




8장 JSX에서 TSX로

리액트 컴포넌트의 타입

`@types/react` 패키지에 정의된 리액트 내장 타입

|  대표적인 리액트 컴포넌트 타입을 살펴보자

8.1.1 클래스 컴포넌트 타입

`React.Component`, `React.PureComponent`의 타입 정의

- `P`: props
- `S`: state 상태

```
// 1. Component 인터페이스
interface Component<P = {}, S = {}, SS = any>
  extends ComponentLifecycle<P, S, SS> {}
// - React 컴포넌트의 기본 타입을 정의하는 인터페이스
// - P(Props), S(State), SS(Snapshot)의 제네릭 타입을 받음
// - ComponentLifecycle 인터페이스를 상속받아 생명주기 메서드들을 포함

// 2. Component 클래스
class Component<P, S> {
  /* ... 생략 */
}
// - 실제 구현이 있는 클래스
// - class MyComponent extends React.Component로 상속받아 사용하는 2

// 3. PureComponent 클래스
class PureComponent<P = {}, S = {}, SS = any> extends Component<
// - Component 클래스를 상속받는 최적화된 버전
// - props와 state를 얕은 비교하여 불필요한 리렌더링 방지
```



클래스 컴포넌트 타입 쓰임 용도

`class Component` : 가장 기본적인 클래스 컴포넌트를 만들 때 사용한다.

- 렌더링 로직과 생명주기를 직접 제어하고 싶을 경우

`class PureComponent` : 성능 최적화가 필요한 컴포넌트에 사용한다.

- props나 state가 자주 변경되는 컴포넌트에 효과적

`interface Component` : 직접 사용하지 않고, 타입 정의용으로 사용된다.

- 커스텀 컴포넌트 타입을 정의할 때 참조한다.

```
interface WelcomeProps {
  name: string;
}

class Welcome extends React.Component<WelcomeProps> { // state E
  /* ... 생략 */
}
```

8.1.2 함수 컴포넌트 타입

함수 컴포넌트의 타입 지정을 위해 제공되는 타입

- `React.FC`
- `React.VFC`

FC : FunctionComponent



React.FC vs. React.VFC

리액트 v18로 넘어오면서 React.VFC가 삭제되고, React.FC에서 children이 사라졌다.

➡ React.FC 또는 props 타입, 반환 타입을 직접 지정하는 방식을 사용한다.

```
// 함수 선언을 사용한 방식
function Welcome(props: WelcomeProps): JSX.Element {}

// 함수 표현식을 사용한 방식 - React.FC 사용
// React.FC에 매개변수 타입과 반환 타입이 이미 정의되어 있다!
const Welcome: React.FC<WelcomeProps> = ({ name }) => {};

// 함수 표현식을 사용한 방식 - JSX.Element를 반환 타입으로 지정
const Welcome = ({ name }: WelcomeProps): JSX.Element => {};

// React의 타입 정의 파일(@types/react)
type FC<P = {}> = FunctionComponent<P>;

interface FunctionComponent<P = {}> {
  // props에 children을 추가
  // 함수 시그니처
  (props: PropsWithChildren<P>, context?: any): ReactElement<any>
  // 정적 프로퍼티들
  propTypes?: WeakValidationMap<P> | undefined;
  contextTypes?: ValidationMap<any> | undefined;
  defaultProps?: Partial<P> | undefined;
  displayName?: string | undefined;
}
```

8.1.3 Children props 타입 지정

```
type PropsWithChildren<P> = P & { children?: ReactNode | undefir
```

가장 보편적인 children 타입 : `ReactNode | undefined`

`ReactNode` : ReactElement 외에도 boolean, number 등 여러 타입을 포함하고 있는 타입

- 더 구체적으로 타이핑하는 용도에는 적합 X

| children에 특정 문자열만 허용하고 싶을 때

```
// example 1
type WelcomeProps = {
  children: "천생연분" | "더 귀한 분" | "귀한 분" | "고마운 분";
};

// example 2
type WelcomeProps = { children: string };

// example 3
type WelcomeProps = { children: ReactElement };
```

8.1.4 render 메서드와 함수 컴포넌트의 반환 타입 -

React.ReactElement VS **JSX.Element** VS **React.ReactNode**

React.ReactElement

리액트 컴포넌트를 객체 형태로 저장하기 위한 포맷

- 리액트는 실제 DOM이 아니라 가상의 DOM을 기반으로 렌더링하는데, 가상 DOM의 엘리먼트는 ReactElement 형태로 저장된다.

JSX.Element

리액트의 ReactElement를 확장하고 있는 타입

- 글로벌 네임스페이스**에 정의되어 있어 외부 라이브러리에서 컴포넌트 타입을 재정의 할 수 있는 유연성 제공한다. → 컴포넌트 타입을 재정의하거나 변경하는 것이 용이함

글로벌 네임스페이스 : 프로그래밍에서 식별자(변수, 함수, 타입 등)가 정의되어 있는 전역적인 범위.

JS나 TS에서는 기본적으로

전역(글로벌) 스코프에서 선언된 변수나 함수 등은 글로벌 네임스페이스에 속한다. 즉, 어떠한 파일이든지 해당 스코프에서 선언된 식별자는 모든 곳에서 접근 가능하다.

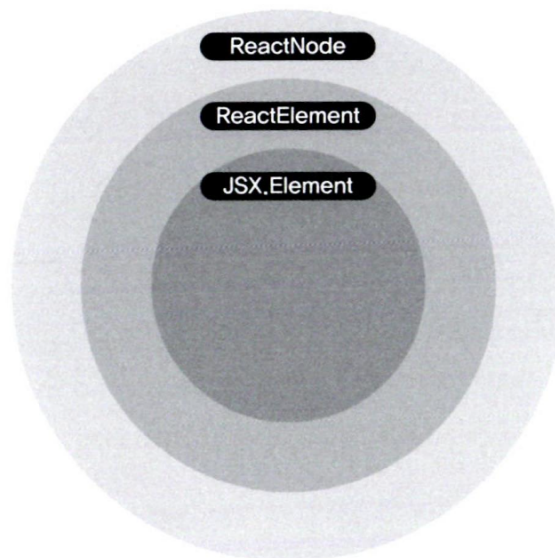
- 전역 스코프** : 모듈 시스템을 사용하지 않는 파일의 최상위 레벨

- 브라우저 환경에서의 전역 스코프
index.html의 <script> 태그 내부, 모듈이 아닌 .js/.ts 파일,
window 객체에 직접 할당된 속성들, var로 선언된 전역 변수들

React.ReactNode

ReactElement 외에도 boolean, string, number 등의 여러 타입을 포함하고 있다.

✓ ReactNode, JSX.Element, ReactElement 간의 포함 관계



8.1.5 ReactElement, ReactNode, JSX.Element 활용하기

리액트의 요소를 나타내는 타입 : `ReactElement` , `ReactNode` , `JSX.Element`

🚩 리액트의 요소를 나타내는 데 왜 이렇게 많은 타입이 존재하는지 알아
보자

ReactElement

`createElement` 메서드 : 리액트 엘리먼트를 생성한다.

React 엘리먼트 : 화면에 표시하려는 내용을 담은 객체

JSX : createElement 메서드를 호출하기 위한 문법. 즉, JSX는 리액트 엘리먼트를 생성하기 위한 문법

JSX

JS의 확장 문법으로 리액트에서 UI를 표현하는 데 사용된다.

- XML과 비슷한 구조
- 리액트 컴포넌트를 선언하고 사용할 때 더욱 간결하고 가독성 있게 코드 작성할 수 있게 도와준다.
- HTML과 유사한 문법 제공
- 컴포넌트 구조와 계층 구조를 편리하게 표현할 수 있게 해준다.

▼ 예시

```
// JSX로 작성된 엘리먼트
const element = <h1>Hello, React</h1>;

// 실제로는 이렇게 변환됨 (createElement 호출)
const element = React.createElement('h1', null, 'Hello, React')

// 결과물 (실제 React 엘리먼트 객체)
{
  type: 'h1',
  props: {
    children: 'Hello, React'
  },
  key: null,
  ref: null
}
```

🌟 JSX → createElement → 리액트 엘리먼트 객체

리액트는 이런 식으로 만들어진 리액트 엘리먼트 객체를 읽어서 DOM을 구성한다.

리액트에는 여러 개의 createElement 오버라이딩 메서드가 존재하는데, 이 메서드들이 반환하는 타입은 ReactElement 타입을 기반으로 한다.



ReactElement 타입 : JSX의 createElement 메서드 호출로 생성된 리액트 엘리먼트를 나타내는 타입

ReactNode

ReactChild : ReactElement | string | number 타입으로, ReactElement보다는 좀 더 넓은 범위를 가지고 있다.

- JSX.Element < ReactElement < ReactChild < ReactNode

ReactNode : 리액트의 render 함수가 반환할 수 있는 모든 형태를 담고 있다. (ReactChild, boolean, null, undefined ...)

JSX.Element

ReactElement의 제네릭으로 props와 타입 필드에 대해 any 타입을 가지도록 확장하고 있다.

```
declare global {
  namespace JSX {
    interface Element extends React.ReactElement<any, any> {
      // ...
    }
    // ...
  }
}
```

JSX.Element : ReactElement의 특정 타입으로, props와 타입 필드를 any로 가지는 타입



JSX.Element < ReactElement < ReactNode

ReactNode : 리액트의 render 함수가 반환할 수 있는 모든 형태를 담고 있다.

ReactElement : JSX의 createElement 메서드 호출로 생성된 리액트 엘리먼트를 나타내는 타입

JSX.Element : ReactElement의 특정 타입으로, props와 타입 필드를 any로 가지는 타입

- 글로벌 네임스페이스에 정의되어 있다.

8.1.6 사용 예시

모두 리액트에서 제공하는 컴포넌트를 나타낸다.

ReactNode

리액트 컴포넌트가 가질 수 있는 모든 타입

✓ ReactNode 타입으로 children을 선언하는 경우

- 어떤 타입이든 children prop으로 지정할 수 있게 하고 싶을 경우

```
interface MyComponentProps {
  children?: React.ReactNode;
  // ...
}
```

- prop으로 리액트 컴포넌트가 다양한 형태를 가질 수 있게 하고 싶을 경우

```
type PropsWithChildren<P = unknown> = P & {
  children?: ReactNode | undefined;
};

interface MyProps {
  // ...
}

type MyComponentProps = PropsWithChildren<MyProps>;
```

▼ 사용 예시


```

// 기본 props 인터페이스
interface CardProps {
  title: string;
  color?: string;
}

// children을 포함한 최종 props 타입
type CardComponentProps = PropsWithChildren<CardProps>;
// 결과
// interface CardProps {
//   title: string;
//   color?: string;
//   children?: ReactNode; // PropsWithChildren이 추가해주는 부분
// }

// 컴포넌트에서 사용
const Card = ({ title, color, children }: CardComponentProps)
  return (
    <div style={{ color }}>
      <h2>{title}</h2>
      {children}
    </div>
  );
};

// 사용 예시
<Card title="제목" color="blue">
  <p>이것이 children입니다</p>
</Card>

```

JSX.Element

props와 타입 필드가 any 타입인 리액트 엘리먼트를 나타내므로 → **리액트 엘리먼트**를 prop으로 전달받아 render props 패턴으로 컴포넌트를 구현할 때 유용하게 활용 가능하다.

- icon prop을 JSX.Element 타입으로 선언함으로써 해당 prop에는 **JSX문법만** 삽입할 수 있다.
- icon.prop에 접근하여 **prop으로 넘겨받은 컴포넌트의 상세한 데이터**를 가져올 수 있다.

```

interface Props {
  icon: JSX.Element; // React 엘리먼트만 받을 수 있음 (JSX 문법으로 작
}

const Item = ({ icon }: Props) => {
  // prop으로 받은 컴포넌트의 props에 접근할 수 있다
  const iconSize = icon.props.size;

  return <li>{icon}</li>;
};

// icon prop에는 JSX.Element 타입을 가진 요소만 할당할 수 있다
const App = () => {
  return <Item icon={<Icon size={14} />} />;
};

```

▼ 실제 사용 예시

```

interface CardProps {
  thumbnail?: JSX.Element;
  action?: JSX.Element;
  children?: React.ReactNode;
}

const Card = ({ thumbnail, action, children }: CardProps) =>
  return (
    <div className="card">
      {thumbnail && <div className="card-thumbnail">{thumbnail}
      <div className="card-content">{children}</div>
      {action && <div className="card-action">{action}</div>}
    </div>
  );
};

// 사용
<Card
  thumbnail={<Image src="/thumbnail.jpg" />}
  action={<Button>자세히 보기</Button>}
>

```

```
<h2>제목</h2>
<p>내용</p>
</Card>
```



▼ `icon.size` 가 아니라 `icon.props.size` 로 접근해야 하는 이유

```
// JSX로 작성한 코드
<Icon size={14} />

// 실제 React 엘리먼트 객체 구조
{
  type: Icon,
  props: {
    size: 14
  },
  key: null,
  ref: null
}
```

React 엘리먼트는 항상 이런 구조를 가지기 때문에:

1. `type`: 해당 엘리먼트의 타입 (HTML 태그나 React 컴포넌트)
2. `props`: 모든 속성들이 이 객체 안에 저장됨

→ 속성에 접근하려면 반드시 `props` 를 통해야 한다.

- JSX.Element(컴포넌트)를 전달받을 때 → `.props` 로 접근
- 일반 값을 전달받을 때 → 직접 접근



JSX 문법으로 작성된 **단일** React 엘리먼트만 허용한다.

ReactElement

JSX.Element 대신에 ReactElement를 사용하면, JSX.Element 예시를 확장하여 추론 관점에서 더 유용하게 활용할 수 있다.

- 원하는 컴포넌트의 props를 ReactElement의 제네릭으로 지정해줄 수 있다.
- JSX.Element가 ReactElement의 props 타입으로 `any`가 지정되었다면, ReactElement 타입을 활용하여 제네릭에 직접 해당 컴포넌트의 props 타입을 명시 해준다.

```
// JSX.Element는 props가 any 타입
interface Props {
  icon: JSX.Element;
}

const Item = ({ icon }: Props) => {
  // 타입 추론이 불가능, any 타입
  console.log(icon.props.size);
  return <li>{icon}</li>;
};
```

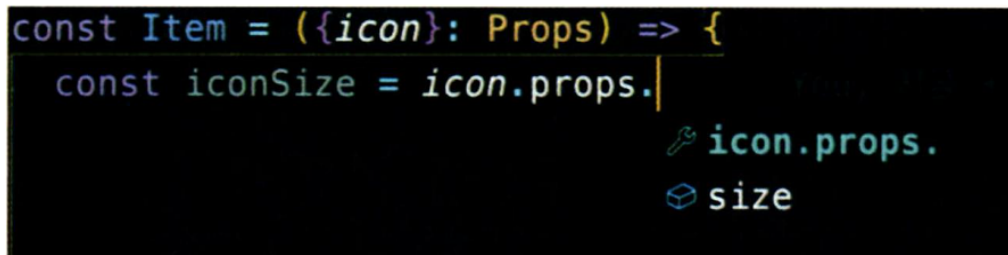
```
interface IconProps {
  size: number;
}

interface Props {
  // ReactElement의 props 타입
  icon: React.ReactElement<IconProps>;
}

const Item = ({ icon }: Props) => {
  // icon prop으로 받은 컴포넌트의 props
  const iconSize = icon.props.size;

  return <li>{icon}</li>;
};
```

→ icon.props에 접근할 때 어떤 props가 있는지 추론되어 IDE에 표시된다!



```
const Item = ({icon}: Props) => {
  const iconSize = icon.props.  

    icon.props.  

    size
```

출처: 리액트 공식 문서 - JSX 소개 <https://ko.reactjs.org/docs/introducing-jsx.html>



사용 상황

- 단순 렌더링 → `JSX.Element`
- props 접근/조작 필요 → `React.ReactElement<Props>`

▼ 예시

```
const IconContainer = ({ icon }: { icon: React.ReactElement<Props> }) => {
  // props의 구체적인 타입 정보가 필요한 경우
  const size = icon.props.size; // number로 정확히 추출
  const color = icon.props.color; // string으로 정확히 추출

  return (
    <div style={{ width: size, color: color }}>
      {icon}
    </div>
  );
};
```

8.1.7 리액트에서 기본 HTML 요소 타입 활용하기

| HTML button 태그를 확장한 Button 컴포넌트

```
// 기존 HTML button과 같은 역할을 하면서도 새로운 기능이나 UI가 추가된 형태
const SquareButton = () => <button>정사각형 버튼</button>;
```



기존 HTML 태그의 속성 타입을 활용하여 타입을 지정하는 방법에 대해 알아보자

DetailedHTMLProps와 ComponentWithoutRef

HTML 태그의 속성 타입을 활용하는 대표적인 2가지 방법: `DetailedHTMLProps`, `ComponentPropsWithoutRef`

`React.DetailedHTMLProps` 활용

```
// HTML 버튼 엘리먼트의 모든 기본 속성들을 포함하는 타입
type NativeButtonProps = React.DetailedHTMLProps<
  React.ButtonHTMLAttributes<HTMLButtonElement>, // 버튼의 모든 HTML 속성
  HTMLButtonElement // 버튼 엘리먼트 자체의 타입
>;

type ButtonProps = {
  onClick?: NativeButtonProps["onClick"]; // 기본 버튼의 onClick 이벤트 핸들러
};
```

`React.ComponentPropsWithoutRef` 타입 활용

```
type NativeButtonType = React.ComponentPropsWithoutRef<"button">
type ButtonProps = {
  onClick?: NativeButtonType["onClick"];
};
```

언제 `ComponentPropsWithoutRef`를 사용하면 좋을까?

컴포넌트의 props로서 HTML 태그 속성을 확장하고 싶은 상황
: HTML button 태그와 동일한 역할을 하지만, 커스텀한 UI를 적용하여 재사용성을 높이기 위한 Button 컴포넌트를 만들자

```
const Button = () => {
  return <button>버튼</button>;
};
```

```
type NativeButtonProps = React.DetailedHTMLProps<
  React.ButtonHTMLAttributes<HTMLButtonElement>,
  HTMLButtonElement
>;
```

```
const Button = (props: NativeButtonProps) => {
  return <button {...props}>버튼</button>; // HTML button 태그를 다
};
```

✓ ref를 props로 받을 경우

ref를 통해 button 태그에 접근하여 DOM 노드를 조작할 수 있을 것으로 예상된다.

ref : 생성된 DOM 노드나 리액트 엘리먼트에 접근하는 방법

▼ ref 예시(클래스/함수 컴포넌트에서)

```
// 클래스 컴포넌트
class Button extends React.Component {
  constructor(props) {
    super(props);
    this.buttonRef = React.createRef();
  }

  render() {
    return <button ref={this.buttonRef}>버튼</button>;
  }
}

// 함수 컴포넌트
function Button(props) {
  const buttonRef = useRef(null);

  return <button ref={buttonRef}>버튼</button>;
}
```

```
type NativeButtonProps = React.DetailedHTMLProps<
  React.ButtonHTMLAttributes<HTMLButtonElement>,
```

```

HTMLButtonElement
>;

// 클래스 컴포넌트
class Button extends React.Component {
  constructor(ref: NativeButtonProps["ref"]) {
    this.buttonRef = ref;
  }

  render() {
    return <button ref={this.buttonRef}>버튼</button>;
  }
}

// 함수 컴포넌트
function Button(ref: NativeButtonProps["ref"]) {
  const buttonRef = useRef(null);

  return <button ref={buttonRef}>버튼</button>;
}

```

! 클래스/함수 컴포넌트에서 ref를 props로 받아 전달하는 방식에 차이가 있다.

클래스 컴포넌트 에서 ref 객체는 마운트된 컴포넌트의 인스턴스를 current 속성값으로 가지지만,

함수 컴포넌트 에서는 생성된 인스턴스가 없기 때문에 ref에 기대한 값이 할당되지 않는다.

- **클래스 컴포넌트** 로 만들어진 버튼

컴포넌트 props로 전달된 ref가 Button 컴포넌트의 button 태그를 그대로 바라보게 된다.

```

// 클래스 컴포넌트로 만들어진 Button 컴포넌트를 사용할 때
class WrappedButton extends React.Component {
  // 클래스는 인스턴스를 생성한다.
  // this 키워드로 인스턴스에 접근 가능하다.
  // 상태와 메서드들이 인스턴스에 저장된다.

  // 클래스는 인스턴스를 생성하므로 this.buttonRef를 통해 Button 콘
  // this.buttonRef.current에는 Button 컴포넌트의 인스턴스가 저장된다.

  constructor() {

```



```

    this.buttonRef = React.createRef();
  }

  render() {
    return (
      <div>
        <Button ref={this.buttonRef} />
      </div>
    );
  }
}

```

- **함수 컴포넌트** 로 만들어진 버튼

기대와 달리 전달받은 ref가 Button 컴포넌트의 button 태그를 바라보지 않는다.

```

// 함수 컴포넌트로 만들어진 Button 컴포넌트를 사용할 때
const WrappedButton = () => {
  // 함수는 그냥 실행될 뿐, 인스턴스를 생성하지 않는다.
  // 호출될 때마다 새로 실행된다.
  // 상태는 hooks를 통해 react가 별도로 관리한다. (useState)

  // buttonRef.current에는 Button 컴포넌트의 인스턴스가 없으므로,

  const buttonRef = useRef();

  return (
    <div>
      <Button ref={buttonRef} />{" "}
    </div>
  );
};

```



React.forwardRef 메서드 : 함수 컴포넌트에서도 ref를 전달받을 수 있도록 해준다.

```

// forwardRef를 사용해 ref를 전달받을 수 있도록 구현
// 함수는 인스턴스가 없어서 button 엘리먼트에 직접 전달해야 한다.
const Button = forwardRef((props, ref) => {
  return (

```

```

        // 받은 ref를 직접 button에 전달
        <button ref={ref} {...props}>
            버튼
        </button>
    );
});

// buttonRef가 Button 컴포넌트의 button 태그를 바라볼 수 있다
const WrappedButton = () => {
    const buttonRef = useRef();

    return (
        <div>
            <Button ref={buttonRef} />
        </div>
    );
};

```

- forwardRef는 2개의 제네릭 인자를 받는다.

1. ref에 대한 타입 정보
2. props에 대한 타입 정보

```

type NativeButtonType = React.ComponentPropsWithoutRef<"button">
// button 태그에 대한 HTML 속성을 모두 포함하지만, ref 속성은 제외된다.

// forwardRef의 제네릭 인자를 통해 ref에 대한 타입으로 HTMLButtonElement
const Button = forwardRef<HTMLButtonElement, NativeButtonType>((
    return (
        <button ref={ref} {...props}>
            버튼
        </button>
    );
});

```



ComponentPropsWithoutRef vs. DetailedHTMLProps

`ComponentPropsWithoutRef` : 해당 요소의 HTML 속성을 모두 포함하지만, ref 속성은 제외된다.

`DetailedHTMLProps` : 해당 요소의 HTML 속성과 ref 속성을 포함한다.

! 함수 컴포넌트의 props로 ref를 포함하는 타입 `DetailedHTMLProps` 을 사용하게 되면, 실제로는 동작하지 않는 ref를 받도록 타입이 지정되어 예기치 않은 에러가 발생할 수 있다.

→ HTML 속성을 확장하는 props를 설계할 때는 `ComponentPropsWithoutRef` 타입을 사용하여, ref가 실제로 `forwardRef`와 함께 사용될 때만 props로 전달되도록 타입을 정의하는 것이 안전하다.