About React

Chapter 1. JSX

김동연

Simple list

왜 JSX를 사용하는가?

JSX의 특징

JSX의 작동 원리

React.createElement()에 대한 설명

JSX의 주요 특징

JSX의 다양한 특징들

구현

최종 코드로 구현

JSX란?

JSX(JavaScript XML)는

- JavaScript 내에서 HTML과 유사한 문법을 사용할 수 있게 해주는 문법 확장
- React에서 주로 사용, UI를 정의하기 위해 작성된 코드를 보다 읽기 쉽게 만들어줌
- JSX는 브라우저에서 직접 실행되지 않으며, Babel 같은 도구를 통해 JS로 변환되어 실행

jsx

```jsx
function Greeting() {
  const name = "김동연";

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>React 공부를 시작해봅시다!</p>
    </div>
  );
}

export default Greeting;
```

Babel이란?

자바스크립트 트랜스파일러(code → code)

- 최신 문법(ES6)을 이전 버전(ES5)으로 변환
- TS, TSX, JSX 등도 JS로 변환

이유 → 코드 호환성을 위해 최신문법을 이전 버전으로 변환

Put in next-gen JavaScript

```
function App() {
  return <h1>Hello, world!</h1>;
}
```

Get browser-compatible JavaScript out

```
function App() {
  return /*#__PURE__*/React.createElement
    ("h1", null, "Hello, world!");
}
```

컴파일러 vs 트랜스파일러?

컴파일러(Compiler)

- 정의 고수준 언어(예: C, Java, Python)를 저수준 언어 (예: 기계어, 어셈블리어)로 변환하는 도구.

트랜스파일러(Transpiler)

- 정의: 같은 수준의 언어(주로 고수준 언어)에서 다른 언어로 변환하는 도구.

기준	컴파일러	트랜스파일러
변환 대상	고수준 언어 → 저수준 언어	고수준 언어 → 고수준 언어
예시	C → 기계어, Java → 바이트코드	ES6 → ES5, TS → JS
사용 목적	CPU 실행 가능 코드 생성	코드 호환성 또는 문법 변환

왜 JSX를 사용하는가?

1. HTML과 JavaScript의 결합

JSX는 HTML과 유사한 구조를 JavaScript 안에서 작성할 수 있음
→ 컴포넌트 기반 UI 설계가 직관적이고 간편

jsx

```jsx
function Greeting() {
  const name = "김동연";

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>React 공부를 시작해봅시다!</p>
    </div>
  );
}

export default Greeting;
```

왜 JSX를 사용하는가?

2. 직관적인 문법

HTML과 닮은 문법 덕분에, 코드 가독성이 높아지고
DOM 요소와 React 컴포넌트를 결합하는 작업이 쉽다.

React 컴포넌트를 HTML 태그처럼 사용할 수 있음.

```
function App() {
  return (
    <div>
      <Header title="Welcome to My Website" />
      <main>
        <p>This is the main content of the website.</p>
      </main>
      <Footer />
    </div>
  );
}
```

DOM & DOM 요소

DOM(Document Object Model)?

- HTML 문서 구조를 브라우저 메모리에 저장한 트리 구조

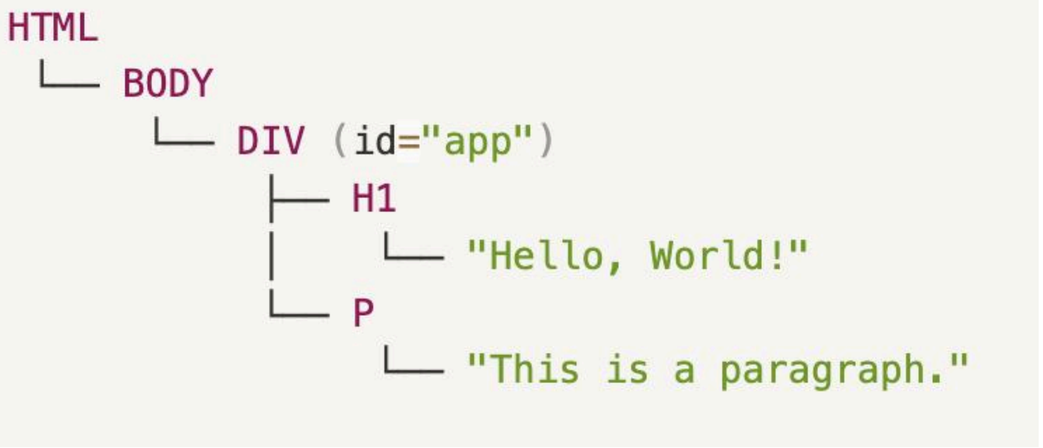
DOM 요소?

- HTML 태그들

HTML

```html
<div id="app">
  <h1>Hello, World!</h1>
  <p>This is a paragraph.</p>
</div>
```

→



왜 JSX를 사용하는가?

3. 동적인 데이터와의 자연스러운 연결

JavaScript 표현식을 포함할 수 있어 동적인 데이터를 UI에 쉽게 반영

```
const name = "Kim Dongyeon";
const element = <h1>Hello, {name}</h1>;
```

표현식?

값을 생성하는 코드 조각 = 표현식이 실행하면 어떤 값이 계산되어 결과로 반환됨

표현식 vs 문

표현식	문(statement)
값을 반환	동작을 수행
다른 코드의 일부로 사용 가능	단독으로 사용
`2+2`, `Math.max(3, 5)`, `x > y` 등	`if`, `for`, `while`, `let`, `return` 등

JSX의 작동 원리

- JSX는 Babel을 통해 **React.createElement()** 호출로 변환(React에서)

- 이 함수는 Virtual DOM 요소를 생성하고, React가 이를 이용해 실제 DOM을 업데이트할 수 있음

React.createElement() 란?

- JSX를 사용할 때 내부적으로 호출되는 함수
- React의 Virtual DOM에서 사용할 객체를 생성

```
React.createElement(
    type,           // 요소의 타입 (HTML 태그 또는 컴포넌트 명)
    [props],        // 속성 (객체 형태, ex: { className: 'container', id: 'main' }, 없으면 null)
    [...children]   // 자식 요소 (단일 자식: 문자열, 숫자, JSX 요소, 여러 자식: 배열 형태)
  )
```

React.createElement() 이 하는 일

- Virtual DOM 요소 생성
  - 실제 DOM에 렌더링할 수 있는 객체를 만듦
  - React는 이 객체를 이용해 변경 사항을 추적하고, 실제 DOM을 업데이트
- 재사용 가능한 컴포넌트 구조 지원
  - 사용자 정의 컴포넌트(function 또는 class)를 호출하여 컴포넌트 계층 구조를 생성합니다.

```
React.createElement(
    type,           // 요소의 타입 (HTML 태그 또는 컴포넌트 명)
    [props],        // 속성 (객체 형태, ex: { className: 'container', id: 'main' }, 없으면 null)
    [...children]   // 자식 요소 (단일 자식: 문자열, 숫자, JSX 요소, 여러 자식: 배열 형태)
  )
```

React.createElement() 의 작동 예시

1. JSX 코드

```jsx
const element = <h1 className="title">Hello, world!</h1>;
```

2. 변환된 JS 코드

```jsx
const element = React.createElement(
  'h1',
  { className: 'title' },
  'Hello, world!'
);
```

3. React.createElement의 변환 결과 (React Element)

```jsx
{
  type: 'h1',
  props: {
    className: 'title',
    children: 'Hello, world!'
  }
}
```

React.createElement() 의 작동 방식

1. 속성 병합
    - 속성을 병합하여 props 객체를 생성합니다.
    - 예: { className: 'container', id: 'main' }

2. 키 및 참조 처리 (React Element에 존재)
    - key 속성은 React가 리스트의 요소를 구분하는 데 사용
    - ref 속성은 DOM 요소나 컴포넌트에 접근하기 위한 참조를 제공

3. 자식 요소 처리
    - 자식 요소가 문자열이라면 그대로 children 속성에 할당
    - 여러 자식 요소는 배열로 병합됩니다.

JSX의 주요 특징

1. JavaScript와 XML의 결합

- JSX는 JavaScript 코드 내에 XML처럼 보이는 문법을 작성할 수 있게 해줍니다. 이를 통해 UI를 정의하는 코드가 직관적으로 보이도록 도와줍니다.
- JavaScript 표현식 삽입: 중괄호 {}를 사용하여 동적인 값을 추가하거나, 함수 호출, 삼항 연산자 등 다양한 표현식을 사용할 수 있습니다.

```
const name = "Kim Dongyeon";
const element = <h1>Hello, {name}</h1>;
```

JSX의 주요 특징

2. DOM 요소와 React 컴포넌트 구분

- 소문자 시작: HTML DOM 요소를 생성. (예: <h1>, 등)
- 대문자 시작: React 컴포넌트로 간주하여 import된 컴포넌트를 렌더링합니다. (예: <Header/> 등)
- 예외: HTML과 충돌하지 않는 사용자 정의 태그를 사용할 수도 있습니다(예: <custom-element />).

JSX의 주요 특징

3. JSX 문법의 유효성

- 모든 태그는 닫혀야 함: HTML에서는 <img> 같은 태그가 닫히지 않아도 되지만, JSX에서는 모든 태그가 반드시 닫혀야 합니다.
- 하나의 루트 노드: JSX는 반드시 하나의 루트 요소를 반환해야 합니다. 여러 요소를 반환하려면 <></> 또는 <React.Fragment>로 감싸야 합니다.

```
return (
  <>
    <h1>Title</h1>
    <p>Description</p>
  </>
);
```

JSX의 주요 특징

4. CSS 스타일링

- JSX에서 style 속성은 객체 형태로 작성하며, 속성 이름은 camelCase로 작성해야 합니다.
- 동적인 스타일을 추가할 때에도 중괄호 {}로 표현할 수 있습니다

```
const color = "blue";
const element = <h1 style={{ color }}>Dynamic Style</h1>;
```

JSX의 주요 특징

5. HTML 속성과 차이점

- JSX에서 일부 속성은 HTML 속성과 이름이 다릅니다.
- 예를 들어
  - class → className
  - for → htmlFor

```jsx
<label htmlFor="name">Name:</label>
<input id="name" className="input-field" />
```

JSX의 주요 특징

6. JSX는 JavaScript로 변환됨

- JSX는 실제로는 JavaScript 함수 호출로 변환됩니다.

```jsx
<label htmlFor="name">Name:</label>
<input id="name" className="input-field" />
```

구현

1. createElement.js

```
// 1단계, Virtual DOM 생성

export function createElement(type, props, ...children) {
  return {
    type,
    props: {
      ...props,
      children: children.map((child) =>
        typeof child === "object" ? child : createTextElement(child)
      ),
    },
  };
}

export function createTextElement(text) {
  return {
    type: "TEXT_ELEMENT",
    props: { nodeValue: text, children: [] },
  };
}
```

2. render.js

```
// 2단계, Virtual DOM을 실제 DOM으로 변환

import { JSDOM } from "jsdom";

// 가상 DOM 생성
const { window } = new JSDOM(`<html><body><div id="root"></div></body></html>`);
const { document } = window;

export function render(vNode, container) {
  const dom =
    vNode.type === "TEXT_ELEMENT"
      ? document.createTextNode(vNode.props.nodeValue)
      : document.createElement(vNode.type);

  Object.keys(vNode.props)
    .filter((key) => key !== "children")
    .forEach((name) => {
      dom[name] = vNode.props[name];
    });

  vNode.props.children.forEach((child) => render(child, dom));
  container.appendChild(dom);
  // DOM 구조를 콘솔에 출력
  console.log("생성된 Virtual DOM 구조:");
  console.dir(vNode, { depth: null });

  console.log("\n실제 DOM 구조:");
  console.log(container.innerHTML);
}
```

3. updateDom.js

```
// 3단계, 실제 DOM 업데이트 로직(reconcile 함수)

import { render } from "./render.js";

function updateDom(dom, prevProps, nextProps) {
  // 기존 속성 제거
  Object.keys(prevProps)
    .filter((name) => name !== "children")
    .forEach((name) => {
      if (!(name in nextProps)) {
        dom[name] = "";
      }
    });

  // 새 속성 추가
  Object.keys(nextProps)
    .filter((name) => name !== "children")
    .forEach((name) => {
      dom[name] = nextProps[name];
    });
}
```

```
export function reconcile(parent, oldVNode, newVNode) {
  console.log("=== Reconciliation Start ===");
  console.log("Parent:", parent);
  console.log("Old VNode:", oldVNode);
  console.log("New VNode:", newVNode);

  if (!oldVNode) {
    console.log("📌 Adding new node");
    render(newVNode, parent);
  } else if (!newVNode) {
    console.log("🗑️ Removing old node");
    parent.removeChild(parent.childNodes[0]);
  } else if (oldVNode.type !== newVNode.type) {
    console.log("🔄 Replacing node");
    console.log("Old type:", oldVNode.type);
    console.log("New type:", newVNode.type);
    parent.replaceChild(
      render(newVNode, document.createElement(newVNode.type)),
      parent.childNodes[0]
    );
  } else {
    console.log("🔧 Updating node properties");
    updateDom(parent.firstChild, oldVNode.props, newVNode.props);

    // 자식 노드 재귀 처리
    const maxLength = Math.max(
      oldVNode.props.children.length,
      newVNode.props.children.length
    );
    console.log(`👶 Processing ${maxLength} children`);

    for (let i = 0; i < maxLength; i++) {
      console.log(`Child ${i + 1}/${maxLength}`);
      reconcile(
        parent.firstChild,
        oldVNode.props.children[i],
        newVNode.props.children[i]
      );
    }
  }

  console.log("=== Reconciliation End ===\n");
}
```

4. Test.js

```
// 가상 DOM 환경 생성
const { window } = new JSDOM(`<html><body><div id="root"></div></body></html>`);
const { document } = window;

// 초기 Virtual DOM
const oldVNode = createElement(
  "div",
  { id: "app" },
  createElement("h1", null, "Hello, world!"),
  createElement("p", { style: "color: red;" }, "This is old content.")
);

// 변경된 Virtual DOM
const newVNode = createElement(
  "div",
  { id: "app" },
  createElement("h1", null, "Hello, Virtual DOM!"),
  createElement("p", { style: "color: blue;" }, "This is updated content."),
  createElement(
    "button",
    { onclick: () => console.log("Clicked!") },
    "Click me!"
  )
);
```

```
// 실제 DOM 컨테이너 생성
const root = document.getElementById("root");

// 초기 렌더링
console.log("🌟 Initial Render");
reconcile(root, null, oldVNode);
console.log(root.outerHTML);

// 업데이트 렌더링
console.log("🌟 Updating DOM");
reconcile(root, oldVNode, newVNode);
console.log(root.outerHTML);
```

결과