# Web Programming

## React programming

# Hook

## Hook 등장 배경

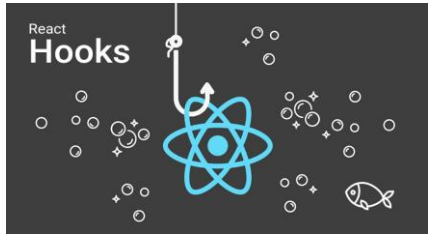
- 리액트 컴포넌트는 함수형 컴포넌트(Functional Component)와 클래스형 컴포넌트(Class Component)로 나뉜다.
- 리액트 초기에는 일반적으로 함수형 컴포넌트(Functional Component)를 사용하였으나, 값의 상태를 관리(state) 혹은 Lice Cycle Method(생명 주기=컴포넌트가 생성되고 사라지는 과정이 존재 할 때)를 사용하여야 할 때에만 클래스형 컴포넌트(Class Component)를 사용하였다.
- 함수형 컴포넌트(Functional Component)가 사용된 이유는 아래와 같은 클래스형 컴포넌트(Class Component)의 대표적인 단점 때문이였다.
  - ✓ 코드의 구성이 어렵고 Component의 재사용성이 떨어진다.
  - ✓ 컴파일 단계에서 코드 최적화를 어렵게 한다.
  - ✓ 최신 기술의 적용이 효과적이지 않다.
- 이러한 클래스형 컴포넌트(Class Component)의 단점을 보완하여, 함수형 컴포넌트(Functional Component)를 사용 할 수 있도록 등장한 것이 바로 React Hook(리액트 훅)이다

# Hook

## Hook 개요

- React Hook 이란, 리액트 v16.8에 새로 도입된 기능으로, 함수형 컴포넌트(Functional Component)에서 사용되는 몇가지 기술들을 일컫는다.
- 함수형 컴포넌트(Functional Component)가 클래스형 컴포넌트(Class Component)의기능을 사용 할 수 있도록 해주며 대표적인 예로는 useState, useEffect 등이 존재한다.

- Class
  - 클래스 컴포넌트에서는 생성자에서 state를 정의하고 setState함수를 통해 state를 업데이트하게 된다.
- Function
  - 기존 함수 컴포넌트는 이러한 state를 정의해서 사용하거나 컴포넌트 생명주기에 맞춰 실행되도록 할 수 없기 때문에 나온 것이 바로 훅(Hook)이다.
  - 이러한 훅의 이름은 모두 use로 시작한다.



# Hook

## Hook 장점

- 상태 로직 단순화 :
  - ✓ Hooks를 사용하면 함수형 컴포넌트에 상태를 추가하여 전반적인 로직을 단순화하고 코드를 이해하기 쉽게 만들 수 있다.

- 코드 재사용성과 관심사 분리 :
  - ✓ Hooks를 사용하면 컴포넌트 계층 구조를 변경하지 않고도 여러 컴포넌트 간에 상태 로직을 재사용할 수 있다.

- 사이드 이펙트 감소 :
  - ✓ Hooks는 함수형 컴포넌트에 생명주기 메서드와 유사한 기능을 제공하여 사이드 이펙트를 더 효율적으로 처리할 수 있습니다.

성능 비교 - 클래스형 vs 함수형　　React Hook의 어두운면

# Hook

## Hook의 규칙

- Hook은 JavaScript 함수입니다. 하지만 Hook을 사용할 때는 두 가지 규칙을 준수해야 한다.
- 우리는 이러한 규칙들을 자동으로 강제하기 위한 linter 플러그인(Create React App에 기본적으로 포함되어 있음)을 제공하고 있다.

- **최상위(at the Top Level)에서만 Hook을 호출해야 한다.**

  반복문, 조건문 혹은 중첩된 함수 내에서 Hook을 호출하면 안됨. 대신 early return이 실행되기 전에 항상 React 함수의 최상위(at the top level)에서 Hook을 호출해야 한다. 이 규칙을 따르면 컴포넌트가 렌더링 될 때마다 항상 동일한 순서로 Hook이 호출되는 것이 보장된다. 이러한 점은 React가 useState 와 useEffect 가 여러 번 호출되는 중에도 Hook의 상태를 올바르게 유지할 수 있도록 해준다.


- **오직 React 함수 내에서 Hook을 호출해야 한다.**

  Hook을 일반적인 JavaScript 함수에서 호출하지 마세요. 대신 아래와 같이 호출할 수 있다.

  - ✓ React 함수 컴포넌트에서 Hook을 호출
  - ✓ Custom Hook에서 Hook을 호출

  이 규칙을 지키면 컴포넌트의 모든 상태 관련 로직을 소스코드에서 명확하게 보이도록 할 수 있습니다.

# Hook

## Hook의 규칙

```
function Counter() {
  // ✅ Good: top-level in a function component
  const [count, setCount] = useState(0);
  // ...
}

function useWindowWidth() {
  // ✅ Good: top-level in a custom Hook
  const [width, setWidth] = useState(window.innerWidth);
  // ...
}
```

```
function FriendList() {
  const [onlineStatus, setOnlineStatus] = useOnlineStatus(); // ✅
}

function setOnlineStatus() { // ❌ Not a component or custom Hook!
  const [onlineStatus, setOnlineStatus] = useOnlineStatus();
}
```

Don't call Hooks from regular JavaScript functions. Instead, you can:

Call Hooks from React function components.
Call Hooks from custom Hooks.

# Hook

## Hook의 규칙

- Do not call Hooks inside conditions or loops.
- Do not call Hooks after a conditional return statement.
- Do not call Hooks in event handlers.
- Do not call Hooks in class components.
- Do not call Hooks inside functions passed to useMemo, useReducer, or useEffect.
- Do not call Hooks inside try/catch/finally blocks.

```
function Bad({ cond }) {
  if (cond) {
    // 🔴 Bad: inside a condition (to fix, move it outside!)
    const theme = useContext(ThemeContext);
  }
  // ...
}

function Bad() {
  for (let i = 0; i < 10; i++) {
    // 🔴 Bad: inside a loop (to fix, move it outside!)
    const theme = useContext(ThemeContext);
  }
  // ...
}
```

```
function Bad({ cond }) {
  if (cond) {
    return;
  }
  // 🔴 Bad: after a conditional return (to fix, move it before the return!)
  const theme = useContext(ThemeContext);
  // ...
}

function Bad() {
  function handleClick() {
    // 🔴 Bad: inside an event handler (to fix, move it outside!)
    const theme = useContext(ThemeContext);
  }
  // ...
}
```

# Hook

## Hook의 규칙

```
function Bad() {
  try {
    // 🔴 Bad: inside try/catch/finally block (to fix, move it outside!)
    const [x, setX] = useState(0);
  } catch {
    const [x, setX] = useState(1);
  }
}
```

```
function Bad() {
  const style = useMemo(() => {
    // 🔴 Bad: inside useMemo (to fix, move it outside!)
    const theme = useContext(ThemeContext);
    return createStyle(theme);
  });
  // ...
}

class Bad extends React.Component {
  render() {
    // 🔴 Bad: inside a class component (to fix, write a function component instead of a class!)
    useEffect(() => {})
    // ...
  }
}
```

# Hook

## Built-in React Hooks

| Hook | 종류 | | | |
|---|---|---|---|---|
| State Hooks | useState | useReducer | | |
| Context Hooks | useContext | | | |
| Ref Hooks | useRef | useImperativeHandle | | |
| Effect Hooks | useEffect | useLayoutEffect | useInsertionEffect | |
| Performance Hooks | useMemo | useCallback | useTransition | useDeferredValue |
| Resource Hooks | use | | | |
| Other Hooks | useDebugValue | useId | useSyncExternalStore | |
| Your own Hooks | custom Hooks | | | |

# useState

## useState

다시 작업 예정

- 컴포넌트에 state variable를 추가할 수 있다.

```
const [state, setState] = useState(initialState)
```

## Reference

- useState(initialState)
  - 상태 변수를 선언하려면 구성 요소의 최상위 수준에서 useState를 호출한다.

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


function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

  - [something, setSomething]과 같은 상태 변수의 이름은 array destructuring을 이용한다.

# useState

## Reference

- Parameters
  - initialState: 상태의 초기값. 모든 유형의 값이 될 수 있지만 함수에는 특별한 동작이 있다. 이 인수는 초기 렌더링 후에는 무시된다.
  - initialState는 초기화 함수처럼 처리된다. 순수해야 하고, 인수를 취하지 않아야 하며, 모든 유형의 값을 반환해야 한다. React는 컴포넌트를 초기화할 때 초기화 함수를 호출하고 반환 값을 초기 상태로 저장한다.
- Returns
  - useState 두 개의 값이 있는 배열을 반환한다.
    - 현재 상태. 첫 번째 렌더링 중에는 initialState 값이다.
    - Set 함수. 상태를 다른 값으로 다시 업데이트하고, 리랜더링을 트리거할 수 있는 함수이다.
- Caveats
  - useState는 Hook이므로 컴포넌트의 최상위 수준이나 자체 Hook에서만 호출할 수 있다. 루프나 조건 내에서는 호출할 수 없다. 필요한 경우 새 구성 요소를 추출하고 상태를 해당 구성 요소로 옮긴다.
  - Strict 모드에서 React는 실수로 발생한 불순물을 찾는 데 도움을 주기 위해 초기화 함수를 두 번 호출 한다. 이는 개발 전용 동작이며 프로덕션에는 영향을 주지 않는다. 초기화 함수가 순수(순수해야 함)인 경우 이는 동작에 영향을 주지 않는다. 호출 중 하나의 결과는 무시된다.

# useState

## Reference

- set functions, like setSomething(nextState)
  - useState에 의해 반환되는 set 함수는 상태를 다른 값으로 업데이트하고 리렌더링을 트리거할 수 있다. 다음 상태를 직접 전달하거나 이전 상태에서 이를 계산하는 함수를 전달할 수 있다.

```
const [name, setName] = useState('Edward');

function handleClick() {
  setName('Taylor');
  setAge(a => a + 1);
  // ...
```

- Parameters
  - useState에 의해 반환되는 set 함수는 상태를 다른 값으로 업데이트하고 리렌더링을 트리거할 수 있다. 다음 상태를 직접 전달하거나 이전 상태에서 이를 계산하는 함수를 전달할 수 있다.

# useState

## Reference

- Parameters
  - nextState: 원하는 상태 값. 모든 유형의 값이 될 수 있지만 함수에는 특별한 동작을 수행한다.
  - nextState를 실행하면 updater function처럼 처리된다. 순수해야 하고 보류중인(pendding) 상태를 유일한 인수로 사용해야 하며 다음 상태를 반환해야 한다. React는 업데이트 기능을 대기열에 넣고 구성 요소를 다시 렌더링한다. 다음 렌더링 동안 React는 대기 중인 모든 업데이트를 이전 상태에 적용하여 다음 상태를 계산한다.
- Returns
  - Set 함수는 리턴 값이 없다.

# useState

## Reference

- Caveats(주의사항)
  - Set 함수는 다음 렌더링에 대한 상태 변수만 업데이트한다. set함수를 호출한 후 상태 변수를 읽으면 호출하기 전에 화면에 있었던 이전 값을 계속 얻을 수 있다 .
  - Object.is 비교를 통해 확인한 바와 같이 제공하는 새 값이 현재 상태와 동일한 경우 React는 구성 요소 및 구성 요소의 하위 항목을 re-rendering하는 것을 건너뛴다.. 이것은 최적화이다. 경우에 따라 React는 하위 요소를 건너뛰기 전에 구성 요소를 호출해야 할 수도 있지만 코드에 영향을 주지는 않는다.
  - React는 상태 업데이트를 일괄 처리한다. 모든 이벤트 핸들러가 실행 되고 해당 set 함수를 호출한 후에 화면을 업데이트한다. 이렇게 하면 단일 이벤트 중에 여러 번 리렌더링되는 것을 방지할 수 있다. 예를 들어 DOM에 액세스하기 위해 React를 강제로 먼저 화면을 업데이트해야 하는 경우에는 flushSync를 사용할 수 있다.
  - 렌더링 중 set 함수 호출은 현재 렌더링 구성 요소 내에서만 허용된다. React는 출력을 삭제하고 즉시 새 상태로 다시 렌더링을 시도한다. 이 패턴은 거의 필요하지 않지만 이전 렌더링의 정보를 저장 하는 데 사용할 수 있다.
  - Strict 모드에서 React는 우발적인 불순물을 찾는데 도움을 주기 위해 업데이트 기능을 두 번 호출 한다. 이는 개발 전용 동작이며 프로덕션에는 영향을 주지 않는다. 업데이터 함수가 순수하다면(있는 그대로) 동작에 영향을 주지 않아야 한다. 호출 중 하나의 결과는 무시된다.

# useState

## Usage

- Adding state to a component(구성요소에 상태 추가)
  - useState하나 이상의 상태 변수를 선언하려면 구성 요소의 최상위 수준에서 호출하라 .

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

function MyComponent() {
  const [age, setAge] = useState(42);
  const [name, setName] = useState('Taylor');
  // ...
```

  - array destructuring를 사용하여 [Something, setSomething]과 같은 상태 변수의 이름을 지정한다.
  - useState정확히 두 개의 항목이 포함된 배열을 반환한다.
  - 이 상태 변수의 현재 상태는 처음에 사용자가 제공한 초기 상태로 설정된다 .
  - set 함수는 상호 작용에 대한 응답으로 다른 값으로 변경할 수 있다.
  - 화면의 내용을 업데이트하려면 다음 상태 값을 가진 set 함수를 호출하면 된다.

# useState

## Usage

- Adding state to a component

```
function handleClick() {
  setName('Robin');
}
```

- React는 다음 상태를 저장하고, 새 값으로 구성 요소를 다시 렌더링하고, UI를 업데이트한다.

Pitfall(함정)

set 함수를 호출해도 이미 실행 중인 코드의 현재 상태는 변경 되지 않는다 .

```
function handleClick() {
  setName('Robin');
  console.log(name); // Still "Taylor"!
}
```

다음 useState 렌더링부터 반환되는 항목에만 영향을 미친다 .

# useState

## Basic useState examples

- Counter (number)
  - count 상태 변수는 숫자를 유지한다. 버튼을 클릭하면 증가한다.

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

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
    </button>
  );
}
```

You pressed me 6 times

# useState

## Basic useState examples

- Text field (string)
  - text 상태 변수는 문자열을 유지한다. text에 입력을 하면 브라우저 입력 DOM 요소에서 최신 입력 값을 읽고 setText를 호출하여 상태를 업데이트한다. 이를 통해 현재 텍스트를 아래에 표시할 수 있다.

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

export default function MyInput() {
  const [text, setText] = useState('hello');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <>
      <input value={text} onChange={handleChange} />
      <p>You typed: {text}</p>
      <button onClick={() => setText('hello')}>
        Reset
      </button>
    </>
  );
}
```

# useState

## Basic useState examples

- Checkbox (boolean)
  - liked 상태 변수는 부울(boolean) 값을 유지한다. Input(checkbox)을 클릭하면 setLiked는 브라우저 checkbox input 이 선택되었는지 여부로 liked 상태 변수를 업데이트한다. liked 변수는 checkbox 아래에 텍스트를 렌더링하는 데 사용된다.

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

export default function MyCheckbox() {
  const [liked, setLiked] = useState(true);

  function handleChange(e) {
    setLiked(e.target.checked);
  }
```

```
  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={liked}
          onChange={handleChange}
        />
        I liked this
      </label>
      <p>You {liked ? 'liked' : 'did not like'} this.</p>
    </>
  );
}
```

# useState

## Basic useState examples

- Form (two variables)
  - 동일한 component에서 둘 이상의 상태 변수를 선언할 수 있다. 각 상태 변수는 완전히 독립적이다.

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

export default function Form() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);

  return (
    <>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={() => setAge(age + 1)}>
        Increment age
      </button>
      <p>Hello, {name}. You are {age}.</p>
    </>
  );
}
```

# useState

## Updating state based on the previous state (이전 상태를 기준으로 상태 업데이트)

- age가 42라고 가정한다. 아래  핸들러는 setAge(age + 1)를 세 번 호출한다:

```
function handleClick() {
    setAge(age + 1); // setAge(42 + 1)
    setAge(age + 1); // setAge(42 + 1)
    setAge(age + 1); // setAge(42 + 1)
}
```

- 그러나 한 번 클릭하면 나이가 45세가 아닌 43세가 된다!
- set 함수를 호출한다고 해서 이미 실행 중인 코드의 age 상태 변수가 업데이트되는 것은 아니기 때문이다.
- 따라서 세 번의 setAge(age+1) 호출은 setAge(43)가 된다.

# useState

## Updating state based on the previous state

- 다음 상태(next state)가 아닌 updater function을 setAge에 전달하여 앞의 문제를 해결할 수 있다.

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

- 여기서 a => a + 1은 updater function이다. 이는 보류 상태(pending state)를 취하고 그것으로부터 다음 상태를 계산한다.

# useState

## Updating state based on the previous state

- React는 updater function을 queue에 넣는다. 그런 다음, 다음 렌더링하는 동안 그것들을 같은 순서로 호출한다:

- a => a + 1은 42를 보류 상태로 받고 43을 다음 상태로 반환한다.
- a => a + 1은 43을 보류 상태로 받고 44를 다음 상태로 반환한다.
- a => a + 1은 44를 보류 상태로 받고 45를 다음 상태로 반환한다.

- 대기 중인 다른 업데이트는 없으므로 React는 최종적으로 45를 현재 상태로 저장한다.
- 관례적으로 상태 변수 이름의 첫 글자에 대해 보류 중인 상태 인수의 이름을 age의 경우 a와 같이 지정하는 것이 일반적이다. 그러나 prevAge 또는 더 명확한 다른 이름으로 지정할 수도 있다.
- React는 개발 모드에서 updater를 두 번 호출하여 updater가 순수한지를 확인한다.

# useState

## The difference between passing an updater and passing the next state directly example

- Passing the next state directly
  - Updater function을 패스하지 못하므로 " +3 " 버튼이 의도한 대로 작동하지 않는다.

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

export default function Counter() {
  const [age, setAge] = useState(42);

  function increment() {
    setAge(age + 1);
  }
```

```
  return (
    <>
      <h1>Your age: {age}</h1>
      <button onClick={() => {
        increment();
        increment();
        increment();
      }}>+3</button>
      <button onClick={() => {
        increment();
      }}>+1</button>
    </>
  );
}
```

# useState

## The difference between passing an updater and passing the next state directly example

- Passing the updater function
  - Updater function을 통과하므로 "+3" 버튼이 작동한다.

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

export default function Counter() {
  const [age, setAge] = useState(42);

  function increment() {
    setAge(a => a + 1);
  }
```

```
  return (
    <>
      <h1>Your age: {age}</h1>
      <button onClick={() => {
        increment();
        increment();
        increment();
      }}>+3</button>
      <button onClick={() => {
        increment();
      }}>+1</button>
    </>
  );
}
```

# useState

## Updating state based on the previous state

- 상태(state) 안에 객체와 배열을 만들 수 있습니다.
- React에서 상태는 읽기 전용으로 간주되므로 기존 객체를 변경하는 대신 대체해야 한다.
- 예를 들어, 상태에 form 객체가 있는 경우에는 객체를 변경하지 마라:

```
// ⚑ Don't mutate an object in state like this:
form.firstName = 'Taylor';
```

- 대신 새 객체를 생성하여 전체 객체를 바꾼다:

```
// ✓ Replace state with a new object
setForm({
  ...form,
  firstName: 'Taylor'
});
```

Read updating objects in state and updating arrays in state to learn more.

# useState

## Examples of objects and arrays in state

- Form (object)
  - form 상태 변수는 객체를 유지한다. 각 input은 전체 form을 다음 상태로 변경하는 setForm 호출하는 change handle가 있다. {...form} 확산 구문(spread syntax)을 사용하면 상태 객체가 변경되지 않고 대체된다.

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

export default function Form() {
  const [form, setForm] = useState({
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com',
  });
```

```
return (
  <>
    <label>
      First name:
      <input
        value={form.firstName}
        onChange={e => {
          setForm({
            ...form,
            firstName: e.target.value
          });
        }}
      />
```

```
</label>
      <label>
        Last name:
        <input
          value={form.lastName}
          onChange={e => {
            setForm({
              ...form,
              lastName: e.target.value
            });
          }}
        />
      </label>
```

# useState

## Examples of objects and arrays in state

```
  <label>
        Email:
        <input
          value={form.email}
          onChange={e => {
            setForm({
              ...form,
              email: e.target.value
            });
          }}
        />
      </label>
      <p>
        {form.firstName}{' '}
        {form.lastName}{' '}
        ({form.email})
      </p>
    </>
  );
}-
```

# useState

## Examples of objects and arrays in state

- Form (nested object)
  - 상태가 중첩된다. 중첩된 상태를 업데이트할 때 업데이트할 객체의 복사본 뿐만 아니라 위쪽으로 " 포함 " 되는 객체도 만들어야 한다.

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

export default function Form() {
  const [person, setPerson] = useState({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg',
    }
  });

  function handleNameChange(e) {
    setPerson({
      ...person,
      name: e.target.value
    });
  }
```

```
function handleTitleChange(e) {
  setPerson({
    ...person,
    artwork: {
      ...person.artwork,
      title: e.target.value
    }
  });
}

function handleCityChange(e) {
  setPerson({
    ...person,
    artwork: {
      ...person.artwork,
      city: e.target.value
    }
  });
}
```

Read updating a nested object to learn more.

# useState

## Examples of objects and arrays in state

```
function handleImageChange(e) {
  setPerson({
    ...person,
    artwork: {
      ...person.artwork,
      image: e.target.value
    }
  });
}

return (
  <>
    <label>
      Name:
      <input
        value={person.name}
        onChange={handleNameChange}
      />
    </label>
```

```
    <label>
      Title:
      <input
        value={person.artwork.title}
        onChange={handleTitleChange}
      />
    </label>
    <label>
      City:
      <input
        value={person.artwork.city}
        onChange={handleCityChange}
      />
    </label>
    <label>
      Image:
      <input
        value={person.artwork.image}
        onChange={handleImageChange}
      />
    </label>
```

```
    <p>
      <i>{person.artwork.title}</i>
      {' by '}
      {person.name}
      <br />
      (located in {person.artwork.city})
    </p>
    <img
      src={person.artwork.image}
      alt={person.artwork.title}
    />
  </>
);
}
```

# useState

## Examples of objects and arrays in state

- List (array)
  - todos 상태 변수는 배열을 유지합니다. 각 버튼 처리기는 해당 배열의 다음 버전으로 setTodos를 호출한다. [...todos] 스프레드 구문, todos.map() 및 todos.filter()는 상태 배열이 변경되지 않고 대체된다.

App.js

```
import { useState } from 'react';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [
  { id: 0, title: 'Buy milk', done: true },
  { id: 1, title: 'Eat tacos', done: false },
  { id: 2, title: 'Brew tea', done: false },
];

export default function TaskApp() {
  const [todos, setTodos] = useState(initialTodos);
```

App.js

```
  function handleAddTodo(title) {
    setTodos([
      ...todos,
      {
        id: nextId++,
        title: title,
        done: false
      }
    ]);
  }
  function handleChangeTodo(nextTodo) {
    setTodos(todos.map(t => {
      if (t.id === nextTodo.id) {
        return nextTodo;
      } else {
        return t;
      }
    }));
  }
```

# useState

## Examples of objects and arrays in state

App.js

```
  function handleDeleteTodo(todoId) {
    setTodos(
      todos.filter(t => t.id !== todoId)
    );
  }

  return (
    <>
      <AddTodo
        onAddTodo={handleAddTodo}
      />
      <TaskList
        todos={todos}
        onChangeTodo={handleChangeTodo}
        onDeleteTodo={handleDeleteTodo}
      />
    </>
  );
}
```

AddTodo.js

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

export default function AddTodo({ onAddTodo }) {
  const [title, setTitle] = useState('');
  return (
    <>
      <input
        placeholder="Add todo"
        value={title}
        onChange={e => setTitle(e.target.value)}
      />
      <button onClick={() => {
        setTitle('');
        onAddTodo(title);
      }}>Add</button>
    </>
  )
}
```

# useState

## Examples of objects and arrays in state

TakeList.js

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

export default function TaskList({
  todos,
  onChangeTodo,
  onDeleteTodo
}) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <Task
            todo={todo}
            onChange={onChangeTodo}
            onDelete={onDeleteTodo}
          />
        </li>
      ))}
    </ul>
  );
}
```

```
function Task({ todo, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let todoContent;
  if (isEditing) {
    todoContent = (
      <>
        <input
          value={todo.title}
          onChange={e => {
            onChange({
              ...todo,
              title: e.target.value
            });
          }} />
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
```

# useState

## Examples of objects and arrays in state

```
} else {
   todoContent = (
     <>
      {todo.title}
      <button onClick={() => setIsEditing(true)}>
        Edit
      </button>
     </>
   );
 }
```

```
return (
   <label>
     <input
       type="checkbox"
       checked={todo.done}
       onChange={e => {
         onChange({
           ...todo,
           done: e.target.checked
         });
       }}
     />
     {todoContent}
     <button onClick={() => onDelete(todo.id)}>
       Delete
     </button>
   </label>
 );
}
```

# useState

## Examples of objects and arrays in state

- Writing concise update logic with Immer(Immer로 간결한 업데이트 로직 작성)
  - 변경되지 않는 배열과 객체를 업데이트하는 것이 지루하게 느껴진다면, 반복되는 코드를 줄이기 위해 Immer와 같은 라이브러리를 사용할 수 있다. Immer는 객체를 변환하는 것처럼 간결한 코드를 작성할 수 있게 해주지만, 후드(hood) 아래에서는 불변의 업데이트를 수행한다:

package.json

```
{
  "dependencies": {
    "immer": "1.7.3",
    "react": "latest",
    "react-dom": "latest",
    "react-scripts": "latest",
    "use-immer": "0.5.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  },
  "devDependencies": {}
}
```

App.js

```
import { useState } from 'react';
import { useImmer } from 'use-immer';

let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];
```

# useState

## Examples of objects and arrays in state

```
export default function BucketList() {
  const [list, updateList] = useImmer(initialList);

  function handleToggle(artworkId, nextSeen) {
    updateList(draft => {
      const artwork = draft.find(a =>
        a.id === artworkId
      );
      artwork.seen = nextSeen;
    });
  }

  return (
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
      <ItemList
        artworks={list}
        onToggle={handleToggle} />
    </>
  );
}
```

```
function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

# useState

## Avoiding recreating the initial state(초기 상태를 다시 만드는 것을 피함)

- React는  초기 상태를 한 번 저장하고 다음 렌더링에서 무시한다.

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos());
  // ...
```

- createInitialTodos()의 결과는 초기 렌더에만 사용되지만, 여전히 모든 렌더에서 이 함수를 호출하고 있다. 큰 배열을 생성하거나 복잡한 계산을 수행하는 경우에는 이 함수가 낭비될 수 있다.
- 이를 해결하기 위해 useState에 초기화 함수(initializer function)로 전달할 수 있다:

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  // ...
```

- .함수 자체인 createInitialTodos를 전달(passing)한 것이지 호출한 결과인 createInitialTodos()를 전달한 것은 아니다. 함수를 useState로 전달하면 초기화 중에만 React가 이 함수를 호출한다.

# useState

## The difference between passing an initializer and passing the initial state directly example

- Passing the initializer function(이니셜라이저 함수 패싱)
  - initializer 함수를 전달하므로 createInitialTodos 함수는 초기화 중에만 실행된다. 입력을 입력할 때와 같이 구성 요소가 다시 렌더링 될 때는 실행되지 않는다.

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

function createInitialTodos() {
  const initialTodos = [];
  for (let i = 0; i < 50; i++) {
    initialTodos.push({
      id: i,
      text: 'Item ' + (i + 1)
    });
  }
  return initialTodos;
}

export default function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  const [text, setText] = useState('');
```

```
  return (
    <>
      <input
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={() => {
        setText('');
        setTodos([{
          id: todos.length,
          text: text
        }, ...todos]);
      }}>Add</button>
```

```
      <ul>
        {todos.map(item => (
          <li key={item.id}>
            {item.text}
          </li>
        ))}
      </ul>
    </>
  );
}
```

# useState

## The difference between passing an initializer and passing the initial state directly example

- Passing the initial state directly(초기 상태를 직접 패싱)
  - 이니셜라이저 함수를 전달하지 않으므로 input에 입력할 때와 같은 모든 렌더에서 createInitialTodos 함수가 실행된다. 동작에서 눈에 띄는 차이는 없지만 이 코드는 효율성이 떨어진다.

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

function createInitialTodos() {
  const initialTodos = [];
  for (let i = 0; i < 50; i++) {
    initialTodos.push({
      id: i,
      text: 'Item ' + (i + 1)
    });
  }
  return initialTodos;
}

export default function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos());
  const [text, setText] = useState('');
```

```
return (
  <>
    <input
      value={text}
      onChange={e => setText(e.target.value)}
    />
    <button onClick={() => {
      setText('');
      setTodos([{
        id: todos.length,
        text: text
      }, ...todos]);
    }}>Add</button>
```

```
    <ul>
      {todos.map(item => (
        <li key={item.id}>
          {item.text}
        </li>
      ))}
    </ul>
  </>
);
}
```

# useState

## Resetting state with a key(키로 상태 재설정)

- 목록을 렌더링할 때 키(key) 속성을 자주 접하게 된다. 하지만 이는 또 다른 목적을 제공하기도 한다.
- 구성 요소에 다른 키를 전달하여 구성 요소의 상태를 재설정할 수 있다. 이 예에서 Reset 버튼은 상태 변수의 버전을 변경하며, 이를 Form의 키로 전달한다. 키가 변경되면 React는 Form 구성 요소(및 모든 하위 구성 요소)를 처음부터 다시 생성하므로 상태가 재설정된다.

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

export default function App() {
  const [version, setVersion] = useState(0);

  function handleReset() {
    setVersion(version + 1);
  }

  return (
    <>
      <button onClick={handleReset}>Reset</button>
      <Form key={version} />
    </>
  );
}
```

```
function Form() {
  const [name, setName] = useState('Taylor');

  return (
    <>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <p>Hello, {name}.</p>
    </>
  );
}
```

Read preserving and resetting state to learn more.

# useState

## Storing information from previous renders(이전 렌더링에서 정보 저장)

- 일반적으로 이벤트 핸들러에서 상태를 업데이트한다. 그러나 드물게 렌더링에 응답하여 상태를 조정할 수도 있다. 예를 들어 prop이 변경될 때 상태 변수를 변경할 수도 있다.

- 대부분의 경우 다음과 같은 작업이 필요하지 않습니다:
  - 필요한 값을 현재 props나 다른 상태에서 완전히 계산할 수 있다면 그 중복 상태를 완전히 제거하라. 너무 자주 다시 계산하는 것이 걱정된다면 useMemo Hook을 사용하는 것이 도움이 될 수 있다.
  - 전체 구성요소 트리의 상태를 재설정하려면 component에 다른 키를 전달한다.
  - 가능한 경우 이벤트 핸들러의 모든 관련 상태를 업데이트한다.
  - 아주 드물게 이들 중 어느 것도 적용되지 않는 경우, 구성 요소가 렌더링되는 동안 설정 함수를 호출하여 지금까지 렌더링된 값을 기준으로 상태를 업데이트하는 데 사용할 수 있는 패턴이 있다.

# useState

## Storing information from previous renders

- 다음 예제가 있다. 이 CountLabel 구성 요소는 해당 구성 요소에 전달된 카운트 prop을 표시한다:

```
export default function CountLabel({ count }) {
  return <h1>{count}</h1>
}
```

- 카운터가 마지막으로 변경된 이후에 증가했는지 감소했는지 보여주고자 한다. 이전 값을 추적할려고 하지만 카운트 prop은 이것을 알려주지 않는다. prevCount 상태 변수를 추가하면 이전 값을 추적할 수 있다. trend라는 다른 상태 변수를 추가하여 카운트가 증가했는지 또는 감소했는지 여부를 유지한다. prevCount와 카운트를 비교하고, 카운터가 동일하지 않으면 prevCount와 trend를 모두 업데이트한다. 이제 현재 카운트 prop과 마지막 렌더링 이후에 어떻게 변경되었는지 모두 보여줄 수 있다.

# useState

## Storing information from previous renders

```
import { useState } from 'react';
import CountLabel from './CountLabel.js';

export default function App() {
  const [count, setCount] = useState(0);
  return (
    <>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
      <button onClick={() => setCount(count - 1)}>
        Decrement
      </button>
      <CountLabel count={count} />
    </>
  );
}
```

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

export default function CountLabel({ count }) {
  const [prevCount, setPrevCount] = useState(count);
  const [trend, setTrend] = useState(null);
  if (prevCount !== count) {
    setPrevCount(count);
    setTrend(count > prevCount ? 'increasing' : 'decreasing');
  }
  return (
    <>
      <h1>{count}</h1>
      {trend && <p>The count is {trend}</p>}
    </>
  );
}
```

# useState

## Storing information from previous renders

- 렌더링 중에 set function을 호출하면 조건 안에 prevCount !== count와 setPrevCount(count)와 같은 호출이 있어야 한다. 그렇지 않으면 component가 충돌할 때까지 루프에서 re-render가 발생한다. 또한 현재 렌더링 중인 구성 요소의 상태만 업데이트할 수도 있다. 렌더링 중에 다른 구성 요소의 set function을 호출하는 것은 오류다. 마지막으로 set 호출은 여전히 변하지 않고 상태를 업데이트해야 한다. 그렇다고 해서 다른 순수 함수 규칙을 어길 수 있는 것은 아니다.

- 이 패턴은 이해하기 어려울 수 있으며 일반적으로 피하는 것이 가장 좋다. 그러나 effect에서 상태를 업데이트하는 것보다 낫다. 렌더링 중에 set 함수를 호출하면 React는 return과 함께 component가 종료된 직후 해당 component를 다시 렌더링하고 자식들을 렌더링하기 전에 다시 렌더링한다. 이렇게 하면 자식들이 두 번 렌더링할 필요가 없다. 나머지 component 함수는 여전히 실행되고(그리고 결과는 버려진다). 조건이 모든 Hook 호출보다 아래에 있으면 조기 return을 추가하여 렌더링을 더 일찍 다시 시작할 수 있다.

# useState

## Troubleshooting(문제 해결)

- **I've updated the state, but logging gives me the old value**

  **(상태를 저장했지만 로깅이 이전 상태를 제공)**

  - set 함수를 호출해도 실행 코드의 상태는 변경되지 않는다:

```
function handleClick() {
  console.log(count);  // 0

  setCount(count + 1); // Request a re-render with 1
  console.log(count);  // Still 0!

  setTimeout(() => {
    console.log(count); // Also 0!
  }, 5000);
}
```

```
const nextCount = count + 1;
setCount(nextCount);

console.log(count);      // 0
console.log(nextCount); // 1
```

  - 이는 상태가 스냅샷처럼 동작하기 때문이다. 상태를 업데이트하면 새 상태 값을 가진 다른 렌더가 요청되지만 이미 실행 중인 이벤트 핸들러의 자바스크립트 카운트 변수에는 영향을 미치지 않는다.
  - 다음 상태를 사용해야 하는 경우 설정된 함수에 전달하기 전에 변수에 저장할 수 있다:

# useState

## Troubleshooting

- **I've updated the state, but the screen doesn't update**

  **(상태를 업데이트 했지만, 화면이 업데이트 되지 않음)**

  - Object.is 비교를 통해 다음 상태가 이전 상태와 같으면 React는 업데이트를 무시한다. 이는 일반적으로 상태의 개체 또는 배열을 직접 변경할 때 발생한다:

```
obj.x = 10;  // ▶ Wrong: mutating existing object
setObj(obj); // ▶ Doesn't do anything
```

  - 기존 개체를 변경한 후 setObj로 다시 전달했기 때문에 React는 업데이트를 무시한다. 이 문제를 해결하려면 객체 및 배열을 변경하는 대신 항상 상태가 유지되도록 해야 한다:

```
// ✓ Correct: creating a new object
setObj({
  ...obj,
  x: 10
});
```

# useState

## Troubleshooting

- **I've updated the state, but the screen doesn't update**
  - 다음과 같은 오류가 발생할 수 있다: 무한 루프를 방지하기 위해 리액트는 렌더링 횟수를 제한한다. 일반적으로 렌더링 중에 조건 없는 상태를 설정하므로 component가 루프에 들어간다: render, set state(렌더 원인), render, set state(렌더 원인) 등.
  - 이는 이벤트 처리기를 지정하는 실수로 인해 발생하는 경우가 많다:

```
// ⚑ Wrong: calls the handler during render
return <button onClick={handleClick()}>Click me</button>

// ✓ Correct: passes down the event handler
return <button onClick={handleClick}>Click me</button>

// ✓ Correct: passes down an inline function
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

  - 이 오류의 원인을 찾을 수 없는 경우 콘솔에서 오류 옆에 있는 화살표를 클릭하고 자바스크립트 스택을 훑어보면 오류의 원인이 되는 특정 설정 함수 호출을 찾을 수 있다.

# useState

## Troubleshooting

- **My initializer or updater function runs twice**

  **(Initializer나 updater 함수가 두 번 실행됨)**

  - Strict Mode에서 React는 일부 기능을 한 번이 아니라 두 번 호출한다:

```
function TodoList() {
  // This component function will run twice for every render.

  const [todos, setTodos] = useState(() => {
    // This initializer function will run twice during initialization.
    return createTodos();
  });

  function handleClick() {
    setTodos(prevTodos => {
      // This updater function will run twice for every click.
      return [...prevTodos, createTodo()];
    });
  }
  // ...
```

# useState

## Troubleshooting

- 이것은 예상되는 것이며 귀하의 코드를 위반해서는 안 된다.
- 이 개발 전용 동작은 구성 요소를 순수하게 유지하는 데 도움이 된다. 리액트는 한 번의 호출 결과를 사용하고, 다른 한 번의 호출 결과는 무시한다. component, initializer 및 updater 함수가 순수한(pure) 한, 이는 논리에 영향을 주지 않는다. 그러나 실수로 불순한(impure) 경우 이는 실수를 알아차리는 데 도움이 된다.
- 예를 들어, 이 impure updater 함수는 상태에 있는 배열을 변경한다:

```
setTodos(prevTodos => {
  // ▶ Mistake: mutating state
  prevTodos.push(createTodo());
});
```

- React는 updater 함수를 두 번 호출하기 때문에 todo가 두 번 추가된 것을 볼 수 있으므로 오류가 있음을 알 수 있다. 이 예에서는 배열을 변경하는 대신에 배열을 교체하여 오류를 해결할 수 있다:

```
setTodos(prevTodos => {
// ✓ Correct: replacing with new state
return [...prevTodos, createTodo()];
});
```

# useState

## Troubleshooting

- 이 업데이터 기능이 순수하기 때문에 추가 시간이라고 해서 행동에 차이가 나는 것은 아니다.
- 이것이 리액트가 실수를 찾는 데 도움이 되는 이유다.
- Component, initializer, updater 함수만 순수하면 된다.
- 이벤트 핸들러는 순수할 필요가 없으므로 React는 이벤트 핸들러를 두 번 호출하지 않는다.
- component를 순수하게 유지하는 방법을 읽어 보시오.

Read keeping components pure to learn more.

# useState

## Troubleshooting

- **I'm trying to set state to a function, but it gets called instead**

  **(함수를 상태에 설정하려고 하는데 호출이 됨)**

  - 다음과 같이 함수를 상태로 만들 수 없다:

```
const [fn, setFn] = useState(someFunction);

function handleClick() {
  setFn(someOtherFunction);
}
```

  - 함수를 전달하고 있기 때문에, React는 someFunction을 초기화 함수라고 가정하고, someOtherFunction을 updater 함수라고 가정하고, 결과를 저장하려고 한다. 실제로 함수를 저장하기 위해서는 두 경우 모두 그들 앞에 () =>를 놓아야 한다. 그러면 React는 당신이 전달한 함수를 저장할 것이다.

```
const [fn, setFn] = useState(() => someFunction);

function handleClick() {
  setFn(() => someOtherFunction);
}
```

## Report

레포트가 너무 많다는 의견이 있음 <- 의견 수렴

## 발표주제

클로저(Closure)
함수 합성
고차 함수

중간고사 다음 주 발표 주제

# Reference


- https://goddaehee.tistory.com/299
- https://wikidocs.net/197615
- https://react.dev/reference/react/useState

- 모던 자바스크립트 개발자를 위한 리액트 프로그래밍, 쿠지라히코우즈키에, 윤인성, 위키북스
- 명품 웹 프로그래밍, 황기태, 생능출판사
- 리액트 & 리액트 네이티브 통합 교과서, 아담 보두치, 강경일, 신희철, 에이콘
- Do it! 리액트 모던 웹 개발 with 타입스크립트, 전예홍, 이지스퍼블리싱