7. 훅스로 컴포넌트 개선하기

Prof. **Seunghyun Park** (sp@hansung.ac.kr)

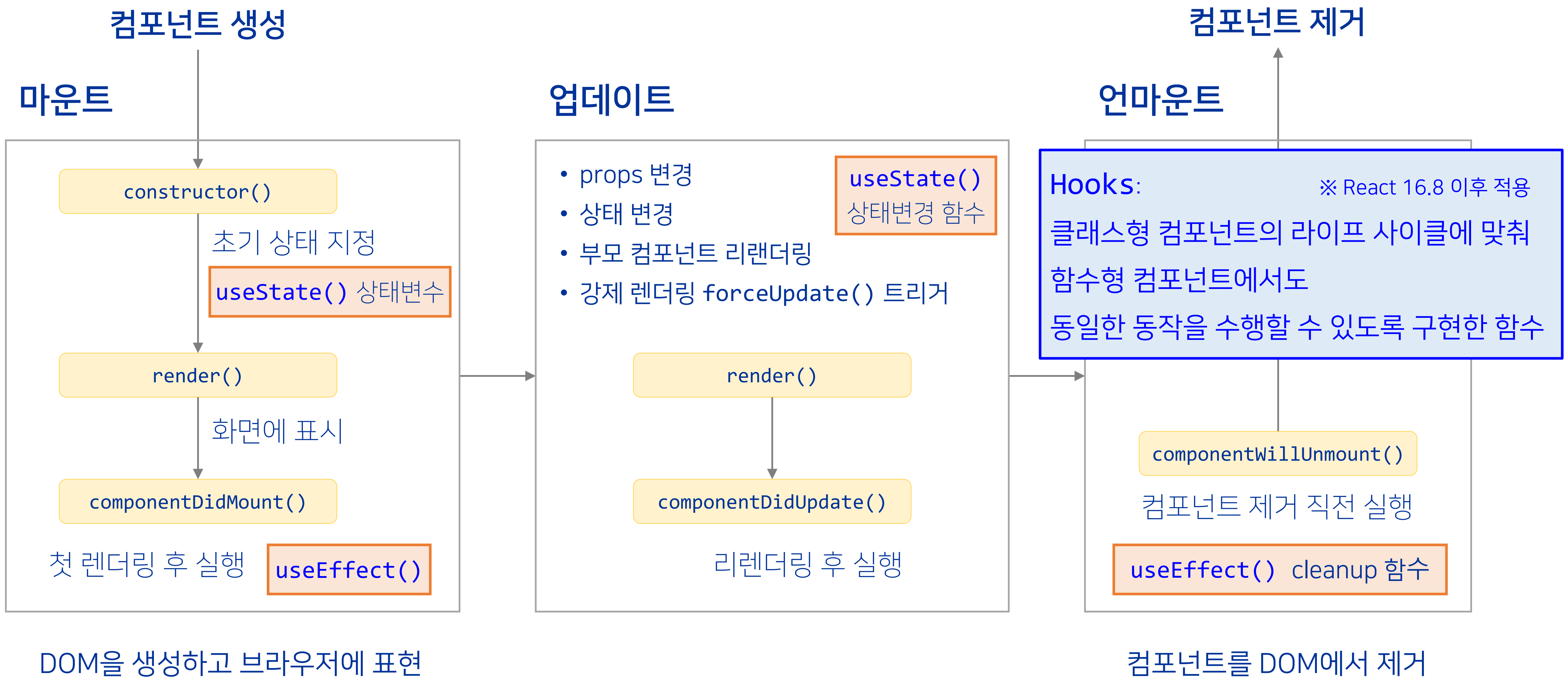
Division of Computer Engineering

한성대학교 HANSUNG UNIVERSITY

학습 목표: 7장. 훅스로 컴포넌트 개선하기

- 리액트 컴포넌트 라이프 사이클
- useState()
- useEffect()
- useMemo()
- useReducer()
- useCallback(), useContext(), and custom Hooks

리액트 컴포넌트 라이프사이클 (클래스 컴포넌트와 함수 컴포넌트의 Hooks)



Hooks: ※ React 16.8 이후 적용

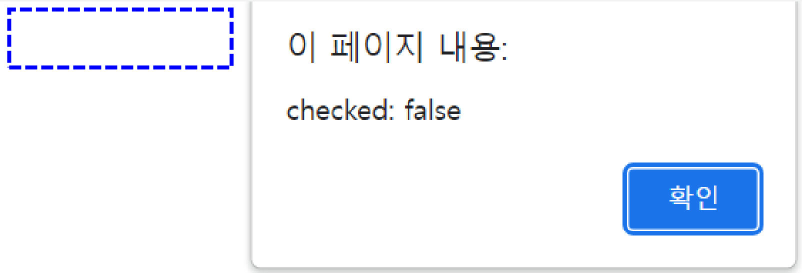
클래스형 컴포넌트의 라이프 사이클에 맞춰
함수형 컴포넌트에서도
동일한 동작을 수행할 수 있도록 구현한 함수

useEffect()

https://ko.reactjs.org/docs/hooks-effect.html

```
<div id="root"></div>

/* ch07-01-1.html */
const Checkbox = () => {
  const [checked, setCheck] = React.useState(false);

  alert(`checked: ${checked.toString()}`);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setCheck( checked => !checked )}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
};

const root =
  ReactDOM.creat
root.render(<Che
```

Checkbox() 컴포넌트가
<>...</> 엘리먼트를 반환하기 전 alert() 수행


이 페이지 내용:

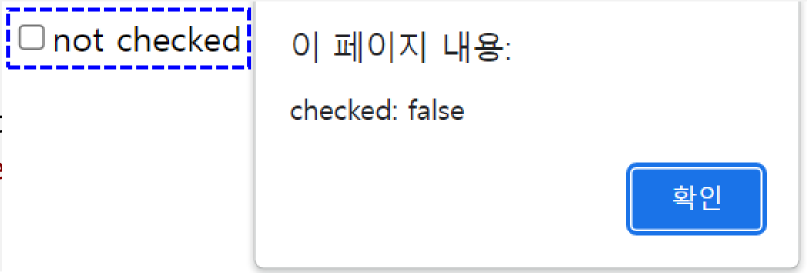
checked: false

- useEffect(): 리액트 컴포넌트가 렌더링된 후에 수행할 동작을 정의하는 Hooks

```
React.useEffect( callback );
```

```
/* ch07-01-2.html */
const Checkbox = () => {
  const [checked, setCheck] = React.useState(false);

  React.useEffect( () => {
    alert(`checked: ${checked.toString()}`);
  });

  return ( ... );
};

const root =
  ReactDOM.creat
root.render(<Che
```

Checkbox() 컴포넌트가
렌더링된 이후 alert() 수행


☐ not checked

이 페이지 내용:

checked: false

useEffect() – 조건부 effect 발생: 의존관계 배열 활용 (1)

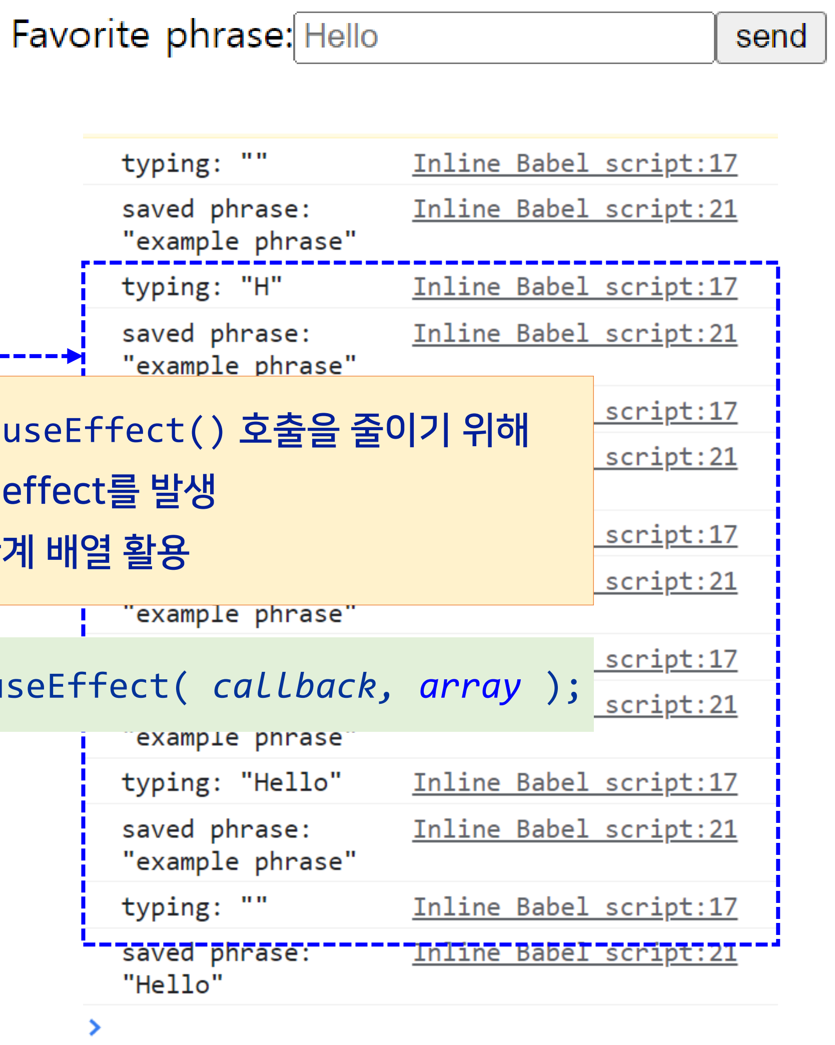
https://ko.reactjs.org/docs/hooks-reference.html#conditionally-firing-an-effect

```
/* ch07-02-1.html */
const Favorite = () => {
  const [typed, setTyped] = React.useState("");
  const [phrase, setPhrase] = React.useState("ex-phrase");

  const createPhrase = () => {
    setPhrase(typed);
    setTyped("");
  };

  React.useEffect(() => console.log(`typing: "${typed}"`));
  React.useEffect(() => console.log(`saved: "${phrase}"`));

  return (
    <>
      <label>Favorite phrase:</label>
      <input value={typed} placeholder={phrase}
        onChange={e => setTyped(e.target.value)} />
      <button onClick={createPhrase}>send</button>
    </> );
};

const root =
  ReactDOM.createRoot(document.getElementById('root'));
root.render(<Favorite />);
```

2) 상태가 변경될 때마다 렌더링 발생 → useEffect() Hooks 동작

1) 키 입력이 발생할 때마다 setTyped()로 typed 상태를 변경

3) 버튼을 클릭하면 createPhrase()가 typed와 phrase 상태를 갱신



```
typing: ""                      Inline Babel script:17
saved phrase:                   Inline Babel script:21
"example phrase"
typing: "H"                     Inline Babel script:17
saved phrase:                   Inline Babel script:21
"example phrase"
                                             script:17
                                             script:21
                                             script:17
                                             script:21
"example phrase"
                                             script:17
                                             script:21
"example phrase"
typing: "Hello"                 Inline Babel script:17
saved phrase:                   Inline Babel script:21
"example phrase"
typing: ""                      Inline Babel script:17
saved phrase:                   Inline Babel script:21
"Hello"
>
```

불필요한 useEffect() 호출을 줄이기 위해 조건부로 effect를 발생
→ 의존관계 배열 활용

`React.useEffect( callback, array );`

useEffect() – 조건부 effect 발생: 의존관계 배열 활용 (2)

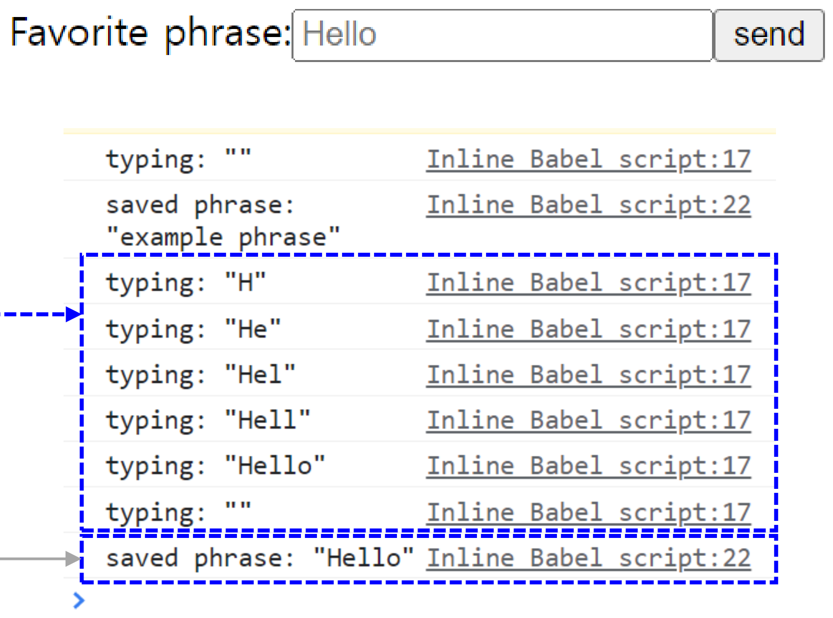
https://ko.reactjs.org/docs/hooks-reference.html#conditionally-firing-an-effect

```
/* ch07-02-2.html */
const Favorite = () => {
  ...
  const createPhrase = () => {
    setPhrase(typed);
    setTyped("");
  };

  React.useEffect(() => console.log(`typing: "${typed}"`), [typed]);
  React.useEffect(() => console.log(`saved phrase: "${phrase}"`), [phrase]);

  return (
    <>
      ...
      <input value={typed} placeholder={phrase}
        onChange={e => setTyped(e.target.value)} />
      <button onClick={createPhrase}>send</button>
    </>
  );
};
```

렌더링 이후 수행할 useEffect(*callback*)에 대한 조건을 배열로 명시

typed가 변경된 렌더링에만 실행

phrase가 변경된 렌더링에만 실행



```
typing: ""                          Inline Babel script:17
saved phrase:                       Inline Babel script:22
"example phrase"
typing: "H"                         Inline Babel script:17
typing: "He"                        Inline Babel script:17
typing: "Hel"                       Inline Babel script:17
typing: "Hell"                      Inline Babel script:17
typing: "Hello"                     Inline Babel script:17
typing: ""                          Inline Babel script:17
saved phrase: "Hello"               Inline Babel script:22
>
```

```
React.useEffect( callback, array );
```

배열의 값에 따라 callback 실행 조건이 달라짐

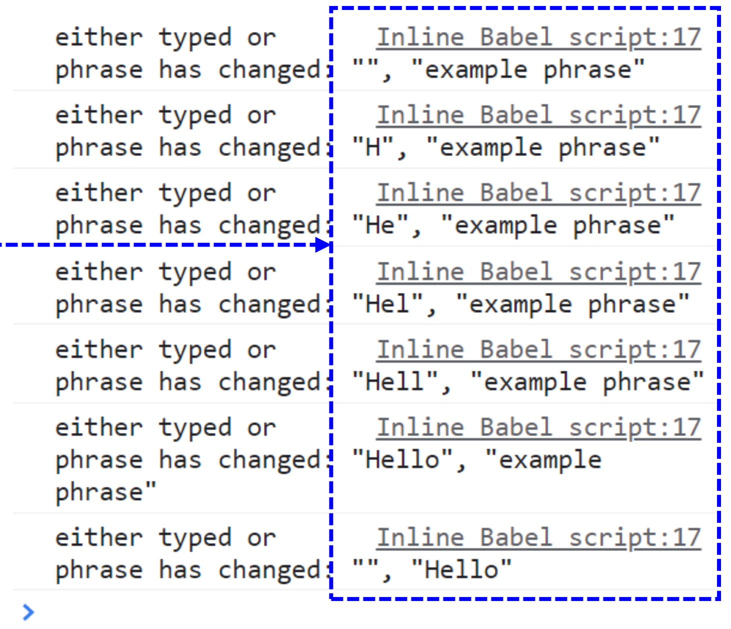
useEffect() – 조건부 effect 발생: 의존관계 배열 활용 (3)

https://ko.reactjs.org/docs/hooks-reference.html#conditionally-firing-an-effect

```
React.useEffect( callback, array );
```

```
/* ch07-02-3.html */
  ...
  React.useEffect(
    () => console.log(`either typed or phrase changed: "${typed}", "${phrase}"`),
    [typed, phrase]
  );

  return (...);
...
```

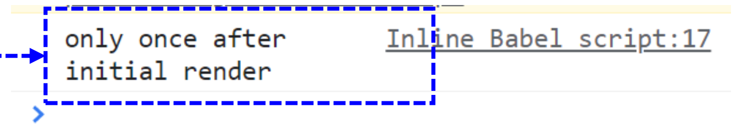
여러 요소를 갖는 배열:
typed 또는 phrase가 변경된 렌더링에만 실행



```
either typed or phrase has changed: "", "example phrase"        Inline Babel script:17
either typed or phrase has changed: "H", "example phrase"       Inline Babel script:17
either typed or phrase has changed: "He", "example phrase"      Inline Babel script:17
either typed or phrase has changed: "Hel", "example phrase"     Inline Babel script:17
either typed or phrase has changed: "Hell", "example phrase"    Inline Babel script:17
either typed or phrase has changed: "Hello", "example phrase"   Inline Babel script:17
either typed or phrase has changed: "", "Hello"                 Inline Babel script:17
>
```

```
/* ch07-02-4.html */
  ...
  React.useEffect(
    () => console.log(`only once after initial render`),
    []
  );

  return (...);
...
```

원소가 없는 배열:
최초의 렌더링에만 1번 실행



```
only once after initial render        Inline Babel script:17
>
```

useEffect() – 조건부 effect 발생: 함수의 반환

```
/* ch07-03-1.html */
const Info = () => {
  const [name, setName] = React.useState("");
  const [nickname, setNickname] = React.useState("");

  React.useEffect(
    () => {
      console.log('useEffect(): 화면에 나타남');
      console.log(`name: ${name}`);
      return () => {
        console.log('useEffect(), cleanup: 화면에서 사라짐');
        console.log(`name: ${name}`);
      }
    },
    []
  );

  const onChangeName = e => setName(e.target.value);
  const onChangeNickname = e => setNickname(e.target.value);

  return ( <>
      <div>
        <input value={name} onChange={onChangeName} />
        <input value={nickname} onChange={onChangeNickname} />
      </div>
      <div><b>Name: </b>{name}</div>
      <div><b>Nickname: </b>{nickname}</div>
    </> )
};
```

cleanup 함수:
컴포넌트가 사라지기 전 실행

[]: 최초의 렌더링에만 1번 실행

```
const App = () => {
  const [visible, setVisible] = React.useState(false);
  return (
    <>
      <button onClick={() => setVisible(!visible)}>
        {visible ? "숨기기" : "보이기"}
      </button>
      {visible && <Info />}
    </>
  );
};

const root =
  ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

```
React.useEffect(
  () => {
    ... ;
    return () => {
      ...
    }
  }
  ,
  array
);
```

callback

cleanup func.

*callback*이
함수를 반환하는 경우,
cleanup 함수는
컴포넌트가 사라지기 전 실행

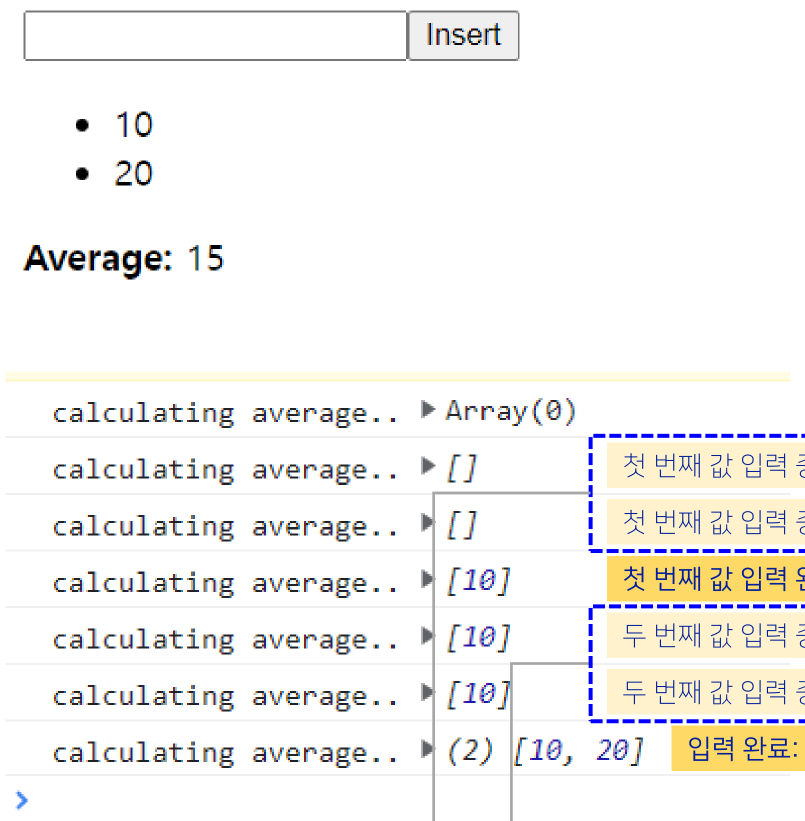
useMemo()

```
/* ch07-04-1.html */
const Average = () => {
  const [list, setList] = React.useState([]);
  const [number, setNumber] = React.useState('');

  const onChange = e => setNumber(e.target.value);
  const onInsert = e => {
    const nextList = list.concat(parseInt(number));
    setList(nextList);
    setNumber('');
  };
  const getAverage = numbers =>{
    console.log("calculating average..", list);
    if (numbers.length === 0) return 0;
    const sum = numbers.reduce((a, b) => a + b);
    return sum/numbers.length;
  };

  return (
    <div>
      <input value={number} onChange={onChange} />
      <button onClick={onInsert}>Insert</button>
      <ul>{list.map((value, i) => <li key={i}>{value}</li>)}</ul>
      <div><b>Average: </b>{getAverage(list)}</div>
    </div>
  )
};
```

버튼 클릭 → input의 값 number 추가, list 갱신 → 상태변경 → 재렌더링

렌더링될 때마다 getAverage() 실행 → 재계산

input 값 입력 → number 상태 변경 → 재렌더링



Insert

- 10
- 20

Average: 15

```
calculating average.. ▶ Array(0)
calculating average.. ▶ []
calculating average.. ▶ []
calculating average.. ▶ [10]
calculating average.. ▶ [10]
calculating average.. ▶ [10]
calculating average.. ▶ (2) [10, 20]
>
```

첫 번째 값 입력 중: 1

첫 번째 값 입력 중: 10

첫 번째 값 입력 완료: 10

두 번째 값 입력 중: 2

두 번째 값 입력 중: 20

입력 완료: 10, 20

실제 계산에 필요한 입력 값이 전달되기 전 (값을 입력하는 중)에는 재계산과 재렌더링이 필요 없음

useMemo()

https://ko.reactjs.org/docs/hooks-reference.html#usememo

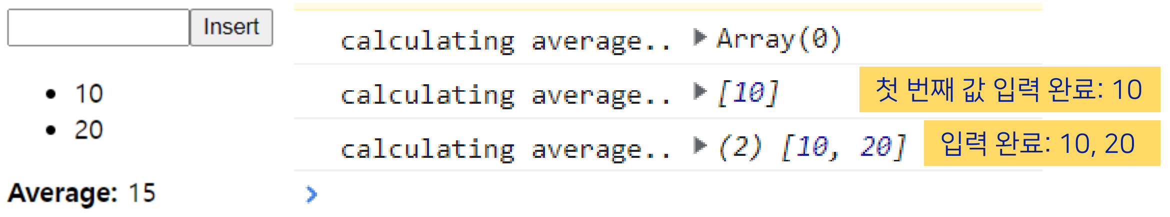
```
/* ch07-04-2.html */
const Average = () => {
  const [list, setList] = React.useState([]);
  const [number, setNumber] = React.useState('');

  const onChange = ...;  onChange()는 list를 변경하지 않음
  const onInsert = e => {
    const nextList = list.concat(parseInt(number));
    setList(nextList);  onInsert()는 list를 변경
    setNumber('');
  };
  const getAverage = ...;

  const avg = React.useMemo(
    () => getAverage(list),
    [list]
  );                     변경된 값 전달

  return (
    <div>
      <input value={number} onChange={onChange} />
      <button onClick={onInsert}>Insert</button>
      <ul>{list.map((value, i) => <li key={i}>{value}</li>)}</ul>
      <div><b>Average: </b>{avg}</div>
    </div>
  )
};
```

- useMemo(): 의존성 array가 변경되었을 때에만 다시 계산하는 Hooks, 결과 값을 반환

```
memoizedValue = React.useMemo( callback, array );
```

활용 예)

```
const memoizedValue = React.useMemo(
  () => compute(a, b),          callback
  [a, b]
);
```



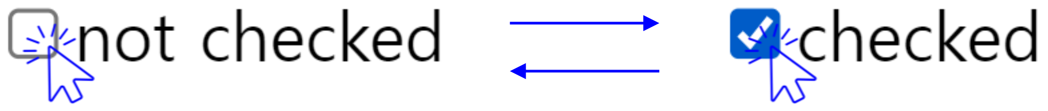
```
Insert

• 10
• 20

Average: 15
```

```
calculating average.. ▶Array(0)
calculating average.. ▶[10]          첫 번째 값 입력 완료: 10
calculating average.. ▶(2) [10, 20]  입력 완료: 10, 20
>
```

useReducer()

```
/* ch07-05-1.html */

const Checkbox = () => {
  const [checked, setCheck] = React.useState(false);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => (
          setCheck( checked => !checked )
        )} />
      {checked ? "checked" : "not checked"}
    </>
  );
};

const root =
  ReactDOM.createRoot(document.getElementById('root'));
root.render(<Checkbox />);
```

상태 변경함수를 호출하면서 직접 값을 변경



```
/* ch07-05-2.html */

const Checkbox = () => {
  const [checked, setCheck] = React.useState(false);
  const toggle = () => setCheck( checked => !checked );

  return (
    <>
      <input  type="checkbox" value={checked}
        onChange={toggle} />
      {checked ? "checked" : "not checked"}
    </>
  );
};
```

useReducer()

https://ko.reactjs.org/docs/hooks-reference.html#usereducer

```
const [state, dispatch] = React.useReducer(
  reducer,
  initialArg,
  init
);
```

```
const reducer = (state, action) => newState;
```

- useReducer(): 현재 상태와 액션을 전달받아 새로운 상태를 반환하는 Hooks

  ※ 액션: 상태 변경을 위해 필요한 정보를 담은 callback

```
/* ch07-05-3.html */
const Checkbox = () => {
  const [checked, toggle] = React.useReducer(
    checked => !checked,
    false
  );
  return (
    <>
      <input  type="checkbox" value={checked}
        onChange={toggle} />
      {checked ? "checked" : "not checked"}
    </>
  );
};
```

reducer():
현재의 상태 checked를
새로운 상태 !checked로 변경

```
/* ch07-05-4.html */

const Checkbox = () => {
  const reducer = a => !a;

  const [checked, toggle] = React.useReducer(
    reducer,
    false
  );
  ...
```

reducer()를
별도의 함수로 정의하여 활용

useReducer()

```
const [state, dispatch] = React.useReducer(
  reducer, initialArg, init
);
```

```
const reducer = (state, action) => newState;
```

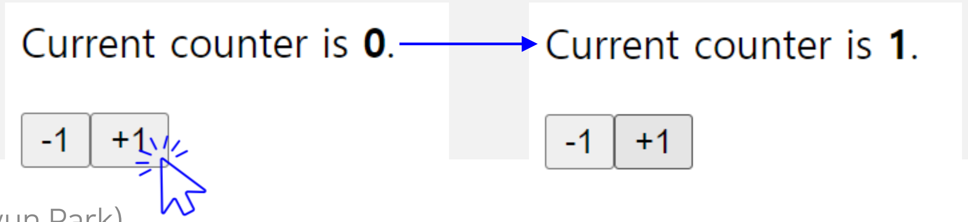
```
/* ch07-06-1.html */

const Counter = () => {
  const [value, setValue] = React.useState(0);

  return (
    <>
      <p>Current counter is <b>{value}</b>.</p>
      <button onClick={() => setValue(value-1)}>-1</button>
      <button onClick={() => setValue(value+1)}>+1</button>
    </>
  );
};

const root =
  ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
```

```
/* ch07-06-2.html */

const Counter = () => {
  const reducer = (state, action) => {
    switch (action.type) {
      case 'DECREMENT':
        return { value: state.value - 1 };
      case 'INCREMENT':
        return { value: state.value + 1 };
      default:
        return state;
    }
  };
  const [state, dispatch] = React.useReducer(reducer, { value: 0 });

  return (
    <> ...
     <button onClick={() => dispatch({type:'DECREMENT'})}>-1</button>
     <button onClick={() => dispatch({type:'INCREMENT'})}>+1</button>
    </>
  );
};
```

새로운 상태에 적용할 액션 type 반환

초기 상태: { value: 0 }

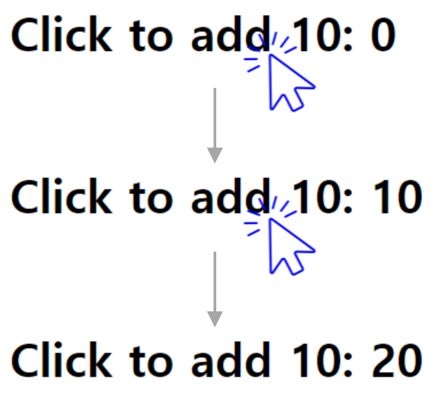
액션 type: DECREMENT


Current counter is **0**. → Current counter is **1**.

useReducer()

```
/* ch07-07-1.html */

const Adder = () => {
  const reducer = (number, nextNumber) => number + nextNumber;
  const [number, setNumber] = React.useReducer(reducer, 0);

  const unit = 10;

  return (
    <h1 onClick={() => setNumber(unit)}>
      Click to add {unit}: {number}
    </h1>
  );
}

const root =
  ReactDOM.createRoot(document.getElementById('root'));
root.render(<Adder />);
```

reducer():
array.reduce(*callback*, *initValue*)와 유사한 형태


Click to add 10: 0
Click to add 10: 10
Click to add 10: 20

```
const numbers = [28, 34, 67, 68];
const adder = numbers =>
  numbers.reduce(
    (prevValue, crntValue) => prevValue + crntValue,
    0
  );
console.log(`Sum of ${numbers} is ${adder(numbers)}`);
```

callback

initValue

array.reduce(*callback*, *initValue*)

callback parameters:
> accumulator, current value, ...

학습 정리: 7장. 훅스로 컴포넌트 개선하기

- 리액트 컴포넌트 라이프 사이클
- useState()
- useEffect()
- useMemo()
- useReducer()
- useCallback(), useContext(), custom Hooks