1. useEffect를 사용하여 마운트/언마운트/업데이트시 할 작업 설정하기

* useEffect Hook을 사용하여 컴포넌트가 마운트 됐을 떄(처음 나타났을 때), 언마운트 됐을 때(사라질 때), 그리고 업데이트 될 때(특정 props가 바뀔 때) 특정 작업을 처리하는 방법
* Ex) useEffect(() => {
  + Console.log(‘컴포넌트가 화면에 나타남’);
  + Return () => {
    - Condole.log(‘컴포넌트가 화면에서 사라짐’);
  + };
* },[]);
* // 첫번째 파라미터에는 함수, 두번째 파라미터에는 의존값이 들어있는 배열(deps)을 넣습니다. 만약에
* deps 배열을 비우게 된다면, 컴포넌트가 처음 나타날때에만 useEffect에 등록한 함수가 호출됩니다.
* useEffect에서는 함수를 반환 할 수있는데 이를 cleanUp 함수라고 부릅니다.
* cleanUp함수는 useEffect에 대한 뒷정리를 해준다고 이해하면 된다. deps가 비어있는 경우 컴포넌트가 사라질 떄 cleanup 함수가 호출됩니다.
* 주로, 마운트 시에 하는 작업들은 다음과 같은 사항들이 있습니다.
  + 1. props로 받은 값을 컴포넌트의 로컬 상태로 설정
  + 2. 외부 API 요청(Rest API등)
  + 3. 라이브러리 사용(D3, Video.js 등)
  + 4. setInterval 을 통한 반복작업 혹은 setTimeout 을 통한 작업예약
* 언마운트 시에 하는 작업
  + SetInterval, setTimeout 을 사용하여 등록한 작업들 clear하기
  + 라이브러리 인스턴스 제거
* Deps에 특정 값 넣기
  + Deps에 특정 값을 넣게 된다면, 컴포넌트가 처음 마운트 될 때에도 호출이 되고, 지정한 값이 바뀔 때에도 호출이 됩니다.
  + 그리고, deps안에 특정 값이 있다면 언마운트 시에도 호출이 되고, 값이 바뀌지 직전에도 호출이 됩니다.
  + useEffect 안에서 사용하는 상태나, props가 있다면, useEffect의 deps에 넣어주어야 합니다.
  + 만약 useEffect 안에서 사용하는 상태나 props를 deps에 넣지 않게 된다면 useEffect에 등록한 함수가 실행 될 때 최신 props/상태를 가르키지 않게 됩니다.(중요)
* Deps 파라미터를 생략하기
  + Deps 파라미터를 생략한다면, 컴포넌트가 리렌더링 될 때마다 호출이 됩니다.!!
* 리액트 컴포넌트는 기본적으로 부모컴포넌트가 리렌더링 되면 자식 컴포넌트 또한 리렌더링 됩니다. (바뀐 내용이 없어도)
* 물론, 실제 DOM 에 변화가 반영되는 것은 바뀐 내용이 있는 컴포넌트에만 해당합니다. 하지만, virtual DOM 에는 모든걸 다 렌더링 하고 있습니다.
* 나중에는, 컴포넌트를 최적화 하는 과정에서 기존의 내용을 그대로 사용하면서 VirtualDOM에 렌더링하는 리소를 아낄 수도있습니다.

1. useMemo를 사용하여 연산한 값 재사용하기

* “성능최적화”를 위하여 연산된 값을 useMemo라는 Hook을 사용하여 재사용하는 방법을 알아보도록 하겠습니다.
* 활성 사용자 수를 세는건, users 에 변화가 있을때만 세야되는건데, input값이 바뀔 때에도 컴포넌트가 리렌더링 되므로, 불필요할때에도 호출하여서 자원이 낭비되고 있습니다.
* 이러한 상황에는 useMemo라는 Hook함수를 사용하면 성능을 최적화 할 수있습니다.
* useMemo의 첫번째 파라미터에는 어떻게 연산 할지 정의하는 함수를 넣어주면 되고, 두번째 파라미터에는 deps배열을 넣어주면 되는데, 이 배열 안에 넣은 내용이 바뀌면, 우리가 등록한 함수를 호출해서 값을 연산해주고, 만약에 내용이 바뀌지 않았다면, 이전에 연산한 값을 재사용하게 됩니다.
* Ex) function countActiveUsers(users){
  + Console.log(‘활성 사용자 수를 세는중...’);
  + Return users.filter(user => user.active).length;
* }
* Const count = useMemo(
  + ()=>countActiveUsers(users),[users]);
  + <h1>활성화 갯수 : {count}</h1>
  + 원래는 inputs값이 변경될 때마다 리렌더링이 이루어졌었는데, useMemo를 써줌으로써, users의 내용이 변경될때만 우리가 등록한 ㅅ함수를 호출해서 값을 연산해주고, 만약에 내용이 바뀌지 않았다면 이전에 연산한 값을 재사용하게 됩니다.

1. useCallback 을 사용하여 함수 재사용하기

* useMemo는 특정 결과값을 재사용 할 때 사용하는 반면, useCallback은 특정 함수를 새로 만들지 않고 재사용하고 싶을 때 사용합니다.
* onCreate, onRemove, onToggle 함수들은 컴포넌트가 리렌더링 될 때마다 새로 만들어집니다.
* 컴포넌트에서 props가 바뀌지 않았으면 virtualdom에 새로 렌더링 하는 것 조차 하지 않고 컴포넌트의 결과물을 재사용하는 최적화 작업을 하게 되는데, 이 작업을 하려면, 함수를 재사용하는 것이 필수입니다.
* Ex) const onChange = useCallback( e => {
  + Const { name, value } = e.target;
  + setInput({
    - ..inputs,
    - [name]:value
  + });
  + }, [inputs]);
  + 함수 안에서 사용하는 상태 혹은 props가 있다면 꼭, deps 배열안에 포함시켜야 된다는 것입니다. 만약에 deps 배열 안에 함수에서 사용하는값을 넣지 않게 된다면, 함수 내에서 해당 값들을 참조할 떄 가장 최신 값을 참조 할 것이라고 보장 할 수 없습니다.props로 받아온 함수가 있다면, 이 또한 deps에 넣어주어야합니다.
  + useCallback 은 useMemo를 기반으로 만들어졌습니다. 다만, 함수를 위해서 사용할 때 더욱 편하게 해준 것 뿐입니다.
  + 다만, 함수를 위해서 사용할 때 더욱 편하게 해준 것,
  + Const onToggle = useMemo(
    - () => () => {
      * /\* ...\*/
    - }, [users]);

1. React.memo를 사용한 컴포넌트 리렌더링 방지

* 컴포넌트의 props가 바뀌지 않았다면, 리렌더링을 방지하여 컴포넌트의 리렌더링 성능 최적화를 해줄 수있는 React.memo 함수
* 이 함수를 사용하면, 컴포넌트에서 리렌더링이 필요한 상황에서만 리렌더링을 하도록 설정해 줄 수있다.
* Export default React.memo(CreateUser);
* Ex) const User = React.memo(function User({user, onRemove, onToggle})
* useCallback deps에 users가 들어있기 때문에 배열이 바뀔때마다 함수가 새로 만들어지는 건 당연, 이걸 최적화하려면 바로 deps에서 users를 지우고, 함수들에서 현재 useState로 관리하는 users를 참조하지 않게 하는 것입니다.
* Ex) const onRemove = useCallback(id => {
  + setUsers(users => users.filter(user => ( users.id !== id));
  + },[]);
  + 함수형 업데이트를 하게 되면, setUsers 에 등록하는 콜백함수의 파라미터에서 최신 users를 참조 할 수 있기 떄문에 deps에 users를 넣지 않아도 됩니다.
* 추가적으로 React.memo에서 두번째 파라미터에 propsAreEqual이라는 함수를 사용하여 특정 값들만 비교하는 것도 가능합니다.
* Export default React.memo(
  + UserList,
  + (prevProps, nextProps) => prevProps.users === nextProps.users
* );
* => 이걸 잘못사용한다면 오히려 의도치 않은 버그들이 발생하기 쉽다.
* 예를들어, 함수형 업데이트로 전환을 안했는데 이렇게 users만 비교를 하게 된다면, onToggle과 onRemove에서 최신 users 배열을 참조하지 않으므로 심각한 오류가 발생 할 수있습니다,

1. useReducer를 사용하여 상태 업데이트 로직 분리하기

* 상태를 관리하게 될 때 useState를 사용하는 것 말고도 다른 방법이 있다. 바로 useReducer를 사용하는 것인데, 이 hook함수를 사용하면, 컴포넌트의 상태 업데이트 로직을 컴포넌트에서 분리시킬 수있습니다.
* useReducer hook 함수
* Reducer는 현재 상태와 액션 객체를 파라미터로 받아와서 새로운 상태를 반환해주는 함수입니다.
* Function reducer(state, action){
  + //새로운 상태를 만드는 로직
  + //const nextState = ...
  + Return nextState’
* }
* - reducer에서 반환하는 상태는 곧 컴포넌트가 지닐 새로운 상태가 됩니다.
* useReducer의 사용법
* Ex) const [state, dispatch] = useReducer(reducer, initialState);
* state : 우리가 앞으로 컴포넌트에서 사용할 수 있는 상태
* Dispatch : 액션을 발생시키는 함수
* -> 이 함수는 다음과 같이 사용합니다 ex) dispatch({type: ‘INCREMENT’)});