




실증 아카데미 리액트 특강

컴포넌트 렌더링, 최적화 하기

일시 2021년 01월 18일

주최 동아대학교



CONTENTS—

01 React.js의 생명주기

함수형 컴포넌트의 생명주기
useEffect

마운트와 언마운트

재렌더(업데이트)

04 반복 렌더링

JavaScript Array

Array.prototype.map

map 메서드를 이용한 반복 렌더링

02 커스텀 컴포넌트 작성

함수형 컴포넌트

클래스형 컴포넌트

05 상태 최적화 관리

useMemo

useCallback

memo

03 useState와 조건부 렌더링

useState

setState를 통한 조건부 렌더링





01

React.js의 생명주기

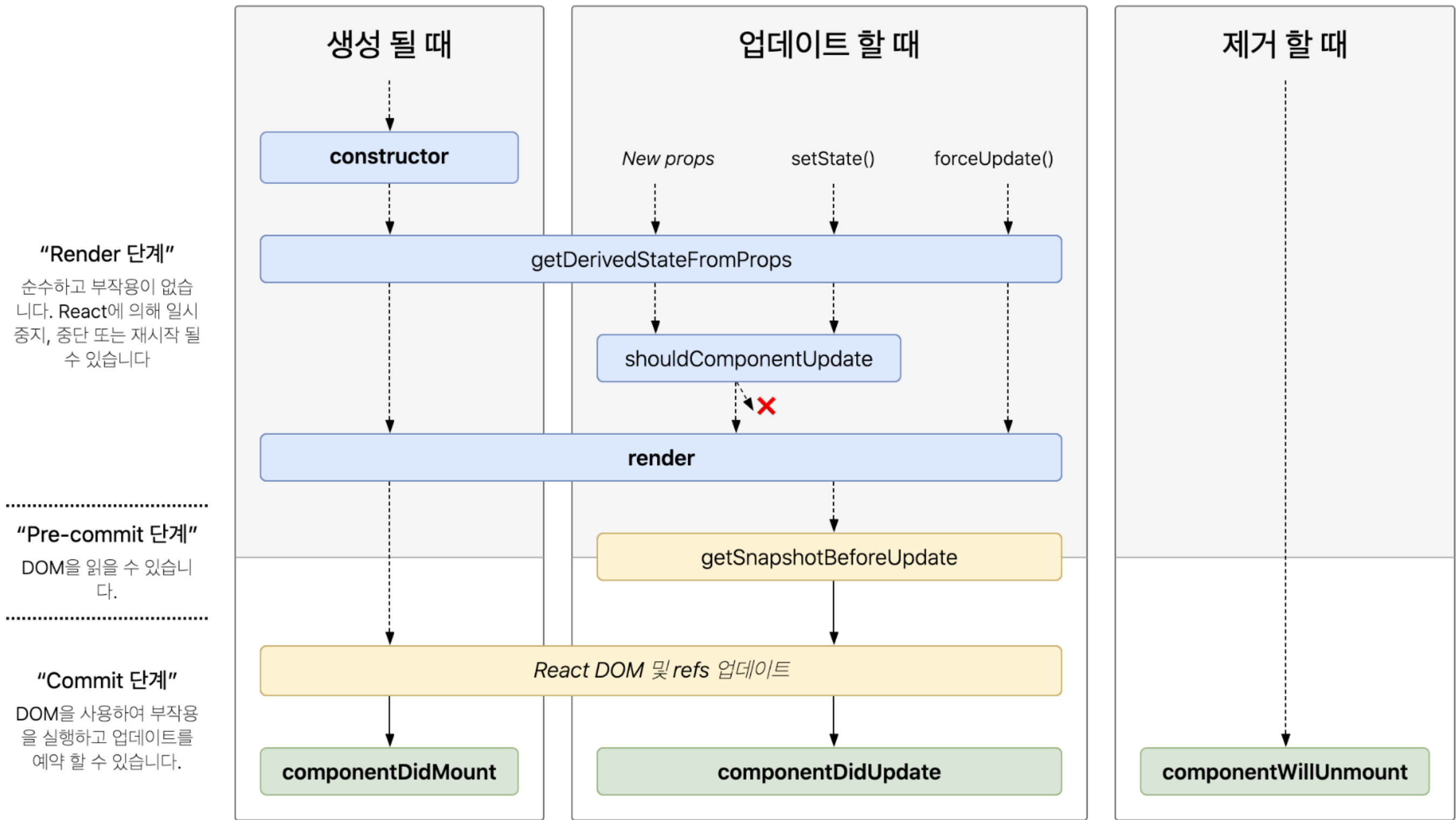
함수형 컴포넌트의 생명주기

useEffect

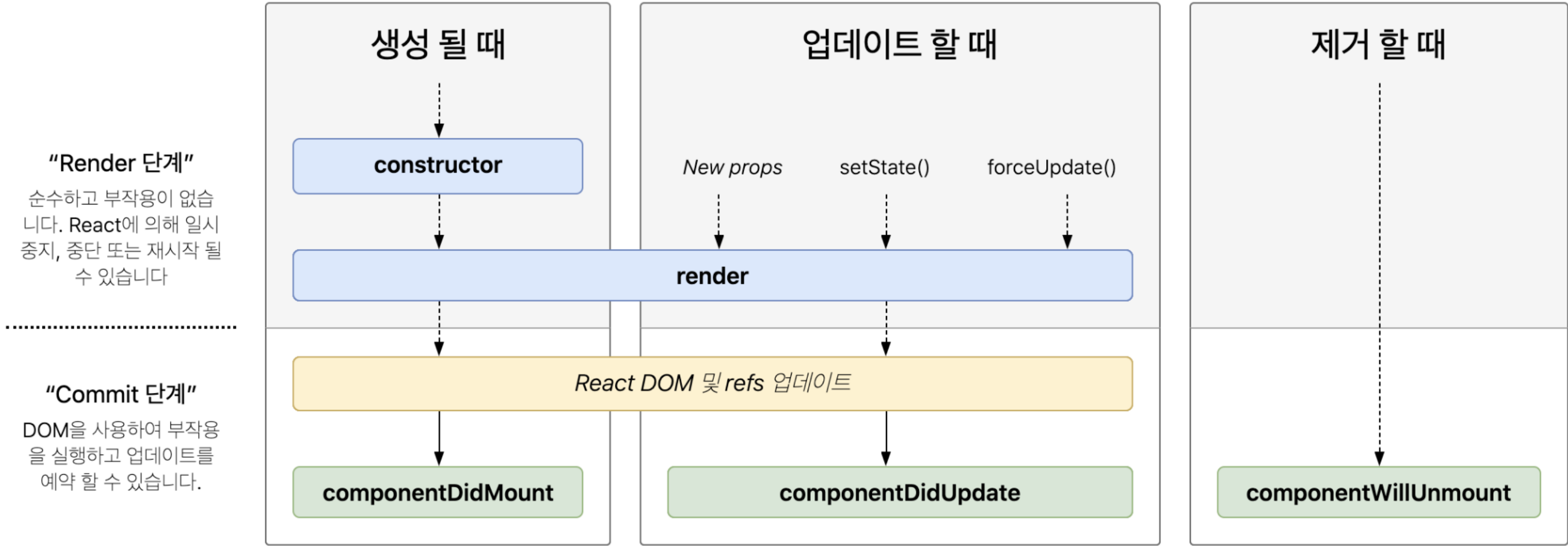
마운트와 언마운트

재렌더(업데이트)

React.js의 생명주기



함수형 컴포넌트의 생명주기



useEffect

React에서 제공하며, Side Effect를 다루어 Lifecycle과 비슷한 조작을 가능하게 해주는 특별한 함수

- **컴포넌트에서 값(상태)의 변화를 감지함**
컴포넌트에서 관리하는 값(props, state)의 변화를 mount 이후부터 지속적으로 감지하여 값이 변화했을 경우 콜백 함수를 통해 변화와 관련된 동작을 조작할 수 있음(이벤트 리스너와 비슷)
- **클래스에서는 Lifecycle을 이용해 조작**
클래스에서는 생명주기 메서드를 상속받아 사용하기 때문에 코드가 난잡해지지만, useEffect는 이를 간단하게 구현함
감지할 값이 없으면 mount와 unmount만 감지
- **clean-up 함수**
useEffect의 인자로 넘어가는 콜백함수에서 return 값으로 함수를 넘겨주는 함수로, 이 함수는 값의 변화가 발생하기 직전, 그리고 unmount가 발생하기 직전에 실행됨

마운트와 언마운트

컴포넌트의 시작과 끝에 한 번 씩 발생

- 마운트 단계에서는 state, defaultProps, context 등을 저장하고 설정하는 작업
- 초기에 렌더링을 한 후에 DOM에 접근 가능 하므로, 마운트 단계가 끝나기 전엔 DOM에 접근할 수 없음
- useEffect에 **deps 배열**에 빈 배열을 넣은 후 실행될 때가 마운트 단계가 끝났을 때임

```
const LifecycleCheck = () => {
  const onClick = () => {
    console.log("onCick Event");
  };
  useEffect(() => {
    console.log('mount');
  }, []);

  return <div onClick={onClick}>Life Cycle Check</div>;
};
```

- 언마운트 단계에서는 주로 DOM에 직접 등록했던 이벤트를 제거하고 외부 라이브러리를 dispose 해주거나, setTimeout, setInterval 등을 clear 함
- useEffect에 **deps 배열**에 빈 배열을 넣은 후 cleanup 함수가 실행될 때 언마운트 단계임

```
const LifecycleCheck = () => {
  const onClick = () => {
    console.log("onCick Event");
  };
  useEffect(() => {
    console.log('mount');
    return () => {
      console.log('unmount');
    }
  }, []);

  return <div onClick={onClick}>Life Cycle Check</div>;
};
```

재렌더(업데이트)

컴포넌트의 props, state가 변경될 때마다 연산

- 상위 컴포넌트에서 전달한 props의 값이 바뀌거나, 현재 컴포넌트의 state가 변경되었을 때, 함수형 컴포넌트는 **함수 그 자체(함수 자체가 render 메서드)가 실행됨**
- 실행되면서 함수 내에 정의된 함수들(onClick 리스너 등) 혹은 변수들은 새로 선언됨
- 때문에 함수 내에서 **변수는 거의 무의미하게 되기 때문에 useState 등의 훅 메서드**를 사용함
- `const onClick = () => ...`과 같이 함수를 선언했다면 렌더가 실행될 때마다 새로운 onClick을 선언함

```
let prevFunc = null;
const LifecycleCheck = () => {
  const [count, setCount] = useState(0);
  const onClick = () => {
    console.log("onClick Event ", count);
    setCount(count + 1);
  };
  console.log("이전 onClick와 현재 onClick: ", prevFunc === onClick);
  prevFunc = onClick;
  useEffect(() => {
    console.log('count update');
  }, [count]);

  return <div onClick={onClick}>Life Cycle {count}</div>;
}
```

상태가 변경되어 렌더 연산을 할 때마다,
좌측의 함수 코드를 꼭 실행한다고 생각하면 됨

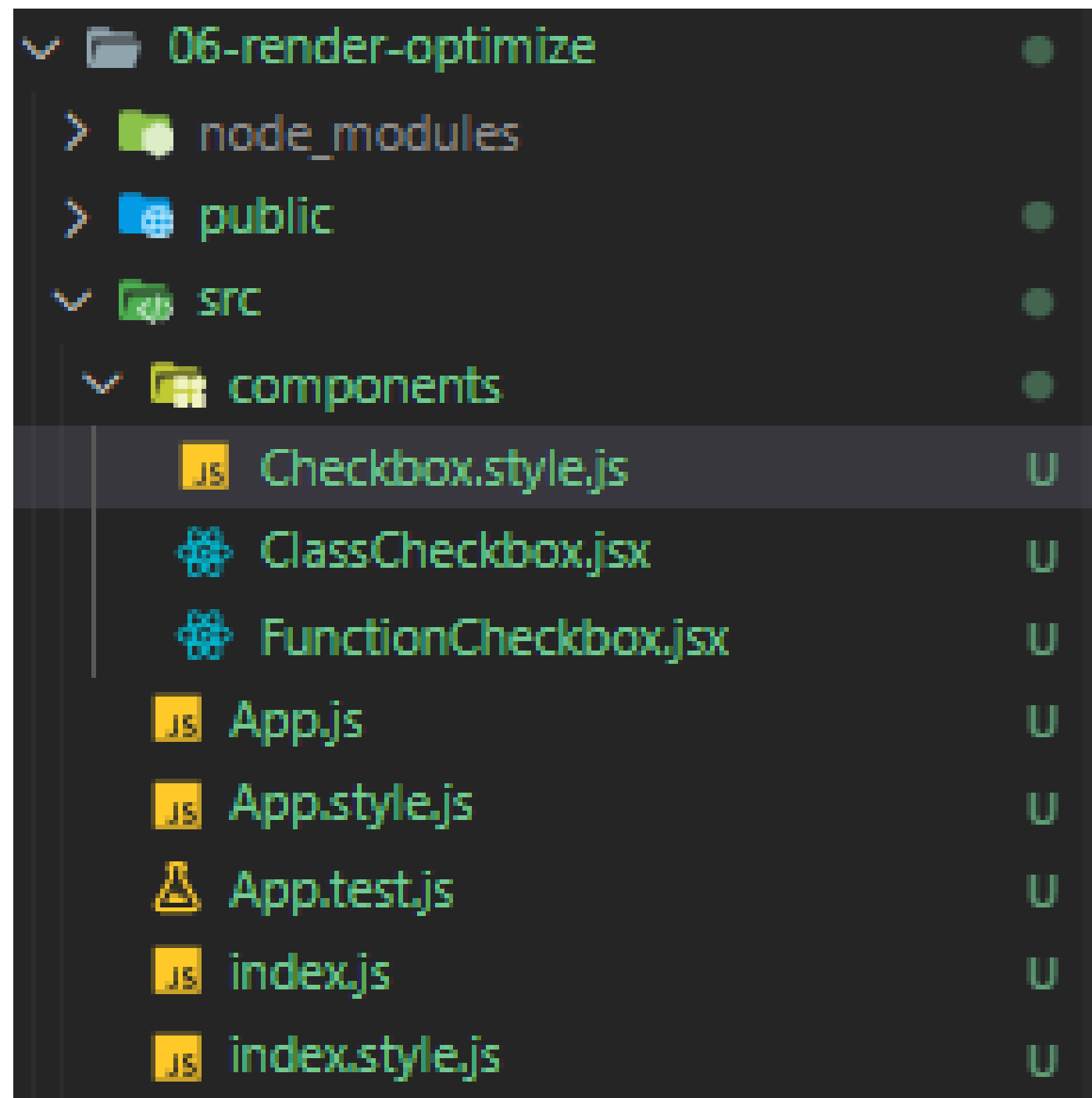
02

커스텀 컴포넌트 작성

함수형 컴포넌트
클래스형 컴포넌트

함수형 컴포넌트

다음과 같이 파일 생성 후 Checkbox.style.js에 다음 코드 작성



```
import styled from "@emotion/styled/macro";

export const StyleCheckboxWrapper = styled.div`
  padding: 0 12px;
  box-shadow: 0px 3px 15px #0001;
`;

export const StyleCheckboxContent = styled.div`
  display: flex;
  align-items: center;
  height: 40px;
`;

export const StyleCheckboxMore = styled.div`
  display: flex;
  align-items: center;
  width: 100%;
  height: 30px;
  color: #777;
  font-size: 12px;
`;

export const StyleCheckboxDetail = styled.div`
  width: 100%;
  padding: 24px 0;
`;
```

함수형 컴포넌트

그 후, FunctionCheckbox.jsx에 다음 코드 작성

```
import { Checkbox } from "@mui/material";
import { StyleCheckboxContent, StyleCheckboxDetail, StyleCheckboxMore, StyleCheckboxWrapper } from "./Checkbox.style";

const FunctionCheckbox = () => {
  return (
    <StyleCheckboxWrapper>
      <StyleCheckboxContent>
        <Checkbox checked={false} />
      </StyleCheckboxContent>
      {true ? (
        <StyleCheckboxMore>사람을 화나게하는 방법은 두가지가 있다고 합니다. 그 첫번째는 말을 하다가 마는 것이고... 더보기</StyleCheckboxMore>
      ) : (
        <StyleCheckboxDetail>대충 긴 내용?</StyleCheckboxDetail>
      )}
    </StyleCheckboxWrapper>
  );
};

export default FunctionCheckbox;
```

클래스형 컴포넌트 ClassCheckbox.jsx에는 다음 코드 작성

```
import { Component } from "react";
import { Checkbox } from "@mui/material";
import { StyleCheckboxContent, StyleCheckboxDetail, StyleCheckboxMore, StyleCheckboxWrapper } from "./Checkbox.style";

class ClassCheckbox extends Component {
  render() {
    return (
      <StyleCheckboxWrapper>
        <StyleCheckboxContent>
          <Checkbox checked={false} />
        </StyleCheckboxContent>
        {true ? (
          <StyleCheckboxMore>사람을 화나게하는 방법은 두가지가 있다고 합니다. 그 첫번째는 말을 하다가 마는 것이고... 더보기</StyleCheckboxMore>
        ) : (
          <StyleCheckboxDetail>대충 긴 내용?</StyleCheckboxDetail>
        )}
      </StyleCheckboxWrapper>
    );
  }
}

export default ClassCheckbox;
```

03

useState와 조건부 렌더링

useState
setState를 통한 조건부 렌더링

useState

React에서 제공하며, 클래스형 컴포넌트 없이 상태를 사용할 수 있게 해주는 특별한 함수

- 컴포넌트의 상태를 관리함

현재 컴포넌트의 상태를 나타내며, 이를 통해 컴포넌트는 알맞는 모습을 나타냄

- 렌더링에 영향을 줌

이 State가 변하면 React는 변화를 인지하고 Virtual DOM에서 재 렌더 여부를 결정함

- 훅 메서드

함수형 컴포넌트에서, 함수 내에 함수를 선언한다고 Virtual DOM의 재 렌더 여부를 결정하는 곳까지 영향을 줄 수 없으나, 훅 메서드를 통해 React 내부와 연결

useState를 통한 조건부 렌더링

FunctionCheckbox.jsx에 다음 코드 작성

```
import { useState } from "react";
// 중략 ...

const FunctionCheckbox = () => {
  const [hideMore, setHideMore] = useState(true);
  const onViewMore = () => {
    console.log("test");
    setHideMore(false);
  };
  const onHideMore = () => {
    setHideMore(true);
  };
  return (
    <StyleCheckboxWrapper>
    // 중략 ...
    {hideMore ? (
      <StyleCheckboxMore onClick={onViewMore}>사람을 화나게하는 방법은 두가지가 있다고 합니다. 그 첫번째는 말을 하다가 마는 것이고... 더보기</StyleCheckboxMore>
    ) : (
      <StyleCheckboxDetail onClick={onHideMore}>대충 긴 내용?</StyleCheckboxDetail>
    )}
    </StyleCheckboxWrapper>
  );
};
// 중략 ...
```

04

반복 렌더링

JavaScript Array

Array.prototype.map

map 메서드를 이용한 반복 렌더링

JavaScript Array

Array 객체는 배열을 생성할 때 사용하는
전역 객체

- 리스트 형태의 자료구조로 순회 가능

다른 언어와는 다르게 내용물의 타입이 고정되어 있지는 않음.
길이가 동적으로 변하며, 인덱스 참조 및 순회가 가능

- 간단한 생성 방식

```
const a = [1, 2, 3];  
const b = ["test", "none", "yes"];
```

와 같이 간단하게 생성할 수 있는 객체

- 객체이므로 내장 함수(메서드)를 가짐

Array 객체는 다양한 내장 함수를 가지고 있으며, 순회,
삽입, 삭제, 복사, 포함 유무 등의 기능들을 제공



Array.prototype.map

배열의 모든 요소를 순회하는 함수를 호출하고, 그 결과 값을 모아 새로운 배열을 반환함

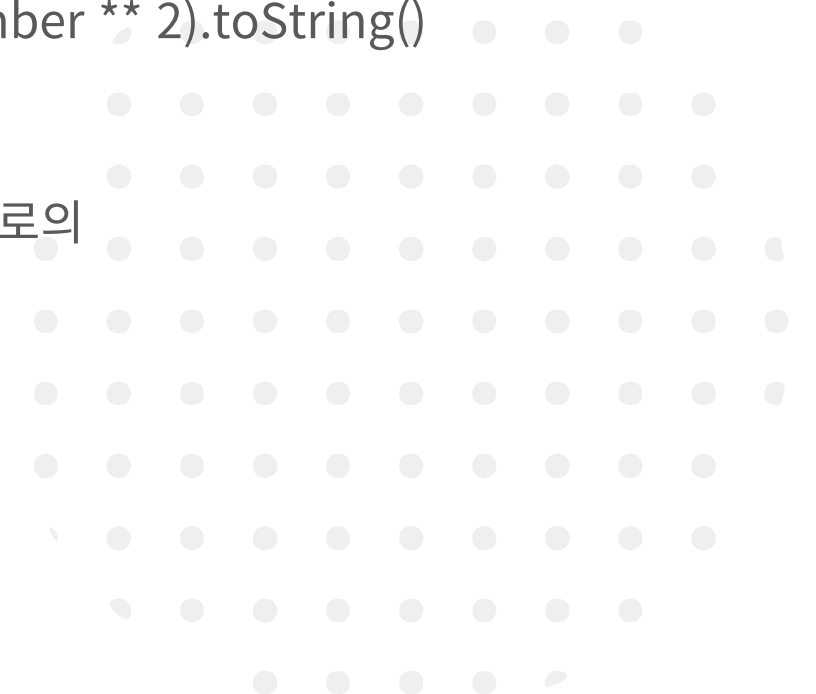
- 배열을 통해 새로운 값을 가지는 배열을 만드는 Array의 내장 메서드
- 새로운 배열을 만들어 내기 때문에 기존의 배열에는 변화를 주지 않음
- 콜백 함수를 매개변수로 넘기며, 콜백 함수의 인자로 현재 값, 현재 값이 있는 인덱스, 전체 배열이 넘어옴
- 콜백 함수에서 return한 값이 해당 인덱스의 값으로 반환됨

```
const numberArray = [1, 2, 3, 4, 5];
```

```
const newArray = numberArray.map((number, index, arr) => {  
    return number ** 2;  
}); // [1, 4, 9, 16, 25] 반환.
```

```
const newArray2 = numberArray.map((number, index, arr) => {  
    return index.toString() + ": " + (number ** 2).toString()  
});
```

이와 같이 number 배열 -> string 배열 등으로의 배열 타입 변환하여 반환해도 문제 없음.



map 함수를 이용한 반복 렌더링 FunctionCheckbox.jsx에 다음 코드 작성

```
const FunctionCheckbox = ({ isHide, checked }) => {
  // ...

  export const FunctionCheckboxList = () => {
    const [checkboxList, setCheckboxList] = useState([
      {
        id: 1,
        isHide: true,
        checked: false,
      },
      {
        id: 2,
        isHide: true,
        checked: false,
      },
    ]);
    return (
      <>
        {checkboxList.map((checkboxItem) => (
          <FunctionCheckbox key={checkboxItem.id} isHide={checkboxItem.isHide} checked={checkboxItem.checked} />
        ))}
      </>
    );
  };
};
```

- 컴포넌트를 선언한 함수의 인수로 부터 컴포넌트에 전달한 값을 가져올 수 있음
- FunctionCheckbox 컴포넌트에서는 isHide와 checked 인수를 사용함
- FunctionCheckboxList 컴포넌트는 FunctionCheckbox 컴포넌트에게 넘겨줄 데이터의 리스트를 갖고 있음
- 데이터의 리스트를 map 함수를 통해 컴포넌트 배열로 반환하고, 컴포넌트 배열은 즉시 렌더링됨
- map 함수로 렌더하게 되면 렌더되는 컴포넌트에는 key라는 매개변수를 추가적으로 전달해야 함
- key 매개변수는 리스트 렌더링에 효율적인 사용을 위해 **다른 리스트와 겹치지 않는 고유의 값**을 전달

05

상태 최적화 관리

useMemo

useCallback

memo

FunctionCheckboxList 수정

Checkbox를 변경할 수 있는 함수를 FunctionCheckboxList 컴포넌트에 선언

```
const FunctionCheckboxList = () => {  
  // 중략 ...  
  const onChangeCheckbox = (id) => {  
    const newCheckboxList = checkboxList.map((checkboxItem) =>  
      checkboxItem.id === id  
        ? { ...checkboxItem, checked: !checkboxItem.checked }  
        : checkboxItem  
    );  
    setCheckboxList(newCheckboxList);  
  };  
  const onClickChangeView = (id) => {  
    const newCheckboxList = checkboxList.map((checkboxItem) =>  
      checkboxItem.id === id  
        ? { ...checkboxItem, isHide: !checkboxItem.isHide }  
        : checkboxItem  
    );  
    setCheckboxList(newCheckboxList);  
  };  
  // 중략 ...  
};
```

- React는 단방향 전달이므로 하위에서 상위로 값을 전달할 수 없으므로,
하위에서 상위의 값을 변경하려면 상위에서 변경하는 함수를 작성 후, 변경하는 함수를 넘겨주고 하위에서 실행만 해야 함

- 전체 checkboxList를 map 메서드로 순회하여
인수로 넘어온 id와 같은 checkboxItem이면 그 id의 checked, isHide의 값을 반대로 변경한 **새로운 배열**을 반환함

FunctionCheckbox 수정

상위 컴포넌트에서 선언한 상태 변경 함수를 전달 받은 후 HTML 요소에 바인딩

```
const FunctionCheckbox = const FunctionCheckbox = ({
  isHide,
  checked,
  onChangeCheckbox,
  onClickChangeView,
}) => {
  return (
    <StyleCheckboxWrapper>
      <StyleCheckboxContent>
        <Checkbox onChange={onChangeCheckbox} checked={checked} />
      </StyleCheckboxContent>
      <div onClick={onClickChangeView}>
        {isHide ? (
          <StyleCheckboxMore>
            사람을 화나게하는 방법은 두가지가 있다고 합니다. 그 첫번째는 말을
            하다가 마는 것이고... 더보기
          </StyleCheckboxMore>
        ) : (
          <StyleCheckboxDetail>대충 긴 내용?</StyleCheckboxDetail>
        )}
      </div>
    </StyleCheckboxWrapper>
  );
};
```

- FunctionCheckboxList에서 props로 넘어온 함수를 FunctionCheckbox에서 HTML에 바인딩하면, HTML 요소를 클릭했을 때 위에 선언된 함수를 실행

- 단방향 전달 방식으로 하위 컴포넌트에서 상위 값을 설정할 수 있음

- 몇 번 hide 됐는지 체크하는 경우를 추가하는 경우, 상위에서 데이터에 count 프로퍼티를 추가하고, count 프로퍼티를 연산하는 로직을 작성 후 props로 전달

전체 체크 유무를 확인하려면?

상위 컴포넌트에서 선언한 상태 변경 함수를 전달 받은 후 HTML 요소에 바인딩

```
const FunctionCheckbox = ({
  // 중략 ...
  label,
}) => {
  return (
    // 중략 ...
    <StyleCheckboxContent>
      // 중략 ...
      <p>{label}</p>
    </StyleCheckboxContent>
    // 중략 ...
  );
};
```

```
const FunctionCheckboxList = () => {
  // 중략 ...
  const AllCheck = checkboxList.every((checkboxItem) => checkboxItem.checked);

  return (
    <>
      <FunctionCheckbox
        checked={AllCheck}
        label="전체 체크 여부"
      />
      // 중략 ...
    </>
  );
};
```

- Array.prototype.every 메서드를 이용하면, 배열 내의 모든 값들을 순회하며 각 요소들이 모든 조건에 부합하는지 확인

```
checkboxList.every((checkboxItem) => checkboxItem.checked);
```

위 코드는 아래 코드와 같음

```
checkboxList.every((checkboxItem) => {
  return checkboxItem.checked;
});
```

- 요소의 조건이 모두 true이면 true를 반환하고, 하나라도 false가 있으면 false를 반환

- AllCheck는 리렌더가 호출될 때마다 계속 연산되는 문제 발생

useMemo

React에서 제공하며, Side Effect를 다루어 변화에만 반응하는 변수 생성을 가능하게 해주는 특별한 함수

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

const FunctionCheckboxList = () => {
  // 중략 ...
  const AllCheck = useMemo(() => {
    return checkboxList.every((checkboxItem) => checkboxItem.checked);
  }, [checkboxList]);

  return (
    <>
      <FunctionCheckbox
        checked={AllCheck}
        label="전체 체크 여부"
      />
      // 중략 ...
    </>
  );
};
```

- useMemo에는 변화를 감지하면 실행하여 **값을 반환할 콜백 함수와, 감지할 값들의 배열**을 전달
- 감지할 값들이 변화될 때만 콜백 함수가 실행되어 값이 변경됨
- 상태를 통해 **추가적으로 값을 연산해야 하는 경우** 사용하기 좋음
- 다른 변화로 인한 렌더링 과정에서, 불필요한 연산을 제거할 수 있어 성능 향상에 도움

전체 선택 기능 구현

전체 선택 Checkbox에 바인딩할 함수 작성

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

const FunctionCheckboxList = () => {
  // 중략 ...
  const onChangeAllCheckbox = () => {
    const newCheckboxList = checkboxList.map((checkboxItem) => ({
      ...checkboxItem,
      checked: !AllCheck,
    }));
    setCheckboxList(newCheckboxList);
  };

  return (
    <>
      <FunctionCheckbox
        checked={AllCheck}
        onChangeCheckbox={onChangeAllCheckbox}
        label="전체 체크 여부"
      />
      // 중략 ...
    </>
  );
};
```

- AllCheck의 반대 값으로 모든 Checkbox 데이터의 checked를 설정함
- 전체 선택이 되어 있으면 취소하고, 안 되어 있으면 전체 선택하는 로직

useCallback

React에서 제공하며, Side Effect를 다루어 변화에만 반응하는
함수 생성을 가능하게 해주는 특별한 함수

```
import { useState, useCallback, useMemo } from "react";
```

```
// 중략 ...
```

```
const onClickChangeView = useCallback((id) => {  
  const newCheckboxList = checkboxList.map((checkboxItem) =>  
    checkboxItem.id === id  
      ? { ...checkboxItem, isHide: !checkboxItem.isHide }  
      : checkboxItem  
  );  
  setCheckboxList(newCheckboxList);  
}, [checkboxList]);
```

```
// 중략 ...
```

```
const onChangeAllCheckbox = useCallback(() => {  
  const newCheckboxList = checkboxList.map((checkboxItem) => ({  
    ...checkboxItem,  
    checked: !AllCheck,  
  }));  
  setCheckboxList(newCheckboxList);  
}, [AllCheck, checkboxList]);
```

- useCallback에는 변화를 감지하면 **생성할 함수와, 감지할 값들의 배열**을 전달
- 감지할 값들이 **변화될 때만 함수가 생성**됨
- 상태에만 영향을 받지 않는 함수는 빈 배열 전달
- 다른 변화로 인한 렌더링 과정에서, 불필요한 연산을 제거할 수 있어 성능 향상에 도움

memo

React에서 제공하며, props이외의 변화가 발생해도 재렌더 하지 않는 컴포넌트 생성

```
import { useState, useCallback, useMemo, memo } from "react";
```

```
// 중략 ...
```

```
const FunctionCheckbox = memo(  
  ({ isHide, checked, label, onChangeCheckbox, onClickChangeView }) => {
```

```
// 중략 ...
```

```
    return (  
      // 중략 ...
```

```
    );
```

```
  });
```

- memo 함수는 컴포넌트를 인자로 받아, 해당 컴포넌트에 전달되는 **props 이외의 상태가 변화하더라도 재렌더 하지 않는 컴포넌트를 반환**하는 함수
- 다른 변화로 인한 불필요한 렌더링을 제거할 수 있어 성능 향상에 도움

실증 아카데미 리액트 특강

THANK
YOU

(주) 인바이즈 박철현
(주) 인바이즈 개발팀