 설정할 수 있는 훅
- useMemo(): 메모이제이션을 통해 함수의 리턴 값을 재사용할 수 있게 해주는 훅
- useCallback(): 함수를 메모이제이션하여 불필요한 렌더링을 줄이게 해주는 훅
- useReducer(): 복잡한 컴포넌트 상태 로직을 리듀서 함수를 통해 관리할 수 있는 훅
- useContext(): 리액트에서 전역적으로 접근 가능한 데이터나 함수를 관리하고,
  필요한 컴포넌트에서 그 값을 바로 가져와 사용할 수 있게 도와주는 훅

더 많은 Hooks 살펴보기 https://react.dev/reference/react

Hook – useMemo()

- 함수형 컴포넌트 내부에서 발생하는 **연산을 최적화**시켜주는 Hook
- Rendering 과정에서 **특정 값이 바뀌었을 때만 연산을 실행**한다.

```
const memoizedValue = useMemo(callback, dependency);
```

- 렌더링 과정에서 두 번째 인자로 받은 의존 배열(dependency) 내 값이 바뀌는 경우에만 첫번째 인자로 받은 콜백함수를 실행해 구한 값을 반환한다.

Hook – useMemo()

- useMemo() 사용 이유

```
function calc(a, b) {
  return a + b
}

// 함수형 컴포넌트
const MyComponent() {
  const result = calc(3,5)

  return <p>{result}</p>
}
```

컴포넌트가 랜더링 됨 = 함수를 호출 → 함수 내부의 모든 변수 초기화

리렌더링 될 때마다 MyComponent 호출,

변수 result가 초기화 되므로 매번 calc 함수 실행!

=> useMemo를 사용하여 부하가 걸리는 함수의 결괏값을 메모리에 저장한 뒤,

리렌더링이 될 때 그 결괏값만 가져와서 재사용해 줌으로써 성능을 최적화!

Hook – useCallback()

codingOn

- **Rendering 최적화**에 사용되는 Hook API
- useMemo와 유사함. useMemo에서는 값을 최적화 시켰지만, **다시 rendering 될 때 함수를 다시 불러오는 것을** 막는다.

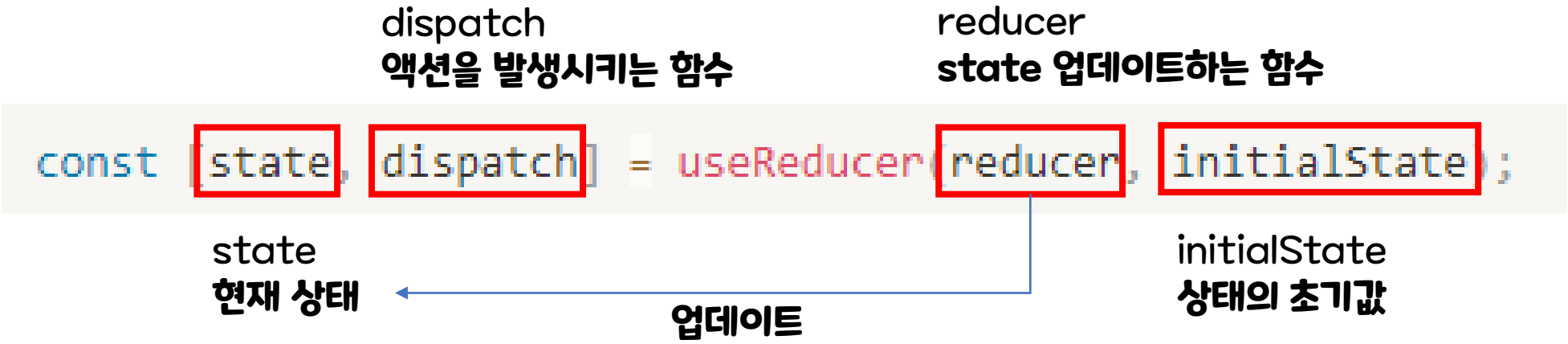
```
const memoizedCallback = useCallback(callback, dependency);

const onClick = useCallback(e => {
    e.preventDefault();
    setNumber(number + 1);
), [number]};
```

Hook – useReducer()

- Reducer 란?

: 현재 상태와 업데이트를 위해 필요한 정보를 담은 액션 값을 전달받아 새로운 상태를 반환하는 함수



Hook – useReducer()

- 장점: 컴포넌트 업데이트 로직을 컴포넌트 외부로 뺄 수 있음
- useReducer은 useState의 대체 함수로 다양한 컴포넌트 상황에 따라 상태값을 설정할 수 있다.

1 Reducer 정의

```
const [number, dispatch] = useReducer(reducer, 0);
```

2 dispatch 함수로 action 값 전달
: dispatch는 action 값을 받아 state와 함께 3 으로 전달

```
return (
  <div>
    <h1>useReducer hooks test</h1>
    <h2>{number}</h2>
    <button onClick={() => dispatch('INCREMENT')}>Plus</button>
    <button onClick={() => dispatch('DECREMENT')}>Minus</button>
    <button onClick={() => dispatch('RESET')}>Reset</button>
  </div>
);
```

3 Reducer
: 현재 state와 action 값을 전달 받아 새로운 state 반환

```
const reducer = (prevNumber, action) => {
  switch (action) {
    case 'INCREMENT':
      return prevNumber + 1;
    case 'DECREMENT':
      return prevNumber - 1;
    case 'RESET':
      return 0;
    default:
      return prevNumber;
  }
};
```

참고) useState() vs. useReducer()

- useReducer()가 useState() 기반이라고 해서 더 좋은 것은 아니다!
- 상황에 따라 적절한 hook 을 선택하자

- state가 **단순**하다면 **useState()**를 사용
- state가 **복잡**하다면 **useReducer()**를 사용
  (객체, 배열 같이 하위 요소가 많은 경우)

Custom Hooks

- 컴포넌트에서 반복되는 로직이 많을 때 custom hooks을 이용하면 편리
- 즉, 컴포넌트 분할과 달리 컴포넌트 로직 자체를 분할하거나 재사용이 가능

- 보통은 hooks/ 디렉토리 안에 커스텀 훅을 정의하여 사용
  - use 로 시작하는 파일을 만드는 것이 관례
  - ex. useScroll.js, useToggle.js 등