

• Typescript part 2



1. 제네릭

❖ 제네릭 (Generic)이란?

- 직역 : 포괄적인, 일반 명칭인
 - 특정 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있도록 하는 방법'이다.
- 함수, 클래스를 사용할 때 사용할 타입을 지정하여 호출하는 프로그래밍 기법

❖ Generic을 사용하지 않는 경우

- 함수의 인자로 여러 유형을 전달하려면? --> any?

```
function arrayConcat(items1:any[], items2 : any[] ) : any[] {  
    return items1.concat(items2);  
}
```

```
//any 타입을 사용하면 아무 타입이나 사용할 수 있지만 타입을 일관되게 사용할 수 없음  
let arr1 = arrayConcat([10,20,30], ['a','b', 40])  
arr1.push(true);
```

- 그렇다면 해결책은?
 - 함수인 경우는 함수 오버로딩
 - 그리고 제네릭

1. 제네릭

❖ 함수 오버로딩

- 가능하지만 여러 타입을 지원하려면 코드량이 많아짐
- 컴파일러 수준에서는 오류를 잡아주지만 실제로는 실행되어 버림

```
function arrayConcat2(items1:number[], items2 : number[] ) : number[];  
function arrayConcat2(items1:string[], items2 : string[] ) : string[];  
function arrayConcat2(items1:any[], items2 : any[] ) : any[] {  
    return items1.concat(items2);  
}
```

```
let arr21 = arrayConcat2([10,20,30], [40,50])  
console.log(arr21)  
let arr22 = arrayConcat2(['a','b','c'], ['d','e'])  
console.log(arr22)  
let arr23 = arrayConcat2([10,20,30], ['d','e'])  
console.log(arr23)
```

The screenshot shows the TypeScript Playground interface. The code editor contains the same code as the previous blocks. A red squiggly line under the call to `arrayConcat2([10, 20, 30], ['d', 'e'])` indicates an error. The error message in the center pane reads: "No overload matches this call. Overload 1 of 2, '(items1: number[], items2: number[]): number[]', gave the following error. Type 'string' is not assignable to type 'number'. Overload 1 of 2, '(items1: number[], items2: number[]): number[]', gave the following error. Type 'string' is not assignable to type 'number'. (2769)". Below this, it says "input.tsx(3, 10): The call would have succeeded against this implementation, but implementation signatures of overloads are not externally visible." and "View Problem (Alt+F8) No quick fixes available". On the right, the "Errors" tab shows one error, and the "Logs" tab shows the output of the console logs: `[LOG]: [10, 20, 30, 40, 50]`, `[LOG]: ["a", "b", "c", "d", "e"]`, and `[LOG]: [10, 20, 30, "d", "e"]`.

1. 제네릭

❖Generic 적용

- 함수 오버로딩과 마찬가지로 컴파일러 수준에서는 오류를 잡아주지만 실제로는 실행되어 버림

```
function arrayConcat3<T>(items1 : T[], items2:T[] ) : T[] {  
    return items1.concat(items2);  
}
```

```
let arr31 = arrayConcat3<number>([10,20,30], [40,50])  
console.log(arr31);  
let arr32 = arrayConcat3<string>(['a','b','c'], ['d','e'])  
console.log(arr32);  
let arr33 = arrayConcat3<string>([10,20,30], ['d','e'])  
console.log(arr33);
```

The screenshot shows the VS Code editor interface. The editor has tabs for 'v5.3.3', 'Run', 'Export', and 'Share'. The code in the editor is as follows:

```
1 //----제네릭을 사용하면? 일관된 타입 사용  
2 function arrayConcat3<T>(items1 : T[], items2:T[] ) : T[] {  
3     | return items1.concat(items2);  
4 }  
5  
6 let arr31 = arrayConcat3<number>([10,20,30], [40,50])  
7 console.log(arr31);  
8 let arr32 = arrayConcat3<string>(['a', 'b', 'c'], ['d', 'e'])  
9 console.log(arr32);  
10 let arr33 = arrayConcat3<string>([10,20,30], ['d', 'e'])  
11 console.log(arr33);
```

On the right side, there is a panel with tabs for '.JS', '.D.TS', 'Errors' (with a red circle containing the number 3), 'Logs', and 'Plugins'. The 'Logs' tab is selected, showing the following log entries:

```
[LOG]: [10, 20, 30, 40, 50]  
-----  
[LOG]: ["a", "b", "c", "d", "e"]  
-----  
[LOG]: [10, 20, 30, "d", "e"]
```

A tooltip is visible over the code on line 10, stating: 'Type 'number' is not assignable to type 'string'. (2322)'. Below the tooltip, there are links for 'View Problem (Alt+F8)' and 'No quick fixes available'.

1. 제네릭

❖ React에서의 Generic 사용 예제

- 중점적으로 살펴볼 부분
 - useState
 - axios
 - Custom hook
- 예제
 - 시작 예제 : react-generic-1
 - 작성할 부분 :
 - Custom hook : useFetch.ts
 - Component : App.tsx
 - useFetch.ts
 - 사용자 정의 훅으로 axios를 이용한 백엔드 API와의 비동기 통신을 담당. 통신 과정에서의 상태를 처리하는 기능도 함께 수행함.
 - 처리데이터 : 응답데이터, spinner UI를 보여주기 위한 loading 상태, 에러 발생시의 에러 정보, 요청을 백엔드로 전송하는 함수
 - App.tsx
 - Custom hook(useFetch.ts)을 이용해 백엔드와 통신하는 기능을 수행하는 컴포넌트

1. 제네릭

❖src/hooks/useFetch.ts

■ 전체 윤곽

```
const useFetch = <T>(url:string, axiosParams: AxiosRequestConfig) => {  
  const [response, setResponse] = useState<AxiosResponse<T>>>();  
  const [error, setError] = useState<AxiosError>();  
  const [isLoading, setIsLoading] = useState<boolean>(false);  
  
  return { response, error, isLoading };  
};  
  
export { useFetch };
```

■ 이어서 완성

```
import { useState } from "react";  
import axios, { AxiosError, AxiosResponse, AxiosRequestConfig } from "axios";  
  
axios.defaults.baseURL = "/api";  
  
const useFetch = <T>(url:string, params: AxiosRequestConfig) => {  
  const [response, setResponse] = useState<AxiosResponse<T>>>();  
  const [error, setError] = useState<AxiosError>();  
  const [isLoading, setIsLoading] = useState<boolean>(false);
```

1. 제네릭

❖src/hooks/useFetch.ts

■ 이어서 완성

```
const fetchData = async () => {  
  setResponse(undefined);  
  try {  
    setIsLoading(true);  
    const result: AxiosResponse<T> = await axios.get<T>(url, params);  
    setResponse(result);  
  } catch (err) {  
    setError(err as unknown as AxiosError);  
  } finally {  
    setIsLoading(false);  
  }  
};  
  
const requestData = () => {  
  fetchData();  
};  
  
return { response, error, isLoading, requestData };  
};  
  
export { useFetch };
```


1. 제네릭

❖src/App.tsx

```
import { useEffect, useRef, useState } from "react";
import { useFetch } from "../hooks/useFetch";
import { ReactCspin } from "react-cssspin";
import 'react-cssspin/dist/style.css';

type TodoItemType = {
  id:number; todo:string; desc:string; done:boolean;
}

const App = () => {
  const [owner, setOwner] = useState<string>("mrlee");
  const refOnwer = useRef<HTMLInputElement|null>(null);
  const { response, isLoading, error, requestData } = useFetch<TodoItemType[]>(`/todolist_long/${owner}`, { timeout: 5000 });

  const setOnwerHandler = () => {
    let newOwner = refOnwer.current?.value;
    if (newOwner) {
      setOwner(newOwner);
    }
  }

  useEffect(()=>{
    requestData();
  }, [owner])
```

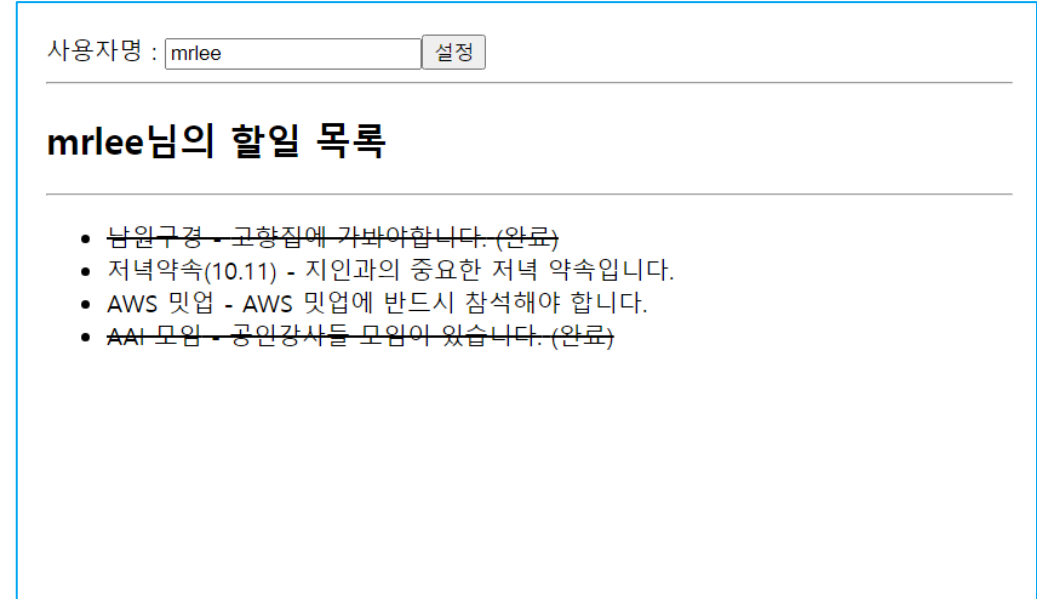
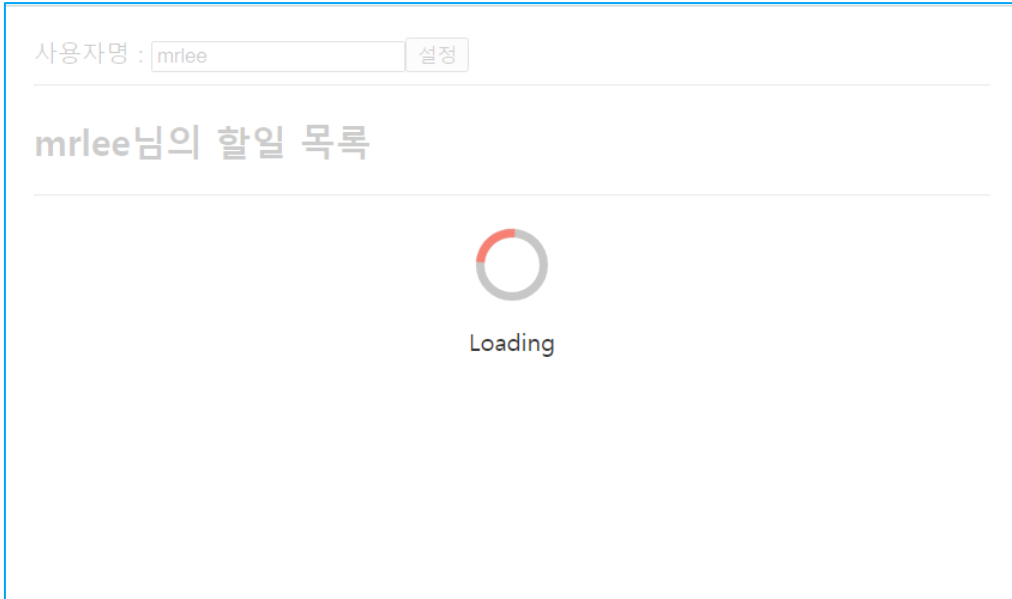

1. 제네릭

❖src/App.tsx (이어서)

```
return (  
  <div>  
    사용자명 : <input type="text" defaultValue={owner} ref={refOnwer} />  
    <button onClick={setOnwerHandler}>설정</button>  
    <hr />  
    <h2>{owner}님의 할일 목록</h2>  
    <hr />  
    <ul>  
      {  
        error ? <div><h3>에러 발생 : {error.message}</h3></div> :  
        response?.data.map((todoItem:TodoItemType)=>{  
          return (  
            <li key={todoItem.id} style={ todoItem.done ? { textDecoration:"line-through"} : {}}>  
              {todoItem.todo} - {todoItem.desc}{ " " }  
              {todoItem.done ? "(완료)" : "" }  
            </li>  
          )  
        })  
      }  
      { isLoading ? <ReactCspin opacity={0.8} /> : ""}  
    </ul>  
  </div>  
)  
};
```

1. 제네릭

❖ 실행 결과



2. 타입 이동

❖타입 이동(Moving Type)이란?

- 기존 변수의 타입을 다른 변수의 타입으로 사용하는 것
- 한 타입을 변경하면 이동된 타입들도 모두 업데이트됨.

❖간단한 예시

```
//기존 변수의 타입을 이동
const a1: number = 1004;
let a2: typeof a1;           //a1 변수의 타입을 a2 타입으로 이동
a2 = 1101; //ok
a2 = 'hello'; //fail

//클래스 멤버의 타입을 이동
class Person {
  constructor(public name:string) {}
}

let name2: Person['name']; //name2의 타입은 Person의 name 멤버의 타입과 동일함
name2 = "홍길동";          //ok
name2 = 123;               //fail
```

2. 타입 이동

❖상수 타입의 이동

```
// 상수의 값, 타입 이동
const ADDTODO = "addTodo";

let COPYTYPE1 : typeof ADDTODO;
let COPYTYPE2 : typeof ADDTODO;

COPYTYPE1 = "addTodo";    //정상
COPYTYPE2 = "deleteTodo"; //오류 발생
```

```
1 // 상수의 값, 타입 이동
2 const ADDTODO = "addTodo";
3
4 Type '"deleteTodo"' is not assignable to type '"addTodo"'. (2322)
5 let COPYTYPE2: "addTodo"
6 View Problem (Alt+F8) No quick fixes available
7
8 COPYTYPE2 = "deleteTodo";
```

2. 타입 이동

❖ 클래스 타입의 이동

- 클래스 타입은 `typeof` 로 타입을 이동할 수 없음
 - `typeof Person ---> "function"`
- 모듈간 참조 : 단순히 `export --> import` 하여 사용함
- 모듈 내 참조 : `namespace`로 묶어서 참조

```
//-----서로 다른 파일에서의 참조
//Person.ts
class Person {
  constructor(public name:string) {}
}
export { Person };

//-----
import { Person } from "./Person";

let p1 : Person = new Person("홍길동");
console.log(p1);
```

```
//-----같은 파일에서의 참조
namespace PersonNS {
  export class Person {
    constructor(public name: string) { }
  }
}

import Person = PersonNS.Person;

let p1: Person = new Person("홍길동");
console.log(p1);
```

3. 유틸리티 타입

❖ 유틸리티 타입이란?

- 타입을 쉽게 변환할 수 있도록 지원하는 전역으로 사용할 수 있는 특수한 타입
- 유틸리티 타입의 종류 : 많구나!!
 - Partial<Type> Required<Type>
 - Readonly<Type>
 - Record<Keys, Type>
 - Pick<Type, Keys> Omit<Type, Keys>
 - Exclude<Type, ExcludedUnion>
 - Extract<Type, Union>
 - NonNullable<Type>
 - Parameters<Type>
 - ReturnType<Type>
 - InstanceType<Type>
 - UpperCase<StringType> LowerCase<StringType>
 - Capitalize<StringType> Uncapitalize<StringType>

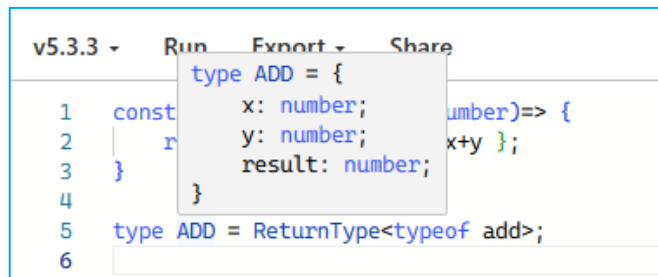
3. 유틸리티 타입

❖ ReturnType<Type>

- 함수 Type의 반환 타입으로 구성된 타입을 생성함

```
const add = (x:number, y:number)=> {  
  return { x, y, result:x+y };  
}
```

```
type ADD = ReturnType<typeof add>;
```



The screenshot shows a code editor with the following code:

```
v5.3.3  Run  Export  Share  
1  const add = (x:number, y:number)=> {  
2    return { x, y, result:x+y };  
3  }  
4  
5  type ADD = ReturnType<typeof add>;  
6
```

A tooltip is visible over the `ReturnType` function, showing its signature: `type ADD = { x: number; y: number; result: number; }`.

```
const addTodo = (todo:string, desc:string) => {  
  return { type:"ADDTODO", payload : { todo, desc } };  
}  
const deleteTodo = (id: number) => {  
  return { type:"DELETETODO", payload : { id } };  
}
```

```
type TODOACTION_TYPE =  
  | ReturnType<typeof addTodo>  
  | ReturnType<typeof deleteTodo>;
```



The screenshot shows a code editor with the following code:

```
v5.3.3  Run  Export  Share  
1  type TODOACTION_TYPE = {  
2    type: string;  
3    payload: {  
4      todo: string;  
5      desc: string;  
6    };  
7  } | {  
8    type: string;  
9    payload: {  
10     id: number;  
11   };  
12 }  
13 type TODOACTION_TYPE =  
14   | ReturnType<typeof addTodo>  
15   | ReturnType<typeof deleteTodo>;  
16
```

A tooltip is visible over the `TODOACTION_TYPE` type definition, showing its structure: `type TODOACTION_TYPE = { type: string; payload: { todo: string; desc: string; }; } | { type: string; payload: { id: number; }; }`.

3. 유틸리티 타입

❖ Required<Type>

- Type의 모든 선택적 속성을 필수 속성으로 설정한 타입을 생성함

❖ Partial<Type>

- Type의 모든 속성을 선택적 속성으로 설정한 타입을 생성함

```
interface Friend {  
  name: string;  
  phone: string;  
  email?: string;  
  address?: string;  
}
```

```
type RequiredFriend = Required<Friend>;  
type PartialFriend = Partial<Friend>;
```

```
type RequiredFriend = {  
  name: string;  
  phone: string;  
  email: string;  
  address: string;  
}
```

```
type PartialFriend = {  
  name?: string | undefined;  
  phone?: string | undefined;  
  email?: string | undefined;  
  address?: string | undefined;  
}
```

3. 유틸리티 타입

❖ Readonly<Type>

- Type의 모든 속성을 Readonly로 설정한 타입을 생성함

```
interface Friend {  
  name: string;  
  phone: string;  
  email: string;  
}  
type ROFriend = Readonly<Friend>;
```

```
type ROFriend = {  
  readonly name: string;  
  readonly phone: string;  
  readonly email: string;  
}
```

❖ ParametersType<Type>

- 함수의 파라미터로 사용된 타입을 이용해 Tuple 타입을 생성함

```
const add = (x:number, y:number) => {  
  return x+y;  
}  
  
type ParameterType = Parameters<typeof add>;
```

```
1  const add = (x:number, y:number) => {  
2    return x+y;  
3  }  
4    type ParameterType = [x: number, y: number]  
5  type ParameterType = Parameters<typeof add>;
```

3. 유틸리티 타입

❖ Pick<Type, Keys> Omit<Type, Keys>

- Pick: Type에서 Keys의 집합에 해당하는 속성으로 타입을 생성함
- Omit: Type에서 Keys의 집합에 해당하는 속성을 제거한 타입을 생성함

```
interface Friend {  
  name: string;  
  phone: string;  
  email: string;  
  address: string;  
}  
type PickFriend = Pick<Friend, "name"|"phone">;  
type OmitFriend = Omit<Friend, "address"|"phone">;
```

```
type PickFriend = {  
  name: string;  
  phone: string;  
}
```

```
type OmitFriend = {  
  name: string;  
  email: string;  
}
```

3. 유틸리티 타입

❖Parameters<Type>

- Type이 함수일 때 사용
- Type 함수의 매개변수들의 타입으로 새로운 튜플 타입을 생성함

```
const add = (x:number, y:number) => {  
    return x+y;  
}  
  
type ParameterType = Parameters<typeof add>;
```

```
1  const add = (x:number, y:number) => {  
2      return x+y;  
3  }  
4      type ParameterType = [x: number, y: number]  
5  type ParameterType = Parameters<typeof add>;
```

3. 유틸리티 타입

❖내장 문자열 조작 타입

- 타입 리터럴 문자열에서의 문자열 조작을 위한 유틸리티 타입
- Uppercase<StringType>
- Lowercase<StringType>
- Capitalize<StringType>
- Uncapitalize<StringType>

```
type Code = "Apple"
type UpperCode = Uppercase<Code>      //APPLE
type LowerCode = Lowercase<Code>      //apple

type Code2 = "mongodb";
type CapitalCode2 = Capitalize<Code2>  //Mongodb

type Code3 = "AMAZON WORLD";
type UncapitalCode3 = Uncapitalize<Code3> //aAMAZON WORLD
```

4. 타입스크립트 기반 React 컴포넌트 작성

- ❖ Typescript 기반 리액트 프로젝트 생성
- ❖ 함수 컴포넌트
- ❖ 클래스 컴포넌트
- ❖ 컴포넌트 속성의 유효성 검사

4.1 Typescript 기반 리액트 프로젝트 생성

❖ Vite

- `npm init vite 프로젝트명 -- --template react-ts`
- `npm create vite 프로젝트명 -- --template react-ts`

❖ CRA

- `npx create-react-app 프로젝트명 --template typescript`

❖ tsconfig.json(vite 예시)

```
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "skipLibCheck": true,

    /* Bundler mode */
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "resolveJsonModule": true,
```

```
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx",

    /* Linting */
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": ["src"],
  "references": [{ "path": "./tsconfig.node.json" }]
}
```


4.2 함수 컴포넌트

❖ 함수 컴포넌트 형태

⟨속성을 사용하는 컴포넌트⟩

```
//속성 타입 : type또는 interface
interface IProps {
  name: string;
  visited: Boolean;
}

const CountryItem = (props: IProps) => {
  return (
    <h2>
      {props.name} {props.visited ? "(방문)" : ""}
    </h2>
  );
};

export default CountryItem;
```

⟨속성을 사용하지 않는 컴포넌트⟩

```
import { useState } from 'react'
import CountryItem from './CountryItem';

const App = () => {
  const [name, setName] = useState<string>("");
  const [visited, setVisited] = useState<boolean>(false);

  return (
    <div>
      나라이름 : <input type="text" value={name}
        onChange={(e)=>setName(e.target.value)} /><br />
      방문여부 : <input type="checkbox" checked={visited}
        onChange={()=>setVisited(!visited)} /> <br />
      <hr />
      <CountryItem name={name} visited={visited} />
    </div>
  )
}

export default App
```

4.3 클래스 컴포넌트

❖클래스 컴포넌트 형태

■ 속성만을 사용하는 컴포넌트

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

type Props = {
  name: string;
  visited: Boolean;
};

class CountryItem extends Component<Props> {

  render() {
    return (
      <h2>
        {this.props.name}
        {this.props.visited ? "(방문)" : ""}
      </h2>
    );
  }
}

export default CountryItem;
```

```
import { Component } from "react";
import CountryItem from "../CountryItem";

type State = { name: string; visited: boolean };

class App extends Component<undefined, State> {
  state = { name: "", visited: false };

  render() {
    return (
      <div>
        나라이름 : <input type="text" value={this.state.name}
          onChange={(e) => this.setState({ name: e.target.value })}
        />
        <br />
        방문여부 : <input type="checkbox"
          checked={this.state.visited}
          onChange={() => this.setState({ visited:
            !this.state.visited })}
        />{" "}
        <br /><hr />
        <CountryItem name={this.state.name}
          visited={this.state.visited} />
      </div>
    );
  }
}
```

4.3 클래스 컴포넌트

❖클래스 컴포넌트 형태

- 상태를 사용하는 컴포넌트

```
import { Component } from "react";
import CountryItem from "../CountryItem";

type State = { name: string; visited: boolean };

class App extends Component<undefined, State> {
  state = { name: "", visited: false };
  render() {
    return (
      <div>
        나라이름 : <input type="text" value={this.state.name}
          onChange={(e) => this.setState({ name: e.target.value })}
        />
        <br />
        방문여부 : <input type="checkbox" checked={this.state.visited}
          onChange={() => this.setState({ visited: !this.state.visited })}
        />{" "}
        <br /><hr />
        <CountryItem name={this.state.name} visited={this.state.visited} />
      </div>
    );
  }
}
export default App;
```

4.4 컴포넌트 속성의 유효성 검사

❖속성의 유효성 검증

- 컴포넌트 기반으로 개발할 때 컴포넌트의 속성은 다음을 쉽게 식별할 수 있어야 함.
 - 컴포넌트에서 사용가능한 속성이 무엇인지...
 - 필수 속성은 무엇인지...
 - 속성에 전달할 수 있는 값의 타입은 무엇인지...
- 이를 위해 속성의 유효성 검사 기능이 필요함.

❖유효성 검증 방법

- Typescript의 정적 타입지원 기능
 - 컴파일(빌드)시에 타입을 검사함
 - IDE를 통해서 Code Intellisense 기능을 지원받을 수 있음
- PropTypes
 - 런타임시에 타입을 검사함
 - 실제로 전달되는 값을 이용해 타입을 확인하고 경고를 일으킴
- 병행 사용할 것을 권장 : Typescript + PropTypes

4.4.1 준비된 예제 확인

❖시작 예제

- 준비된 proptypes-test-1 예제로 시작
 - 미리 제공되는 컴포넌트 : App.tsx, Calc.tsx
- src/Calc.tsx

```
type CalcPropsTypes = {  
  x: number;  
  y: number;  
  oper: string;  
};  
  
const Calc = (props: CalcPropsTypes) => {  
  let result: number = 0;  
  switch (props.oper) {  
    case "+":  
      result = props.x + props.y;  
      break;  
    case "*":  
      result = props.x * props.y;  
      break;  
    default:  
      result = 0;  
  }  
}
```

```
return (  
  <div>  
    <h3>연산 방식 : {props.oper}</h3>  
    <hr />  
    <div>  
      {props.x} {props.oper} {props.y} = {result}  
    </div>  
  </div>  
)  
);  
};  
  
export default Calc;
```

4.4.1 준비된 예제 확인

■ src/App.tsx

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

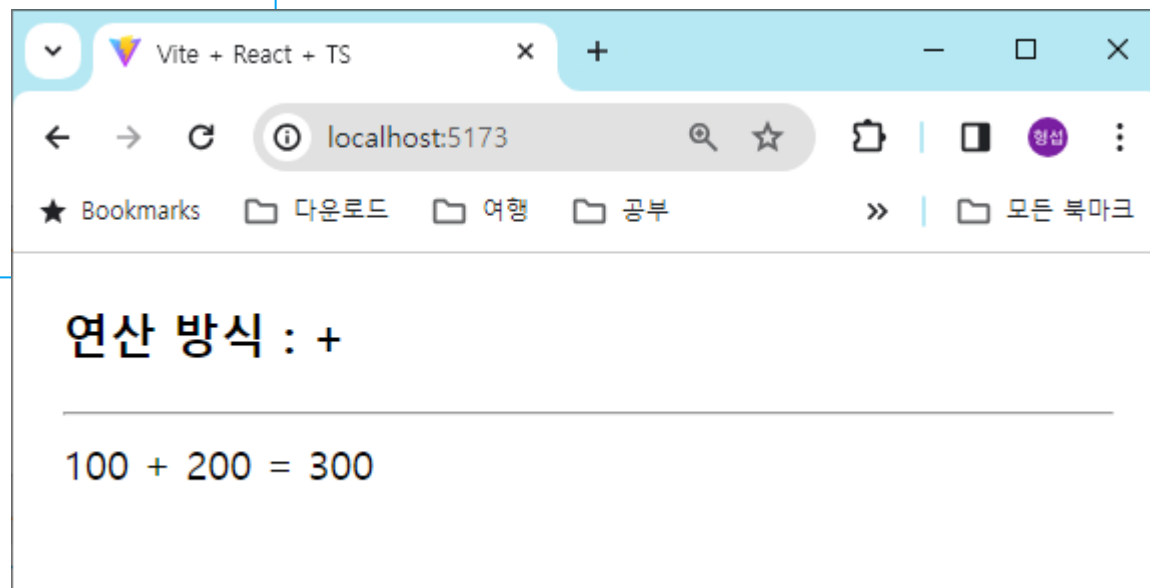
const App = () => {
  const [x, setX] = useState<number>(100);
  const [y, setY] = useState<number>(200);
  const [oper, setOper] = useState<string>("+");

  return (
    <div>
      <Calc x={x} y={y} oper={oper} />
    </div>
  );
};

export default App;
```

■ 실행 결과

- 정상적으로 실행됨
- 하지만 oper를 "&"로 변경한다면?
 - 아무런 오류메시지 없음
 - 결과값이 0으로 나타남



4.4.2 속성 유효성 검사 기능 적용

❖ 기존 예제에 PropTypes 적용

- npm install prop-types
- src/Calc.tsx 변경

```
import PropTypes from "prop-types";

.....(생략-기존 컴포넌트 코드는 그대로)
const calcChecker = (props: CalcPropsTypes, propName: string, componentName: string) => {
  if (propName === "oper") {
    if (props[propName] !== "+" && props[propName] !== "*") {
      return new Error(`${propName}속성의 값은  
반드시 '+', '*'만 허용합니다(at ${componentName}).`);
    }
  }
};

Calc.propTypes = {
  x: PropTypes.number.isRequired,
  y: PropTypes.number.isRequired,
  oper: calcChecker,
};

export default Calc;
```


4.4.2 속성 유효성 검사 기능 적용

- src/App.tsx 변경 : 의도적으로 잘못된 속성 값 전달

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

const App = () => {
  const [x, setX] = useState<number>(101);
  const [y, setY] = useState<string>("ab");
  const [oper, setOper] = useState<string>("&");

  return (
    <div>
      <Calc x={x} y={y} oper={oper} />
    </div>
  );
};

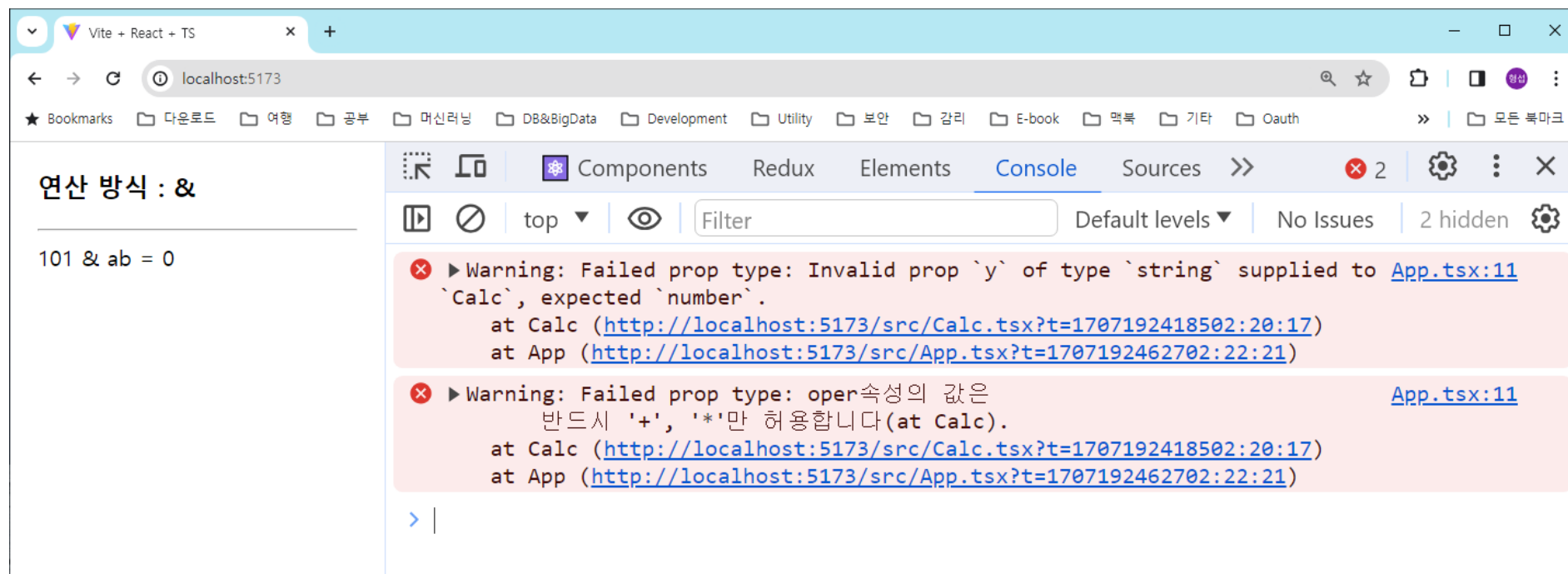
export default App;
```

- 코드 리뷰

- calcChecker 함수의 인자 : props-속성 객체, propName-속성명, componentName-현재 컴포넌트명
 - 이 함수에서 에러 객체가 던져지면 잘못된 속성으로 간주함.
- PropTypes.number.isRequired : number 타입, 필수 입력 여부 설정

4.4.2 속성 유효성 검사 기능 적용

■ 실행 결과



4.4.3 지정 가능한 유효성 검증 타입

❖지정 가능한 유효성 검증 타입

- 단순 타입
 - `PropTypes.array`
 - `PropTypes.bool`
 - `PropTypes.func`
 - `PropTypes.number`
 - `PropTypes.object`
 - `PropTypes.string`
- 복잡한 객체, 배열 속성
 - `PropTypes.instanceOf(Customer)`
 - `PropTypes.oneOf(['+', '*', '/'])`
 - `PropTypes.oneOfType([PropTypes.number, PropTypes.string])`
 - `PropTypes.arrayOf(PropTypes.object)`

4.4.3 지정 가능한 유효성 검증 타입

❖ 지정 가능한 유효성 검증 타입(이어서)

- 복잡한 객체 속성

```
PropTypes.shape({  
  name: PropTypes.string.isRequired,  
  age: PropTypes.number  
})
```

- 함수를 이용한 커스텀 유효성 검증

```
const calcChecker = (props: CalcPropsTypes, propName: string, componentName: string) => {  
  if (propName === "oper") {  
    if (props[propName] !== "+" && props[propName] !== "*") {  
      return new Error(`${propName}속성의 값은  
반드시 '+', '*'만 허용합니다(at ${componentName}).`);  
    }  
  }  
};  
  
Calc.propTypes = {  
  x: PropTypes.number.isRequired,  
  y: PropTypes.number.isRequired,  
  oper: calcChecker,  
};
```

4.4.3 지정 가능한 유효성 검증 타입

❖ y 값이 0~100사이의 짝수라야 한다면?

- `PropTypes.number` 로는 해결 불가
- 추가적인 커스텀 유효성 검증이 필요함.(기존 함수 이용)

```
const calcChecker = (props: CalcpropsType, propName: string, componentName: string) => {
  if (propName === "oper") {
    if (props[propName] !== "+" && props[propName] !== "*") {
      return new Error(`${propName}속성의 값은 반드시 '+', '*'만 허용합니다(at ${componentName}).`);
    }
  }
  if (propName === "y") {
    let y = props[propName];
    if (y > 100 || y < 0 || y % 2 !== 0) {
      return new Error(`${propName}속성의 값은 0이상 100이하의 짝수만 허용합니다.(at ${componentName}).`);
    }
  }
};

Calc.propTypes = {
  x: PropTypes.number.isRequired,
  y: calcChecker,
  oper: calcChecker,
};
```

4.4.3 지정 가능한 유효성 검증 타입

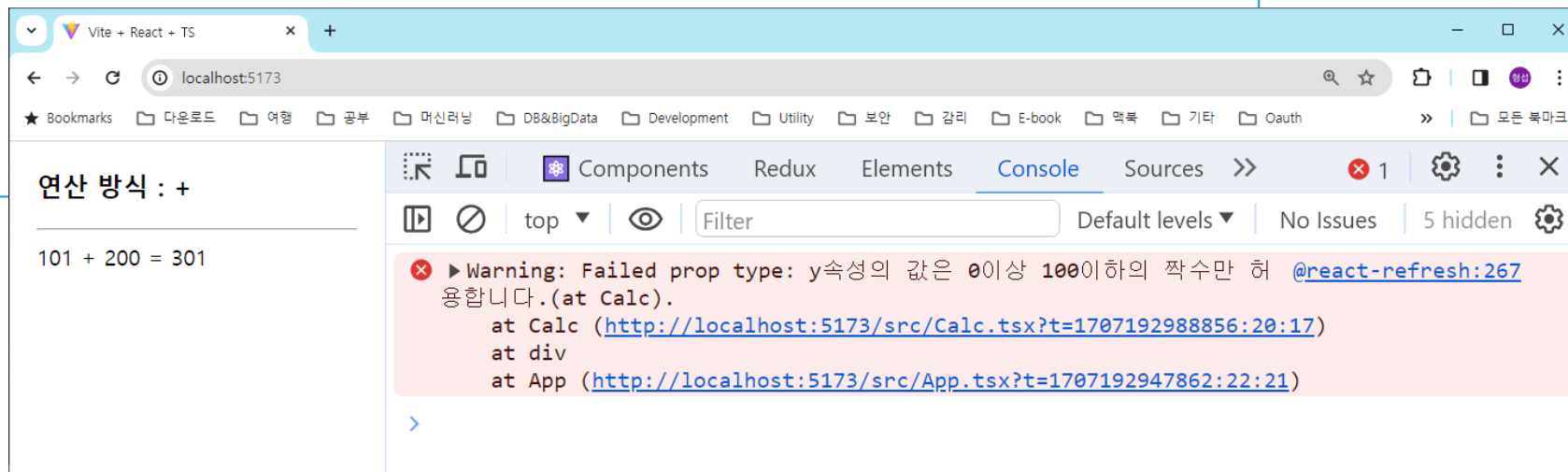
■ src/App.tsx 변경

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

const App = () => {
  const [x, setX] = useState<number>(101);
  const [y, setY] = useState<number>(200);
  const [oper, setOper] = useState<string>("+");

  return (
    <div>
      <Calc x={x} y={y} oper={oper} />
    </div>
  );
};

export default App;
```



4.4.4 속성의 기본값 지정

❖ 속성의 기본값 지정

- 컴포넌트에 defaultProps static 속성을 추가함.
- src/Calc.tsx , src/App.tsx 변경
 - App컴포넌트에서 의도적으로 y, oper 속성을 전달하지 않음

```
.....
Calc.propTypes = {
  x: PropTypes.number,
  y: calcChecker,
  oper: calcChecker,
};

Calc.defaultProps = {
  x: 100,
  y: 20,
  oper: "+",
};

export default Calc;
```

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

const App = () => {
  const [x, setX] = useState<number>(101);
  // const [y, setY] = useState<number>(200);
  // const [oper, setOper] = useState<string>("+");

  return (
    <div>
      <Calc x={x} />
    </div>
  );
};

export default App;
```




Q&A