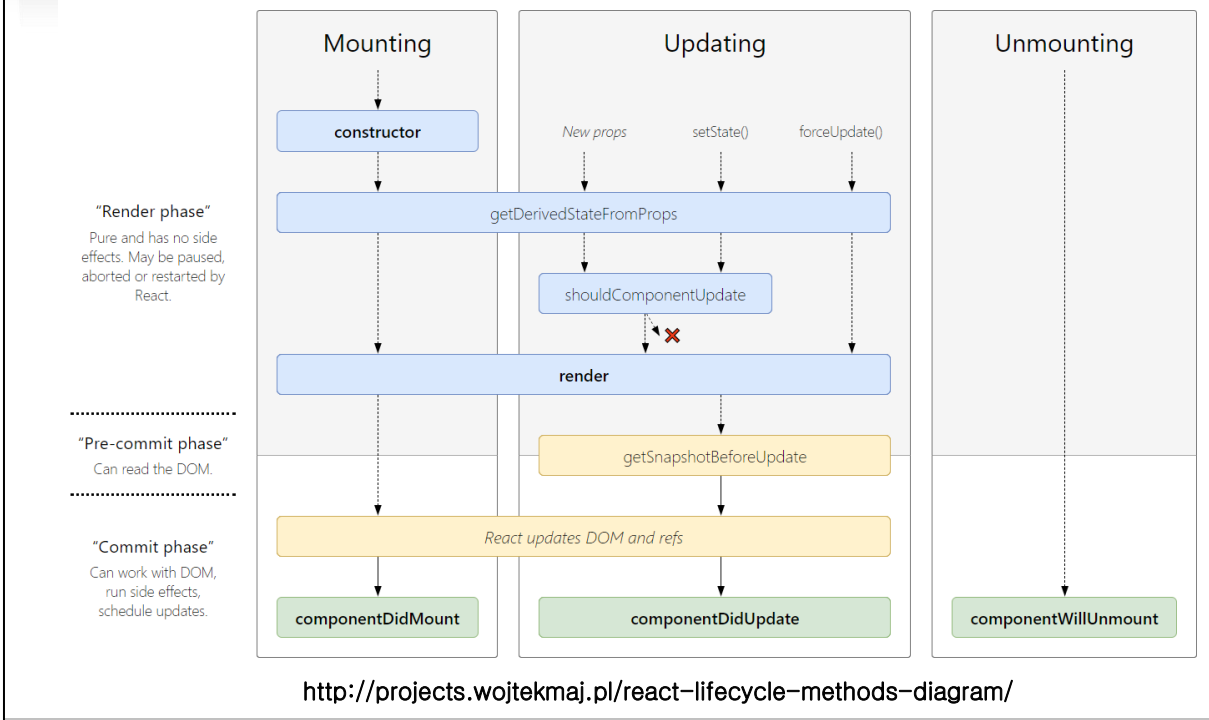
1. 컴포넌트 생명주기 메서드(1)



http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

1

■ 이전 버전에서 다음 3개의 Event가 deprecated 상태가 되었다.

- componentWillMount
- componentWillUpdate
- componentWillReceiveProps

■ 새롭게 다음 Event가 추가되었다.

- static getDerivedStateFromProps
  - 부모 컴포넌트로부터 전달받은 속성을 이용해 기존 컴포넌트의 상태를 변경해 리턴하면 새로운 컴포넌트의 상태가 된다.
- getSnapshotBeforeUpdate
  - 가상DOM의 내용을 Browser DOM에 업데이트하기 전의 이벤트이다.

1. 컴포넌트 생명주기 메서드(2)

컴포넌트 마운트시의 흐름

- 컴포넌트의 인스턴스가 만들어지고 DOM에 추가될 때 호출됨
- constructor(props) : 생성자
  - 마운트 되기 전에 호출되며 상태를 초기화하기 위한 최적의 시점
  - 인자는 속성이며 constructor 내부의 첫 줄에 반드시 super(props)가 포함되어야 함.
  - 상태가 없다면 생성자를 구현할 필요가 없음
  - 부모 컴포넌트로부터 속성을 전달받아 상태를 초기화할 수 있으나 부모 컴포넌트로부터 전달되는 속성이 변경되면 이를 이용해 상태를 매번 변경해줘야 하는 불편함이 있으므로 권장하지 않음. --> 속성이 변경되더라도 초기 상태를 유지해야 하는 경우라면 사용가능
- render()
  - Component의 render() 메서드
  - 일반적으로 JSX를 사용하여 작성함.
  - return 된 객체들이 Virtual DOM에 렌더링됨.
  - 이 메서드 내부에서 setState() 해서는 안됨 --> 무한 루프

1. 컴포넌트 생명주기 메서드(3)

- componentDidMount()
  - 컴포넌트의 마운트가 완료된 후에 호출.
  - DOM의 구조를 확인하고 난 뒤 DOM에 대한 초기화를 수행할 때
  - 원격 서버로부터 데이터를 로드하여 초기화하거나 이벤트 구독(subscription)을 설정하기 적절한 시점
    - 서버와 소켓을 연결하거나 이벤트 구독을 수행했다면 componentWillUnmount 단계에서 반드시 해제해야 함.
  - 이 단계에서 setState()가 호출되면 re-render가 일어나지만 실제 브라우저의 HTML DOM에 반영되기 전에 호출되는 것임
    - 따라서 render() 가 두번 호출되지만 사용자는 최종 상태만 조회하게 됨.

속성, 상태 변경 시의 흐름

- static getDerivedStateFromProps()
  - props가 변경되어 컴포넌트에 전달되면 이 값을 이용해 state를 동기화할 때 사용함.
  - 이 메서드에서 객체를 리턴하면 이 값이 새로운 상태가 됨.
  - 이 메서드에서 null을 리턴하면 상태를 업데이트하지 않겠다는 의미.
  - 자주 사용하는 메서드는 아님

3

■ getDerivedStateFromProps 메서드 예

```
static getDerivedStateFromProps(nextProps, prevState){
   if(nextProps.someValue!==prevState.someValue){
      return { someState: nextProps.someValue};
  }
  else return null;
}
```

1. 컴포넌트 생명주기 메서드(4)

- shouldComponentUpdate(nextProps, nextState)
  - 전달 인자 : 새로운 속성, 새로운 상태
  - 리턴값이 중요함
    - 리턴값이 true : 이후 단계의 메서드를 실행함(componentWillUpdate, render, componentDidUpdate)
    - 리턴값이 false : 이후 단계의 메서드를 실행하지 않음.
  - 이 메서드를 작성하지 않으면 기본적으로 true를 리턴함
    - PureComponent 인 경우 shallowCompare 한후 다른 경우에만 true를 리턴하도록 이미 구현되어 있으므로 이 메서드 작성이 불가능함.
  - 이 단계에서 속성과 상태를 비교할 때 deepCompare는 권장하지 않음
    - 고비용의 작업이므로 오히려 성능을 저하시킴.
    - 따라서 이전 장에서 다룬 불변성 헬퍼를 사용하고 shallowCompare하는 편이 바람직함.
- getSnapshotBeforeUpdate()
  - render()가 호출된 직후 Virtual DOM에는 업데이트가 되고 나면 호출됨
  - 아직은 Browser DOM에 업데이트 되기 전임
  - 자주 사용되는 메서드는 아님.

1. 컴포넌트 생명주기 메서드(5)

- componentDidUpdate(prevProps, prevState)
  - DOM 업데이트가 일어난 직후에 실행됨.
  - 마운트 시에는 실행되지 않음.
  - 실제 HTML DOM이 업데이트된 후이므로 실제 DOM을 이용한 작업을 하기에 적절함
  - 현재 속성과 이전 속성을 비교하여 다른 경우에만 지정된 작업을 실행하도록 할 수 있음
    - 예) 현재의 속성과 이전 속성이 다르면 서버측으로 데이터를 요청하고자 할 때

컴포넌트 언마운트 시의 흐름

- componentWillUnmount
  - 컴포넌트가 언마운트될 때 실행됨(예: 화면이 완전히 전환될 때)
  - 소켓 서버로의 네트워크 연결 해제, 이벤트 구독 해제 등의 작업을 수행할 수 있음

1. 컴포넌트 생명주기 메서드(6)

오류 발생시의 흐름

- componentDidCatch(error, info)
  - 자신과 자식 컴포넌트 트리에서 오류가 발생할 경우 호출됨.
  - 이후 fallback UI 가 지정되어 있다면 fallback UI를 표시할 수 있음
  - 자세한 내용은 다음 내용을 참조
    - https://ko.reactjs.org/docs/error-boundaries.html
    - 에러처리 전용 컴포넌트를 작성하고 에러처리를 하고 싶은 범위의 컴포넌트 트리를 래핑함.
    - 에러 처리 전용 컴포넌트는 자신의 자식 트리상의 컴포넌트에서의 오류만 처리함. 자신의 오류는 처리하지 못함.

2. 생명주기 예제(1)

디지털 시계 앱

- 생명주기 중 componentDidMount, componentWillunmount 를 사용해 오류 없는 디지털 시계를 작성
- 프로젝트 생성
  - npx create-react-app digital-clock-app
  - App.js, App.css, App.test.js 코드 삭제
- 첫번째 단계
  - componentWillUnmount 생명주기에서 리소스를 해제하지 않으면 어떤 문제점이 발생하는지를 살펴봄

2. 생명주기 예제(2)

Clock.js 추가

```
import React, { Component } from 'react';

class Clock extends Component {
    constructor(props) {
        super(props);
        this.state = { currentTime : new Date() }
    }

    componentDidMount() {
        this.handle = window.setInterval(() => {
            console.log("### tick!");
            this.setState({ currentTime : new Date() });
        }, 1000);
    }

    render() {
        return (
            <div style={{ padding:'10px', border:'solid 1px gray' }}>
                <h2>{ this.state.currentTime.toLocaleTimeString() }</h2>
            </div>
        );
    }
}

export default Clock;
```

2. 생명주기 예제(3)

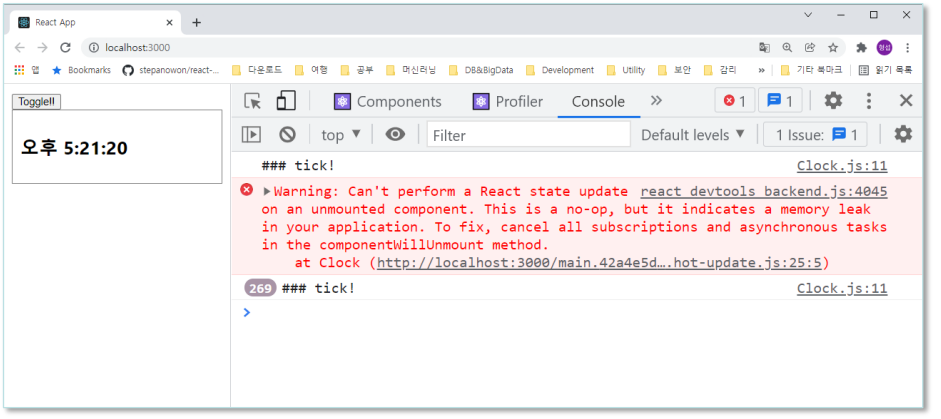
App.js 추가

```
import React, { Component } from 'react';
import Clock from './Clock';

class App extends Component {
    constructor(props) {
        super(props);
        this.state = { isVisible : false };
    }

    toggleVisible = () => {
        this.setState({ isVisible : !this.state.isVisible });
    }

    render() {
        return (
            <div style={{ padding: '10px' }}>
                <button onClick={this.toggleVisible}>Toggle!!</button>
                { this.state.isVisible ? <Clock /> : "" }
            </div>
        );
    }
}

export default App;
```

2. 생명주기 예제(4)

실행하여 Toggle 버튼을 여러번 클릭해봄

- 디지털 시계가 나타났다 사라지기를 반복
  - 그 때마다 setInterval()이 실행됨.
- 브라우저 개발자 도구 콘솔을 열어서 어떤 일이 벌어졌는지를 확인
  - 1초에 한번만 실행되어야 tick 기능이 여러번 실행되고 있음을 알 수 있음.



■ setInterval()로 주기적으로 실행되도록 등록했다면 컴포넌트가 언마운트될 때 등록한 주기적인 작업을 해제해야 함

2. 생명주기 예제(5)

문제해결

- ComponentWillUnmount 생명주기 메서드에서 clear함.

Clock.js 에 생명주기 메서드 추가

```
import React, { Component } from 'react';

class Clock extends Component {
    ……
    componentDidMount() {
      ……
    }

    componentWillUnmount() {
       window.clearInterval(this.handle);
    }
    ……
}

export default Clock;
```

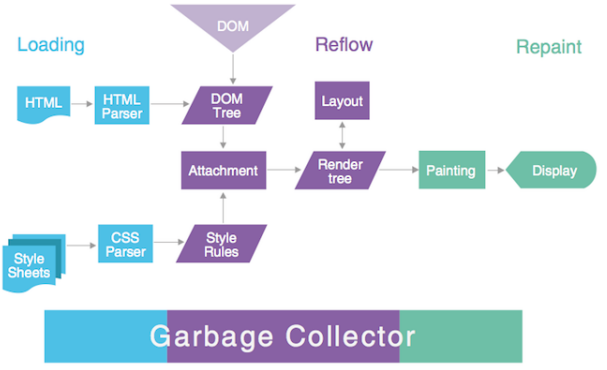
11

■ setInterval() 호출 후에 리턴받은 핸들값을 이용해 clearInterval() 하도록 componentWillUnmount() 생명주기 메서드를 작성함.

3. 가상DOM과 조정 작업(1)

HTML DOM이 느리다?

- DOM 조작은 빠르지만 브라우저에서 reflow, repaint 과정이 느림
  - Reflow : layout이라고도 부름. 렌더링할 DOM Tree를 새로이 만들고 HTML Element 각각의 위치를 계산하고 배치함.
  - Repaint : HTML Element에 스타일을 요소에 입히고 그려냄



https://mobidev.biz/blog/how_to_optimize_the_performance_of_phonegap_apps

12

■ 브라우저에서의 작업 흐름

- HTML Parser가 HTML을 파싱한 후 이것을 기반으로 DOM 트리를 생성함.
- CSS와 inline style을 계산하여 스타일 규칙을 생성함.
- DOM 트리에 스타일 규칙을 적용하여 렌더링 트리를 생성함.
- 이 후에 Reflow와 Repaint가 일어남

■ 성능을 높이려면 Reflow, Repaint를 최소화시키는 것이 중요함.

- 개발자가 이 작업을 직접하려면 쉽지 않음.
- Virtual DOM은 개발자가 Reflow, Repaint 횟수를 신경쓰지 않고 개발할 수 있도록 도와줌.

3. 가상DOM과 조정 작업(2)

React는?

- Always re-render on update!!
- 개발자가 원하는 출력물만을 선언적으로 작성하기 때문에 컴포넌트 전체를 re-render 하듯이 개발할 수 밖에 없음
- 따라서 UI 성능을 위해서 Virtual DOM이 반드시 필요함
- Virtual DOM 트리를 비교하면서 차이가 나는 부분만을 업데이트함.
  - 이것을 조정(Reconciliation) 작업이라 부름
  - 브라우저 DOM의 업데이트 로직은 개발자가 신경쓰지 않아도 됨.

13

3. 가상DOM과 조정 작업(3)

Virtual DOM

- DOM에 대한 추상화된 객체
- 업데이트해야 할 정보를 메모리에 저장함.
- Virtual DOM의 업데이트 비용은 작음
  - 실제 HTML DOM을 업데이트하는 것이 아니므로 Reflow, Repaint가 일어나지 않음
- 이전 스냅샷과 Virtual DOM 변경 후의 스냅샷을 비교해 차이가 발생한 부분에 대해서만 실제 HTML DOM을 업데이트함.

https://github.com/drborges/react-in-a-nutshell

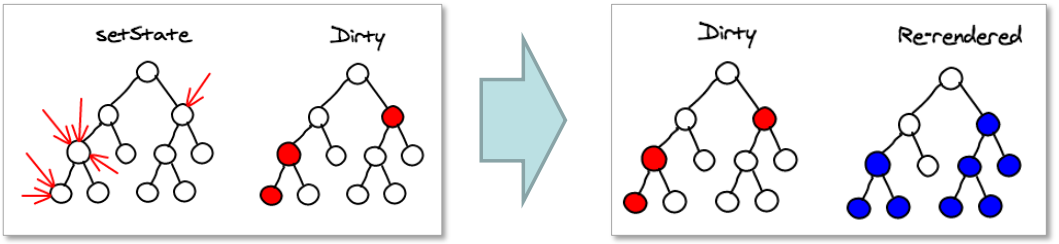
14

■ Virtual DOM을 사용하면 개발자가 어떤 HTML DOM 요소를 수정해야 할지를 신경쓸 필요가 없다. Virtual DOM의 Diff 알고리즘을 이용해 비교하고 차이가 나는 부분만이 업데이트되기 때문이다.

3. 가상DOM과 조정 작업(4)

Virtual DOM의 Diff 알고리즘

- DOM 트리의 노드들을 비교하면서 노드가 다른 유형일 경우 기존 노드를 버리고 새로운 노드로 교체
- 노드가 같은 유형인 경우
  - Attribute와 Style을 비교하여 변경함.
  - 부모 컴포넌트에서 re-render가 실행되면 자식 컴포넌트로 속성을 전달해 자식 컴포넌트도 re-render를 수행함.
  - 선택적 re-render를 위해 shouldComponentUpdate() 생명주기 메서드를 이용할 수 있음.



http://calendar.perfplanet.com/2013/diff/

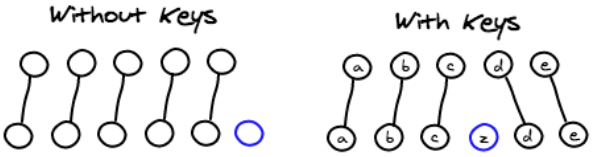
15

■ 부모 컴포넌트에서 render가 호출되면 하위의 자식 컴포넌트들에서도 render()가 호출된다. 자식 컴포넌트에서 render를 선택적으로 수행하려면 React 생명주기 메서드에서 shouldComponentUpdate() 메서드를 재정의하여 작성하면 된다. 이 메서드에서 false가 리턴될 경우 render가 호출되지 않는다.

3. 가상DOM과 조정 작업(5)

key 특성

- 컴포넌트 내부에서 반복적으로 자식 컴포넌트, 요소를 렌더링할 때 사용
- 반복적인 리스트의 변경 사항을 추적하기 힘듬
  - 새로운 요소가 추가, 삽입되는 경우
  - 요소들의 순서가 변경되는 경우
  - 특정 요소가 삭제되는 경우
- 반복적으로 렌더링할 때 key를 부여하지 않으면 React는 경고를 일으킴.
- key 특성의 값으로는 고유한 값을 부여해야 함.(index번호(X))

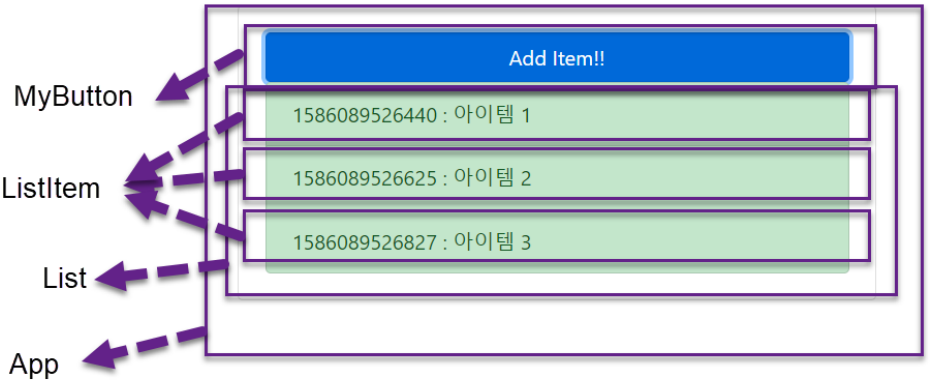


https://calendar.perfplanet.com/2013/diff/

3. 가상DOM과 조정 작업(6)

조정 과정 확인을 위한 예제

- 프로젝트 초기화
  - 4장에서 작성한 parent-child-communication 프로젝트를 이용해서 작성
- 전체 컴포넌트 구조 리뷰



17

3. 가상DOM과 조정 작업(7)

- 콘솔 로그에 출력하는 코드 추가
  - ListItem.js, List.js, MyButton.js 컴포넌트에 추가
  - 단순히 render() 메서드가 실행되고 있음을 확인하는 코드!!

```
class ListItem extends Component {
    render() {
        console.log("### ListItem 컴포넌트 렌더")
        return ( ..... )
    }
}
```

```
class List extends Component {
    render() {
        console.log("### List 컴포넌트 렌더")
        .....
    }
}
```

```
class MyButton extends Component {
    render() {
        console.log("### MyButton 컴포넌트 렌더")
        .....
    }
}
```

18

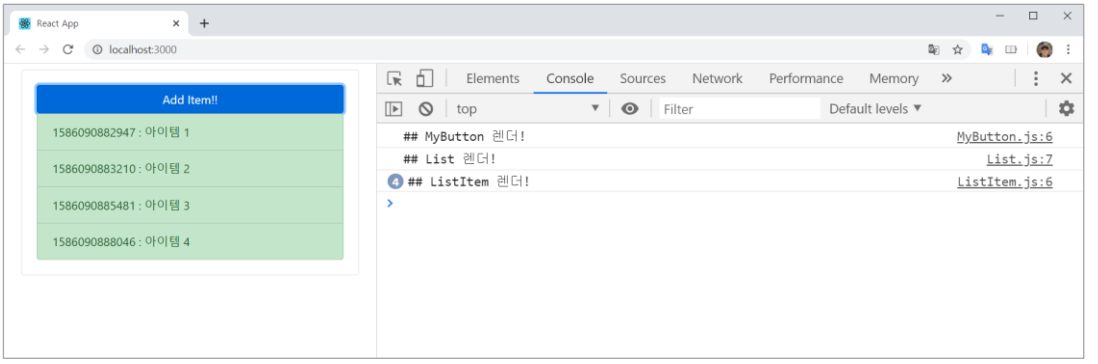
■ index.js 파일에서 React.StrictMode 를 제거한다.

- StrictMode는 애플리케이션 내부의 문제점을 알아내기 위해 사용하는 도구임.
- NODE_ENV가 개발모드(development)일 때 후손 컴포넌트에 대한 부가적인 검사와 경로를 활성화함
- 하지만 정확한 렌더링 실행 횟수를 파악하기 위해서 제거해야 함.
  - 그대로 두고 실행하면 동일한 컴포넌트를 두번 렌더링을 시도함.

```
ReactDOM.render(<App />, document.getElementById('root'));
```

3. 가상DOM과 조정 작업(8)

- 실행 결과 확인
  - 새로운 아이템이 추가되면서 setState() 가 호출되고 App.js에서 render()가 호출되면?
    - 자식 컴포넌트들이 모두 render()!!
    - 하나의 아이템을 추가하지만 ListItem 컴포넌트 모두에서 render()가 호출됨.
    - 기존에 마운트된 ListItem 컴포넌트는 Update될 사항이 없다면 re-render할 이유가 없음.
  - 심지어 MyButton 컴포넌트는 전혀 업데이트될 필요가 없지만 re-render를 호출함.



19

3. 가상DOM과 조정 작업(9)

- shouldComponentUpdate 메서드로 최적화
  - ListItem.js 최적화

```
class ListItem extends Component {
    shouldComponentUpdate(nextProps, nextState) {
        return this.props.item !== nextProps.item;
    }
    render() { ..... }
}
```

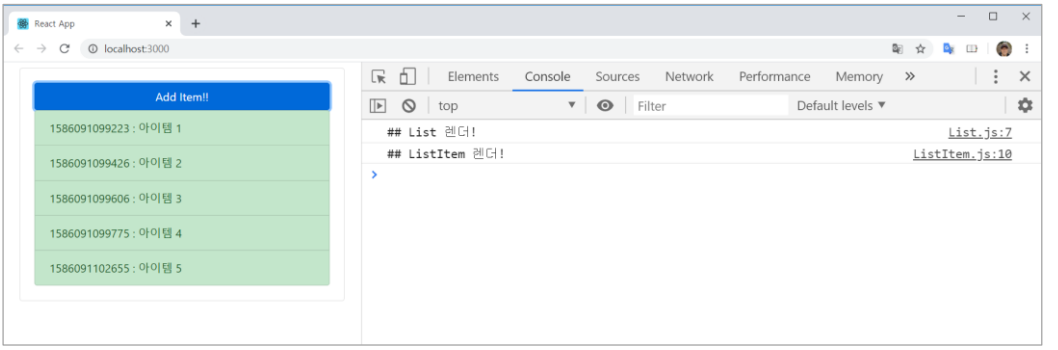
  - MyButton.js 최적화
    - 이 컴포넌트는 속성을 이용해 렌더링하지 않으므로 부모 컴포넌트에서 전달된 속성이 변경되더라도 re-render할 필요가 없음

```
class MyButton extends Component {
    shouldComponentUpdate(nextProps, nextState) {
        return false;
    }
    render() {
        ......
    }
}
```


20

3. 가상DOM과 조정 작업(10)

- 다시 실행하여 브라우저 화면에서 아이템 추가
  - 가장 마지막에 추가된 ListItem 컴포넌트만 렌더링!!
  - MyButton.js는 마운트 될때만 렌더링!!



21

4. PureComponent(1)

React.PureComponent

- React.Component
  - shouldComponentUpdate() 메서드가 구현되어 있지 않기 때문에 setState()가 호출되면 무조건 render()를 호출함.
  - Rendering 과정을 최적화하기 위해
    - 개발자가 직접 shouldComponentUpdate() 메서드를 작성하여 비교하도록 작성해야 함.
- React.PureComponent
  - shouldComponentUpdate()가 shallowCompare 하도록 이미 구현되어 있음
    - 객체의 메모리 주소가 같을 경우 render()를 호출하지 않음.
    - 참조 타입이 아닌 값 타입인 경우는 값이 같으면 render()를 호출하지 않음
    - 명시적으로 shouldComponentUpdate를 작성할 수 없음
  - 표현 컴포넌트에서 사용하기에 적합함.
    - 새롭게 전달받은 props와 현재의 props를 shallowCompare 하여 일치한다면 render()를 수행하지 않으므로 최적화하기가 용이함.
    - 불변성 헬퍼(예:immer)를 사용하는 경우에 더욱 효과적임.

4. PureComponent(2)

PureComponent 적용

- 이전 프로젝트 코드 변경
  - ListItem.js 컴포넌트를 PureComponent로 변경

```
import React, { PureComponent } from 'react';
import PropTypes from 'prop-types';

class ListItem extends PureComponent {

  //shouldComponentUpdate 메서드는 삭제

   render() {
      console.log("### ListItem 컴포넌트 렌더")
      return (
         <li className="list-group-item list-group-item-success">
            {this.props.no} : {this.props.item}
         </li>
      )
   }
}
……
```

  - 실행 결과는 이전과 동일함.

23

4. PureComponent(3)

- MyButton 컴포넌트를 PureComponent로 변경하면?

```
import React, { PureComponent } from 'react';
import PropTypes from 'prop-types';

class MyButton extends PureComponent {
  ……
}
……
```

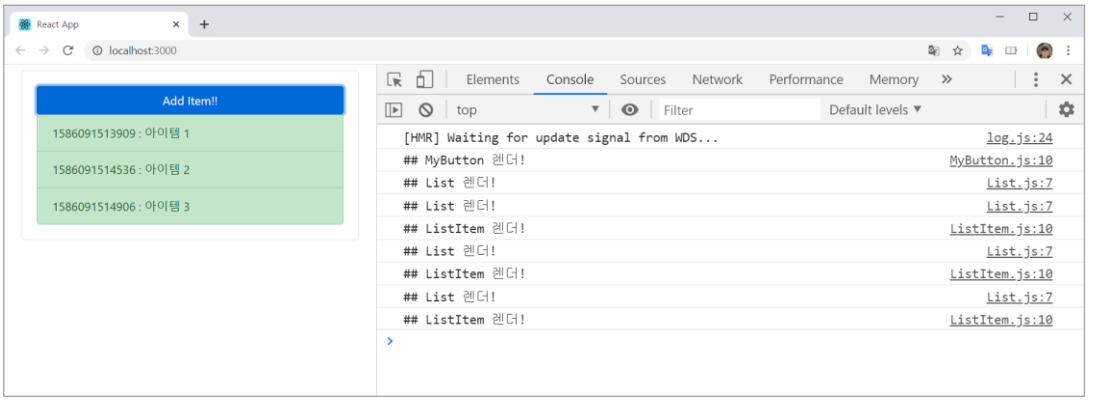
  - 매번 렌더링을 수행함. 원인은 App.js에서 렌더링할 때마다 bind(this)를 매번 호출한 후 속성으로 전달하기 때문임.
  - 이 문제를 해결하려면 다음의 방법을 사용함
    - App.js의 생성자에서 한번만 bind(this)를 수행한 후 속성으로 전달하도록 해야 함. 이를 위해 생성자에서 bind(this)를 미리 한 다음 속성으로 전달함
    - 또는 미리 addItem 메서드를 화살표 함수(Arrow Function)로 작성함.

■ App.js 코드

```
……
class App extends Component {
    constructor(props) {
        ……
        this.addItem = this.addItem.bind(this);
    }
    ……
    render() {
        return (
            ……
            <MyButton addItem={this.addItem} />
            ……
        );
    }
}
……
```

4. PureComponent(4)

- 실행 결과



25

5. 고차 컴포넌트(1)

HOC : Higher Order Component

- 컴포넌트를 입력값으로 받아 새로운 기능을 추가하여 다시 리턴하는 컴포넌트
- 고차 함수 : Higher Order Function
  - 다른 함수를 인자로 받거나 그 결과로 함수를 반환하는 함수다.
- 컴포넌트들 사이의 공통 로직을 분리하고 재사용할 수 있음
  - 사용자 로그인 여부, 권한 상태 확인 기능 추가
  - 에러 발생시 에러 페이지 보여주기
  - 로깅 기능 추가

26

■ 고차 컴포넌트(HOC)로 할 수 있는 일들

- 컴포넌트 내부의 공통 기능을 모듈화하여 코드 재사용
- 렌더링 과정 하이재킹
- 상태의 추상화, 조작
- 속성 조작

5. 고차 컴포넌트(2)

HOC 기능 테스트

- 프로젝트 초기화
  - parent-child-communication 프로젝트에 이어서 작성함.
  - PureComponent를 모두 Component로 변경하고
  - MyButton, List, ListItem 컴포넌트의 render() 메서드 내부에 작성했던 console.log() 코드를 모두 주석처리함.
  - 마이크로초 단위의 시간 측정을 위해 다음 패키지 설치
    - yarn add microseconds
- 로깅 기능을 추가하는 고차 컴포넌트 작성
  - console에 로깅하는 기능 추가
  - 컴포넌트가 마운트 될 때의 render() 시간 측정
    - constructor -> render -> componentDidMount
  - 컴포넌트가 업데이트될 때의 render() 시간 측정
    - shouldComponentUpdate -> render -> componentDidUpdate
  - 어느 컴포넌트인지도 로깅해야 함.
    - componen의 name값을 받아내야 함.
  - 속성을 통해서 로깅할지 여부를 결정할 수 있도록 작성함.

5. 고차 컴포넌트(3)

- src/Logger.js

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import ms from 'microseconds';

let Logger = (TargetComponent) => {
    class Logger extends Component {
        constructor(props) {
            super(props);
            if (this.props.isLog) {
                this.start = ms.now();
            }
        }
        componentDidMount() {
            if (this.props.isLog) {
                let ts = ms.now() - this.start;
                console.log(`### ${TargetComponent.name} 마운트 : ${ts} micro seconds `);
            }
        }
        shouldComponentUpdate(nextProps, nextState) {
            if (nextProps.isLog) {
                this.start = ms.now();
            }
            return true;
        }
```

28

■ microseconds 패키지는 micro seconds 단위의 시간 값까지 획득할 수 있도록 함.

5. 고차 컴포넌트(4)

- src/Logger.js (이어서)

```
        componentDidUpdate(prevProps, prevState) {
            if (this.props.isLog) {
                let ts = ms.now() - this.start;
                console.log(`### ${TargetComponent.name} 업데이트 : ${ts} micro seconds `);
            }
        }
        render() {
            return <TargetComponent  {...this.props} />
        }
    }

    Logger.propTypes = {
        isLog : PropTypes.bool
    };
    Logger.defaultProps = {
        isLog : false
    };
    return Logger;
};

export default Logger;
```

5. 고차 컴포넌트(5)

- src/ListItem.js
  - Logger 고차 컴포넌트를 거쳐서 로깅 기능을 추가하도록 변경

```
……
import Logger from './Logger';

class ListItem extends Component {
  ……
}
……
export default Logger(ListItem);
```

- src/List.js

```
……
import Logger from './Logger';

class List extends Component {
    render() {
        let items = this.props.itemlist.map((item) => {
            return (<ListItem  isLog={process.env.NODE_ENV === "development"} key={item.no} item={item} />)
        });
        ……
    }
}
……
export default Logger(List);
```

■ 개발 모드일 때만 로깅하도록 하기 위해서 isLog={process.env.NODE_ENV === "development"} 와 같이 Boolean 값을 부여하도록 작성하였음.

5. 고차 컴포넌트(6)

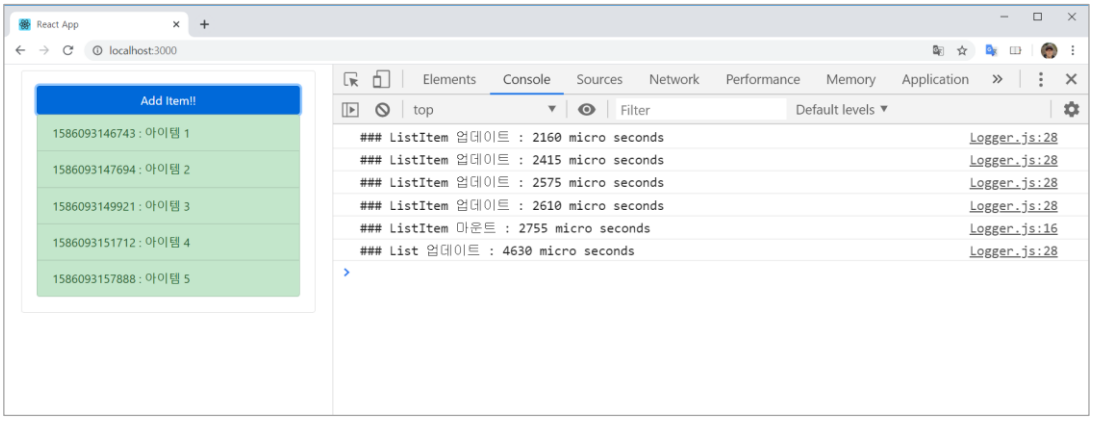
- src/App.js

```
……
class App extends Component {
    ……
    render() {
        return (
            <div className="container">
                <div className="card card-body">
                    <MyButton addItem={this.addItem} />
                    <List itemlist={this.state.itemlist} isLog={process.env.NODE_ENV==="development"} />
                </div>
            </div>
        );
    }
}

export default App;
```


31

5. 고차 컴포넌트(7)

- 실행 결과



32

6. 컴포넌트의 설계(1)

컴포넌트를 설계할 때 고려할 점

- 재사용성
  - 독립적인 요소, 스타일을 가지며 재사용 가능한 수준에서 분할
  - 독립성, 재사용성을 높이려면 속성을 이용한 순수 컴포넌트(표현 컴포넌트, 비상태 컴포넌트)로 작성하는 것이 바람직함.
- 관리성
  - 컴포넌트 단위로 관리, 조정 가능하도록 분할
  - 한 컴포넌트 내부에서 지나치게 복잡한 작업을 수정하지 않도록 컴포넌트를 분할함.
- 렌더링 최적화
  - 렌더링할지 여부를 결정하는 단위는 컴포넌트임.
  - 부모 컴포넌트에서 render() 가 호출되면 자식 컴포넌트는 기본적으로 re-render 함.
  - 이 때 자식 컴포넌트로 세분화하지 않았다면 컴포넌트 전체가 렌더링되는 것임.
  - 예) 이전 프로젝트에서 List.js 단위로만 재사용된다고 하더라도 ListItem 컴포넌트를 작성하여 세분화시키는 것이 바람직함 → 조정 작업 처리!!

6. 컴포넌트의 설계(2)

컴포넌트를 설계할 때 고려할 점(이어서)

- 상속보다는 조합 방식을 사용
  - React.Component를 상속받아 컴포넌트를 작성하지만 계층 구조로 상속받아 전체 UI를 구성하지 않고 컴포넌트들을 조합하도록 구성하는 것이 권장됨.
  - 웹화면 = 요소들의 조합 === 컴포넌트들의 조합
- 여러 컴포넌트에서 공통적으로 사용하는 기능은?
  - 상속보다는 고차 컴포넌트(HOC:Higher Order Component)!!
  - 공통 로직의 분리가 핵심
    - 로직을 구체화시키고 공통 로직을 찾아내 분리함.