**000. 리액트 개발 환경 준비하기**

개발 환경을 준비하기에 앞서, 설치도구들에 대한 정리가 필요하다. 자바스크립트 패키지 관리도구로는 npm와 yarn이 있다. 먼저 npm은 node package manager의 약자로 node.js와 react.js에서 사용하는 대부분의 패키지들을 설치할 수 있다. npm의 경우 node.js와 함께 설치된다. yarn은 facebook에서 만든 패키지 관리 도구로 npm에 비해 캐싱, 보안, 신뢰성 등이 개선되었다. 이 책에서는 part1~4의 react예제에 활용되는 패키지들을 yarn으로 설치하고, node.js 실습을 하는 part5에서 npm을 사용할 것이다.

**1. node.js 다운로드**node.js 공식 홈페이지(nodejs.org)에 접속해 12.16.1 버전(node-v12.16.1-x64.msi)을 다운로드한다. <https://nodejs.org/download/release/v12.16.1/> URL로 접근하면 다양한 확장자의 파일을 다운로드할 수 있다.

|  |
| --- |
|  |

**|그림 000-1| node.js 공식 홈페이지**

**2. node.js 설치**

다운로드 파일을 실행해 설치를 시작한다.

|  |
| --- |
|  |

**|그림 000-2|**

설치진행 중 다음 화면에서 [Automatically install the necessary tools. Note that this will also install Boxstarter and Chocolatey. The script will pop-up in a new window after the installation completes]는 체크하지 않고 [Next]를 클릭한다.

|  |
| --- |
|  |

**|그림 000-3|**

설치를 완료한다.

|  |
| --- |
|  |

**|그림 000-4|**

**3. node.js 및 npm 설치 확인**

[윈도우 키] + R을 누른 뒤, cmd를 입력해 명령 프롬프트를 실행한다. node -v, npm -v명령어로 설치된 node.js와 npm버전을 확인한다.

|  |
| --- |
|  |

**|그림 000-5|**

**5. react프로젝트의 workspace로 사용할 폴더를 생성한다.**

cmd창에서 cd 명령어를 이용해 생성한 폴더 경로로 이동한다.

|  |
| --- |
|  |

**|그림 000-6|**

**6. yarn을 설치한다.**

npm install명령어로 yarn을 설치한다. yarn -v명령어로 설치된 yarn 버전을 확인한다.

|  |
| --- |
|  |

**|그림 000-7|**

**7. create-react-app을 설치한다.**

npm install -g create-react-app명령어로 create-react-app을 설치한다. create-react-app client명령어로 client라는 프로젝트를 생성한다.

|  |
| --- |
|  |

**|그림 000-8|**

프로젝트 생성 후, client 경로를 보면 package.json, node\_modules등의 파일, 폴더가 생성된 것을 확인할 수 있다.

|  |
| --- |
|  |

**|그림 000-9|**

**8. react서버를 실행한다.**

client경로에서 yarn start명령어로 react서버를 실행한다.

|  |
| --- |
|  |

**|그림 000-10|**

chrome브라우저에서 http://localhost:3000 URL을 열면 다음과 같은 화면을 확인할 수 있다.

|  |
| --- |
|  |

**|그림 000-11|**

**001. jsx에 html 적용하기**

**• 학습 내용 :** react에서 html코드를 적용하는 방법을 이해한다.

**• 힌트 내용 :** jsx 소스에서 return()안에 html코드를 입력한다.

create-react-app템플릿으로 프로젝트를 생성하면, App.js의 html코드를 이용해 홈 화면을 그려준다. 여기서 홈 화면이란 localhost:3000(도메인) 또는 127.0.0.1:3000(IP)와 같이, 페이지 상세 URL이 뒤에 붙지 않은 웹사이트의 대표 페이지를 말한다.

create-react-app템플릿의 홈 화면인 App.js 파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12 | import React from 'react';  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>HTML 적용하기</p>  </div>  );  }  export default App; |

5-8: 스타일이 적용되지 않은, 기본 HTML코드만 삽입된 상태이다. 브라우저에서 페이지를 열면, 아래 화면을 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**002. jsx에 css 적용하기**

**• 학습 내용** : react에서 css파일로 html코드에 스타일을 적용하는 방법을 이해한다.

**• 힌트 내용** : css파일을 별로도 만들고 jsx파일에서 import해 사용한다.

css란 html과 같은 문서에 스타일을 적용할 때, 사용하는 언어이다. css를 적용하는 방법은 크게 3가지가 있다. css코드를 1) html코드 내부에 삽입하는 방법, 2) html코드가 있는 파일에 함께 정리하는 방법, 3) css코드만 따로 파일로 작성하는 방법이다. 이번 예제에서는 3번째 방법으로 css를 적용한다.

App.css 파일을 다음과 같이 수정한다.

<소스> App.css

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14 | div {  background-color: rgb(162, 127, 243);  color: rgb(255, 255, 255);  padding: 40px;  font-family: 고딕;  text-align: center;  }  h1 {  color: white;  background-color: #2EFE2E;  padding: 10px;  font-family: 궁서;  } |

1-7: div태그의 속성 중 background-color : 배경색, color : 글자색, padding : 태그 안쪽 여백, font-family : 글자폰트, text-align : 글자 정렬 값을 지정하였다.

9-14: h1 태그의 속성 중 color : 글자색, background-color : 배경색, padding : 태그 안쪽 여백, font-family : 글자폰트 값을 지정하였다.

App.js 파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13 | import React from 'react';  import './App.css';  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  </div>  );  }  export default App; |

2: App.js와 같은 폴더 경로에 위치한 App.css를 import한다. App.css가 App.js보다 한 단계 상위 폴더에 위치한다면, import './../App.css'와 같이 사용해 상위 경로에 접근해야한다.

6-9: App.css에서 작성되어있는 <div>와 <h1>태그에 스타일이 적용된다.

<실행 결과>

|  |
| --- |
|  |

**003. Component 사용하기**

**• 학습 내용 :** react에서 component를 사용해, 다른 파일에 있는 html코드를 이식해 사용하는 방법을 이해한다.

**• 힌트 내용 :** 다른 파일에서 작성한 html코드를 componet단위로 import해 사용한다.

component란 특정 코드 뭉치를 다른 부분에 이식하거나, 재사용하기 위해 사용하는 코드 블록 단위이다. 파일단위로 component를 작성해 필요한 위치에서 import해 사용할 수 있다.

src폴더에 R003\_ImportConponent.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R003\_ImportConponent.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11 | import React, { Component } from 'react';  class R003\_ImportConponent extends Component {  render () {  return (  <h2>[ THIS IS IMPORTED COMPONENT ]</h2>  )  }  }  export default R003\_ImportConponent; |

3: 1번째 줄에서 import한 component를 상속받아, R003\_ImportConponent클래스에서 사용한다.

4-6: return된 html코드를 render()함수를 통해 화면에 표시합니다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14 | import React from 'react';  import './App.css';  import ImportConponent from './R003\_ImportConponent'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <ImportConponent></ImportConponent>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R003\_ImportConponent.js파일을 import해 사용 가능하도록 한다.

10: R003\_ImportConponent.js에서 작성된 component를 사용한다. 이때, render()함수에서 읽어 들였던 '<h2>[ THIS IS IMPORTED COMPONENT ]</h2>' 코드가 이식된다.

<실행 결과>

|  |
| --- |
|  |

**004. 생명주기 함수 render() 사용하기**

**• 학습 내용 :** 컴포넌트의 생명주기 함수 중, render()함수에 대해 이해한다.

**• 힌트 내용 :** render()함수가 실행될 때, 로그를 출력하고 console에서 로그를 확인한다.

react에서 생명주기란 component의 생성 > 변경 > 소멸의 과정을 뜻한다. 4~7번 예제에서 학습하는 constructor(), render(), getDerivedStateFormProps(), componentDidMount() 함수들은 component의 '생성' 과정에 속하는 함수이다.

src폴더에 R004\_LifecycleEx.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R004\_LifecycleEx.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12 | import React, { Component } from 'react';  class R004\_LifecycleEx extends Component {  render() {  console.log('3. render Call');  return (  <h2>[ THIS IS RENDER FUCNTION ]</h2>  )  }  }  export default R004\_LifecycleEx; |

4-9: render()는 return되는 html형식의 코드를 화면에 그려주는 함수이다. 화면 내용이 변경되어야 할 시점에, 자동으로 호출된다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import LifecycleEx from './R004\_LifecycleEx'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <LifecycleEx></LifecycleEx>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R004\_LifecycleEx.js파일을 import해 사용 가능하도록 한다.

10: R004\_LifecycleEx.js에서 작성된 component를 이식한다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 render()함수에서 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**005. 생명주기 함수 constructor(props) 사용하기**

**• 학습 내용 :** 컴포넌트의 생명주기 함수 중, constructor()함수에 대해 이해한다.

**• 힌트 내용 :** 생명주기 함수들이 실행될 때, 로그를 출력하고 실행 순서를 확인한다.

src폴더에 R005\_LifecycleEx.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R005\_LifecycleEx.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18 | import React, { Component } from 'react';  class R005\_LifecycleEx extends Component {  constructor(props) {  super(props);  this.state = {};  console.log('1. constructor Call');  }  render() {  console.log('3. render Call');  return (  <h2>[ THIS IS CONSTRUCTOR FUCNTION ]</h2>  )  }  }  export default R005\_LifecycleEx; |

4-6: constructor(props) 함수는 생명주기 함수 중 가장 먼저 실행되며, 처음 한 번만 호출된다. component내부에서 사용되는 변수(state)를 선언하고, 부모객체에서 전달받은 변수(props)를 초기화할 때 사용한다. super()함수는 가장 위에 호출해야 한다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import LifecycleEx from './R005\_LifecycleEx'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <LifecycleEx></LifecycleEx>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R005LifecycleEx.js파일을 import해 사용 가능하도록 한다.

10: R005LifecycleEx.js에서 작성된 component를 이식한다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 render()함수에서 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**006. 생명주기 함수 static getDerivedStateFormProps(props, state) 사용하기**

**• 학습 내용 :** 컴포넌트의 생명주기 함수 중, getDerivedStateFormProps()함수에 대해 이해한다.

**• 힌트 내용 :** 생명주기 함수들이 실행될 때, 로그를 출력하고 실행 순서를 확인한다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17 | import React from 'react';  import './App.css';  import LifecycleEx from './R006\_LifecycleEx'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <LifecycleEx  prop\_value = 'FromApp.js'  />  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R006\_LifecycleEx.js파일을 import해 사용 가능하도록 한다.

10-12: App.js에서 import한 component인 R006\_LifecycleEx로 prop\_value라는 변수를 전달한다.

src폴더에 R006\_LifecycleEx.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R006\_LifecycleEx.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import React, { Component } from 'react';  class R006\_LifecycleEx extends Component {  static getDerivedStateFromProps(props, state) {  console.log('2. getDerivedStateFromProps Call :'+props.prop\_value);  return {};  }  constructor(props) {  super(props);  this.state = {};  console.log('1. constructor Call');  }  render() {  console.log('3. render Call');  return (  <h2>[ THIS IS CONSTRUCTOR FUCNTION ]</h2>  )  }  }  export default R006\_LifecycleEx; |

4-6: getDerivedStateFromProps(props, state)함수는 constructor()함수 다음으로 실행된다. 컴포넌트가 새로운 props를 받게됐을 때, state을 변경해준다. App.js에서 전달한 prop\_value라는 변수를 props.prop\_value로 접근해 값을 가져올 수 있다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**007. 생명주기 함수 componentDidMount() 사용하기**

**• 학습 내용 :** 컴포넌트의 생명주기 함수 중, componentDidMount()함수에 대해 이해한다.

**• 힌트 내용 :** 생명주기 함수들이 실행될 때, 로그를 출력하고 실행 순서를 확인한다.

App.js파일에서 LifecycleEx를 import 하는 부분을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3 | ..  import LifecycleEx from './R007\_LifecycleEx'  .. |

3: App.js파일과 같은 경로에 위치한 R007\_LifecycleEx.js파일을 import해 사용 가능하도록 한다.

src폴더에 R007\_LifecycleEx.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R007\_LifecycleEx.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import React, { Component } from 'react';  class R007\_LifecycleEx extends Component {  static getDerivedStateFromProps(props, state) {  console.log('2. getDerivedStateFromProps Call :'+props.prop\_value);  return {tmp\_state:props.prop\_value};  }  constructor(props) {  super(props);  this.state = {};  console.log('1. constructor Call');  }  componentDidMount() {  console.log('4. componentDidMount Call');  console.log('5. tmp\_state : '+this.state.tmp\_state);  }  render() {  console.log('3. render Call');  return (  <h2>[ THIS IS COMPONENTDIDMOUNT FUCNTION ]</h2>  )  }  }  export default R007\_LifecycleEx; |

4-6: componentDidMount()함수는 작성한 함수들 중, 가장 마지막으로 실행된다. render()함수가 return되는 html형식의 코드를 화면에 그려준 다음 실행된다. 화면이 모두 그려진 후에 실행되어야 하는 이벤트 처리, 초기화 등 가장 많이 활용되는 함수이다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**008. 생명주기 함수 shouldComponedntUpdate() 사용하기**

**• 학습 내용 :** 컴포넌트의 생명주기 함수 중, shouldComponedntUpdate()함수에 대해 이해한다.

**• 힌트 내용 :** 생명주기 함수들이 실행될 때, 로그를 출력하고 실행 순서를 확인한다.

react에서 생명주기란 component의 생성 > 변경 > 소멸의 과정을 뜻한다. shouldComponedntUpdate() 함수는 component의 '변경' 과정에 속하는 함수이다. 여기서 '변경'이란 props나 state의 변경을 말한다.

App.js파일에서 LifecycleEx를 import 하는 부분을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3 | ..  import LifecycleEx from './R008\_LifecycleEx'  .. |

2: App.js파일과 같은 경로에 위치한 R008\_LifecycleEx.js파일을 import해 사용 가능하도록 한다.

src폴더에 R008\_LifecycleEx.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R008\_LifecycleEx.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import React, { Component } from 'react';  class R008\_LifecycleEx extends Component {  static getDerivedStateFromProps(props, state) {  console.log('2. getDerivedStateFromProps Call :'+props.prop\_value);  return {tmp\_state:props.prop\_value};  }  constructor(props) {  super(props);  this.state = {};  console.log('1. constructor Call');  }  componentDidMount() {  console.log('4. componentDidMount Call');  console.log('5. tmp\_state : '+this.state.tmp\_state);  this.setState({tmp\_state2 : true});  }  shouldComponentUpdate(props, state) {  console.log('6. shouldComponentUpdate Call / tmp\_state2 = ' + state.tmp\_state2);  return state.tmp\_state2  }  render() {  console.log('3. render Call');  return (  <h2>[ THIS IS shouldComponentUpdate FUCNTION ]</h2>  )  }  }  export default R008\_LifecycleEx; |

15: componentDidMount()함수는 '생성'단계의 생명주기 함수 중, 가장 마지막으로 실행된다.

18: tmp\_state2라는 state변수에 true라는 boolean타입의 데이터를 세팅했다. setState()함수는 변수의 선언과 초기화를 동시에 실행한다.

21-23: 18line에서 state의 변경이 발생했기 때문에, '변경' 단계의 생명주기 함수 shouldComponentUpdate()가 실행된다. shouldComponentUpdate()는 boolean타입의 데이터를 반환하는데, return값이 true일 경우에 render()함수를 한 번 더 호출한다.

|  |
| --- |
| NOTE |
| shouldComponentUpdate()함수의 반환 값에 따라 render()함수를 재실행할 수 있다는 점을 이용하면, props나 state변수가 변경될 때 화면을 다시 그리며 제어할 수 있다. |

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**009. 템플릿 문자열 사용하기**

**• 학습 내용 :** ES6문자열 사용방법에 대해 이해한다.

**• 힌트 내용 :** 기존 자바스크립트 문자열과 비교하고 추가된 함수를 확인한다.

ES(ECMA스크립트)는 표준화된 스크립트 언어인데, ES뒤에 붙은 숫자는 버전을 의미한다. 2011년 부터 사용된 ES5가 웹 표준처럼 사용되고 있다. 2015년 발행된 ES6는 많은 유용한 기능들이 추가되었고, 자바스크립트는 이 기술규격을 따른다. react도 자바스크립트 기반의 언어이기 때문에 ES6의 모든 기능들을 사용할 수 있다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import Es6 from './R009\_Es6'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <Es6/>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R009\_Es6.js파일을 import해 사용 가능하도록 한다.

src폴더에 R009\_Es6.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R009\_Es6.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import React, { Component } from 'react';  class R009\_Es6 extends Component {  constructor(props) {  super(props);  this.state = {};  }  componentDidMount() {  var jsString1 = '자바스크립트'  var jsString2 = '입니다\n다음줄입니다.'  console.log(jsString1+' 문자열'+jsString2+'~');    var Es6String1 = 'ES6'  var Es6String2 = '입니다'  console.log(`${Es6String1} 문자열${Es6String2}!!  \_\_\_\_다음줄입니다`);  var LongString = "ES6에추가된String함수들입니다.";  console.log('startsWith : '+LongString.startsWith("ES6에추"));  console.log('endsWith : '+LongString.endsWith("함수들입니다."));  console.log('includes : '+LongString.includes("추가된String"));  }  render() {  return (  <h2>[ THIS IS ES6 STRING ]</h2>  )  }  }  export default R009\_Es6; |

12: 기존 자바스크립트에서는 줄바꿈을 하려면 개행문자(＼n)를 사용해야 한다.

13: 문자열과 변수를 합치기 위해서는 작은 따옴표(또는 큰따옴표)로 문자열을 감싸고, +로 연결해야 한다.

17-18: 따옴표가 아닌 백틱(`)으로 전체 문자열과 변수를 묶어서 사용한다. 변수는 ${변수명}형태로 넣고, 코드상에서 줄바꿈을 하면 개행문자 없이도 사용할 수 있다.

21-23: startsWith(), endsWith(), includes()는 ES6에 추가된 String함수들이다. startsWith()는 변수 앞에서 부터, endsWith()는 뒤에서 부터 일치하는 문자열이 있는지 찾는다. includes()는 위치에 상관없이 변수에 특정문자열이 포함되어 있는지 판단한다. 함수 조건에 부합하면 true, 아니라면 false를 반환한다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**010. var, let, const 사용하기**

**• 학습 내용 :** ES6의 변수 선언 방식인 let과 const에 대해서 이해한다.

**• 힌트 내용 :** 기존 var변수와 비교해 변수의 재선언, 재할당이 가능한지 확인한다.

ES5에서 사용하던 var는 유연한 방식으로 변수를 재선언, 재할당 할 수 있다. 이런 특징으로 인해 변수의 사용범위가 불확실해 지거나, 의도하지 않은 변수 값 변경이 발생할 수 있다. 이러한 var의 단점을 보완하기 위에 ES6에서 let과 const가 추가됐다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import Variable from './R010\_Variable'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <Variable/>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R010\_Variable.js파일을 import해 사용 가능하도록 한다.

src폴더에 R010\_Variable.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R010\_Variable.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import React, { Component } from 'react';  class R010\_Variable extends Component {  constructor(props) {  super(props);  this.state = {};  }  componentDidMount() {  var varName = 'react'  console.log('varName1 : '+varName)  var varName = '200' // 'varName' is already defined no-redeclare  console.log('varName2 : '+varName)  let letName = 'react'  console.log('letName1 : '+letName)  // let letName = '200' // Parsing error: Identifier 'letName' has already been declared  letName = 'react200'  console.log('letName2 : '+letName)  const constName = 'react'  console.log('constName : ' + constName)  // const constName = '200' // Parsing error: Identifier 'constName' has already been declared  // constName = 'react200' // Uncaught TypeError: Assignment to constant variable.  }  render() {  return (  <h2>[ THIS IS Variable ]</h2>  )  }  }  export default R010\_Variable; |

13: 이미 선언한 var변수 varName를 다시 선언했을 때, 'varName' is already defined no-redeclare라는 경고 메시지가 콘솔 로그에 출력된다. 하지만 var변수는 재선언, 재할당을 허용하기 때문에 경고 메시지가 출력되어도, 정상적으로 페이지가 노출된다.

18: 이미 선언한 let변수 letName를 다시 선언했을 때, Parsing error: Identifier 'letName' has already been declared라는 에러 메시지가 콘솔 로그에 출력된다. let변수는 재선언을 허용하지 않기 때문에 오류페이지가 노출된다.

19: let변수는 재할당을 허용한다. 이미 선언한 let변수 letName에 새로운 값을 할당했을 때, 정상적으로 페이지가 노출된다.

24: 이미 선언한 const변수 constName를 다시 선언했을 때, Parsing error: Identifier 'constName' has already been declared라는 에러 메시지가 콘솔 로그에 출력된다. const변수는 재선언을 허용하지 않기 때문에 오류페이지가 노출된다.

25: 이미 선언한 const변수 constName에 새로운 값을 할당했을 때, Uncaught TypeError: Assignment to constant variable라는 에러 메시지가 콘솔 로그에 출력된다. const변수는 재할당을 허용하지 않기 때문에 오류페이지가 노출된다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**011. 전개 연산자 사용하기**

**• 학습 내용 :** 전개 연산자를 통해 배열과 객체의 데이터를 합치거나 추출하는 방법을 이해한다.

**• 힌트 내용 :** 기존 ES5 문법과 비교해, ES6 전개 연산자의 장점과 사용법을 확인한다.

전개 연산자는 배열이나 객체 변수를 좀 더 직관적이고 편리하게, 합치거나 추출할 수 있게 도와주는 문법이다. 변수 앞에 …(마침표 3개)를 입력해 사용한다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import SpreadOperator from './R011\_SpreadOperator'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <SpreadOperator/>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R011\_SpreadOperator.js파일을 import해 사용 가능하도록 한다.

src폴더에 R011\_SpreadOperator.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R011\_SpreadOperator.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import React, { Component } from 'react';  class R011\_SpreadOperator extends Component {  constructor(props) {  super(props);  this.state = {};  }  componentDidMount() {  //javascript Array  var varArray1 = ['num1', 'num2'];  var varArray2 = ['num3', 'num4'];  var sumVarArr = [varArray1[0], varArray1[1], varArray2[0], varArray2[1]];  // var sumVarArr = [].concat(varArray1, varArray2);  console.log('1. sumVarArr : '+sumVarArr)  //ES6 Array  let sumLetArr = [...varArray1, ...varArray2];  console.log('2. sumLetArr : '+sumLetArr)  const [sum1, sum2, ...remain] = sumLetArr;  console.log('3. sum1 : '+sum1+', sum2 : '+sum2+ ', remain : '+remain)  var varObj1 = { key1 : 'val1', key2 : 'val2' }  var varObj2 = { key2 : 'new2', key3 : 'val3' }  //javascript Object  var sumVarObj = Object.assign({}, varObj1, varObj2)  console.log('4. sumVarObj : '+JSON.stringify(sumVarObj))  //ES6 Object  var sumLetObj = {...varObj1, ...varObj2}  console.log('5. sumLetObj : '+JSON.stringify(sumLetObj))  var {key1, key3, ...others} = sumLetObj;  console.log('6. key1 : '+key1+', key3 : '+key3+', others : '+JSON.stringify(others));  }  render() {  return (  <h2>[ THIS IS SpreadOperator ]</h2>  )  }  }  export default R011\_SpreadOperator; |

14-15: 기존 ES5에서 배열 두 개를 합치기 위해서는, 배열 각각에 인덱스로 접근에 값을 가져오거나 concat함수를 이용한다. varArray1, varArray2배열에 각각 인덱스(0, 1)로 접근해, 인자 값(num1, num2, num3, num4)을 가져와 새로운 배열 sumVarArr에 하나씩 넣는다.

18: ES6에서는 전개연산자 …(마침표 3개)을 배열 명 앞에 붙여, 여러 개의 배열을 합칠 수 있다.

20: sumLetArr배열의 값을 추출에 개별 변수에 넣는다. 순서대로 변수 sum1에 sumLetArr[0]값을, sum2에 sumLetArr[1]값을 대입한다. 나머지 배열 값은 마지막에 전개 연산자 처리된 …remain변수에 넣는다.

26: 기존 ES5에서 객체 두 개를 합치기 위해서는, Object.assign()함수를 이용한다. 첫 번째 인자 {}는 함수의 return값이고, 뒤에 인자에 객체들을 ,(콤마)로 연결해 나열하면 여러 개 객체를 합칠 수 있다.

29: ES6에서는 …(마침표 3개)을 객체 명 앞에 붙여, 여러 개의 객체를 합칠 수 있다.

31: sumLetObj객체의 키와 값을 추출에, 키와 동일한 명칭의 개별 변수에 넣는다. 나머지는 마지막에 전개 연산자 처리된 …others변수에 넣는다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**012. class 사용하기**

**• 학습 내용 :** 객체단위로 코드 재사용을 위해 사용하는 class에 대해 이해한다.

**• 힌트 내용 :** 기존 ES5의 prototype과 비교해, ES6 class의 장점과 사용법을 확인한다.

기존 ES5 자바스크립트에서는 객체를 구현하기 위해, prototype을 사용한다. 객체는 상속을 통해 코드를 재사용할 수 있게 해준다. ES6에서 등장한 class는 prototype과 같은 개념인데, 읽고 표현하는 것을 쉽게 하기 위해서 고안된 문법이다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import ClassPrototype from './R012\_Class&Prototype'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <ClassPrototype/>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R012\_Class&Prototype.js파일을 import해 사용 가능하도록 한다.

src폴더에 R012\_Class&Prototype.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R012\_Class&Prototype.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | import React, { Component } from 'react';  class ClassPrototype extends Component {  constructor(props) {  super(props);  this.state = {};  }  componentDidMount() {  //ES5 prototype  var ExamCountFunc = (function () {  function ExamCount(num) {  this.number = num;  }  ExamCount.prototype.showNum = function () {  console.log('1. react\_' + this.number);  };  return ExamCount;  }());    var cnt = new ExamCountFunc('200');  cnt.showNum();  //ES6 class  class ExamCountClass {  constructor(num2) {  this.number2 = num2;  }  showNum() {  console.log(`2. react\_${this.number2}`);  }  }    var cnt2 = new ExamCountClass('2hundred');  cnt2.showNum();  }  render() {  return (  <h2>[ THIS IS Class ]</h2>  )  }  }  export default ClassPrototype; |

|  |
| --- |
| NOTE |
| 자바스크립트에서는 함수를 객체로 사용할 수 있다. |

22: ExamCountFunc()함수(객체)를 실행하고 return되는 결과 값(객체)을 cnt라는 변수에 저장한다

13-15: 생성자 함수를 실행하는데, 파라미터로 전달받은 num변수의 값(200)을 객체변수 number에 저장한다.

23: 객체 안에 선언된 showNum()함수를 실행한다.

16: 생성자 함수명.prototype.함수명 형태로 선언을 해주면, 객체 외부에서 함수(cnt.showNum();)를 실행해 객체내부에 선언된 함수를 사용할 수 있다. 함수가 실행되면, 생성자 함수에서 ‘200’으로 할당된 객체변수 number를 사용한다.

35: ExamCountClass객체를 생성하고, 객체를 cnt2라는 변수에 저장한다.

26: ES6에서는 class로 객체를 선언한다.

27-29: constructor()라는 생성자 함수가 실행되고, 파라미터로 전달받은 num2이라는 변수 값(2hundred)을 객체변수 number2에 저장한다.

36: 객체 안에 선언된 showNum()함수를 실행한다.

30-32: 객체에 접근할 때 실행할 함수(showNum)는, class의 {}괄호(scope)안에 간단하게 선언할 수 있다. 함수가 실행되면, 생성자 함수에서 ‘2hundred’으로 할당된 객체변수 number2를 사용한다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |

**013. 화살표 함수 사용**

**• 학습 내용 :** ES6에 추가된 화살표 함수의 사용 방법을 이해한다.

**• 힌트 내용 :** 기존 ES5의 기본 function과 비교해, ES6 화살표 함수의 장점과 사용법을 확인한다.

ES6에서 등장한 화살표 함수는 ‘function’대신 ‘=>’ 문자열을 사용하며, ‘return’ 문자열도 생략할 수 있다. 그래서 기존 ES5 함수보다 간략하게 함수를 선언할 수 있다. 또 화살표 함수에서는 콜백함수에서 this를 bind해야하는 문제도 발생하지 않는다.

App.js파일을 다음과 같이 수정한다.

<소스> App.js

|  |  |
| --- | --- |
| 1 2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React from 'react';  import './App.css';  import ArrowFunc from './R013\_ArrowFunction'  function App() {  return (  <div>  <h1>Start React 200!</h1>  <p>CSS 적용하기</p>  <ArrowFunc/>  </div>  );  }  export default App; |

3: App.js파일과 같은 경로에 위치한 R012\_Class&Prototype.js파일을 import해 사용 가능하도록 한다.

src폴더에 R013\_ArrowFunction.js 파일을 생성하고, 다음과 같이 입력한다.

<소스> R013\_ArrowFunction.js

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | import React, { Component } from 'react';  class R013\_ArrowFunction extends Component {  constructor(props) {  super(props);  this.state = {  arrowFuc: 'React200',  num: 3  };  }  componentDidMount() {  Function1(1);  this.Function2(1,1);  this.Function3(1,3);  this.Function4();  this.Function5(0,2,3);  function Function1(num1) {  return console.log(num1+'. Es5 Function');  }  }    Function2 = (num1, num2) => {  let num3 = num1 + num2;  console.log(num3+'. Arrow Function : '+this.state.arrowFuc);  }    Function3() {  var this\_bind = this;  setTimeout(function() {  console.log(this\_bind.state.num+'. Es5 Callback Function noBind : ');  console.log(this.state.arrowFuc);  },100);  }    Function4() {  setTimeout(function() {  console.log('4. Es5 Callback Function Bind : '+this.state.arrowFuc);  }.bind(this),100);  }    Function5 = (num1, num2, num3) => {  const num4 = num1 + num2 + num3;  setTimeout(() => {  console.log(num4+'. Arrow Callback Function : '+this.state.arrowFuc);  }, 100);  }  render() {  return (  <h2>[ THIS IS ArrowFunction ]</h2>  )  }  }  export default R013\_ArrowFunction; |

14-18: Function1~5까지의 함수를 순서대로 호출한다.

20-22: 함수를 호출할 때, 전달받은 num1이라는 변수를 함수 내부에서 사용할 수 있다.

25-28: ‘function’ 문자열 대신 ‘=>’로 함수를 선언했다. 함수 내에서 사용하는 this는 R013\_ArrowFunction컴포넌트인데, this로 컴포넌트의 state변수에 접근해 사용할 수 있다.

31-33: 콜백함수 내부에서는 this로 컴포넌트에 접근할 수 없기 때문에, 접근할 수 있는 변수에 this를 백업한다. 백업된 변수 this\_bind를 이용해 컴포넌트의 state변수에 접근할 수 있다.

34: 콜백함수 내부에서 this는 window객체이기 때문에, this로 state변수에 접근하면 undefined오류가 발생한다.

40-41: 콜백함수에 함수 밖의 this를 bind해주면, this를 컴포넌트로 사용할 수 있다.

47: 화살표 함수에서는 this를 bind해주지 않아도, this를 컴포넌트로 사용해 state변수에 접근할 수 있다.

크롬 브라우저에서 [F12]를 눌러 개발자도구 창을 띄운 다음, Console탭을 클릭하면 출력된 로그를 확인할 수 있다.

<실행 결과>

|  |
| --- |
|  |