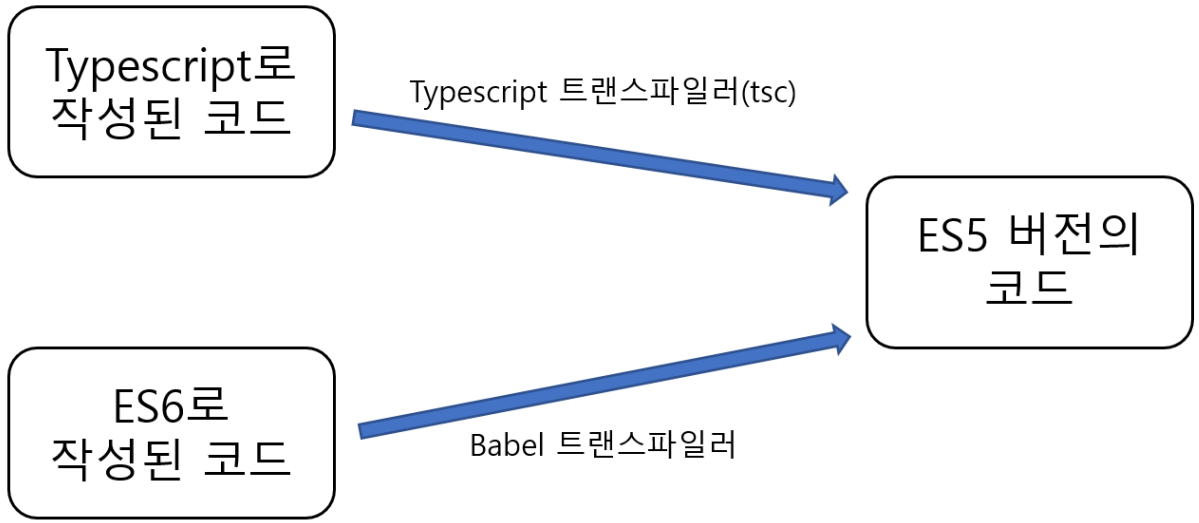
- Typescript part 1

1. Typescript 소개

❖ 트랜스파일러

- Transpile = Translate + Compile
- ES6나 Typescript 언어를 ES5와 같은 이전버전의 자바스크립트 코드로 변환함
- 대표적인 트랜스파일러(Tanspiler)
  - Babel
  - tsc



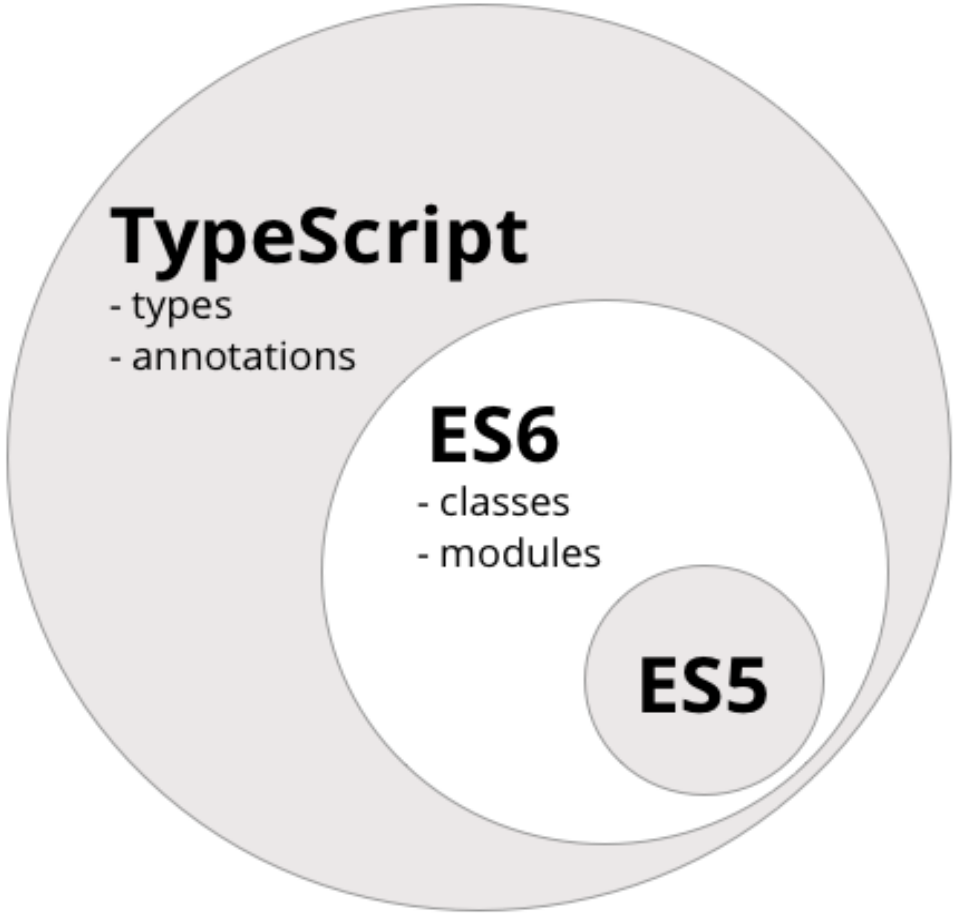
1. Typescript 소개

❖Typescript란?

- ES6에 정적 타입이 추가된 것
- 자바스크립트 언어의 확장버전
  - 기존 ES6 문법을 모두 사용할 수 있음
  - 자바스크립트의 superset
- Microsoft에 의해 관리되고 있음

❖Typescript의 장점

- 정적 타입 사용
  - 코드의 오류를 줄일 수 있음
  - 쉽고 편리한 디버깅
- IDE와 쉽게 통합됨
- 익숙한 문법
  - java나 C#과 문법이 유사함
- js와 마찬가지로 npm을 사용함



1. Typescript 소개

❖ Typescript를 사용하지 않으면 발생할 수 있는 문제점

- case : 여러 개발자 협업 + 대규모 앱
- ES6의 동적 타입은 유연하지만 오타로 인한 에러 발생을 컴파일(빌드)할 때 확인할 수 없음
- 에러는 모두 런타임 오류 : 실행시에 발생
- 디버깅과 테스트에 많은 시간을 허비하므로 생산성 저하

❖Typescript 적용시 가장 주의해야 할 점

- any 타입은 가능하다면 사용하지 않도록 한다.
  - 특히 처음에 오류가 많이 발생하는데, 이런 경우 초보자라면 any를 남발하게 됨
  - 익숙하지 않다면 잠시 strict , lint를 off 로 설정
  - 외부 라이브러리 : 라이브러리 문서 확인, 오픈소스 코드의 .d.ts 파일 확인
- data가 정의되는 곳에서 타입을 선언하고 해당 타입이 다른 모듈에서 이용된다면 함께 export 한다.

1. Typescript 소개

❖동적 타입과 정적 타입

- 동적 타입
  - 변수를 선언할 때 타입을 정의하지 않음 따라서 변수에는 어떤 값이나 할당이 가능함.
  - 값이 할당된 후에는 변수의 타입이 달라짐
  - 다음 코드를 브라우저 콘솔에서 실행해 보셈

```
let a1;
console.log(typeof(a1));
a1 = 100;
console.log(typeof(a1));
a1 = "hello";
console.log(typeof(a1));
a1 = null;
console.log(typeof(a1));
```

- 정적 타입
  - 변수를 선언할 때 해당 변수가 사용할 수 있는 데이터의 타입을 미리 지정함
  - 지정된 타입이 아닌 값이 할당될 때 오류 발생

1. Typescript 소개

❖ 동적 타입과 정적 타입 중 어느쪽이 좋아요?

- 어려운 질문 : 케바케
- 간단한 앱 개발에서는 ES6로도 충분함.
- 하지만 여러 사람이 협업하여 대규모 앱을 개발할 때는 명확히 typescript가 바람직함.

```
//개발자 A가 함수를 만드네
const add = (x, y) => {
  return x+y;
}
//개발자B가 A가 만든 add 함수를 이용하네
//이렇게 이용해도 에러 안남
add("hello", "world");
```

```
//그렇다면 개발자 A는 함수를 만들때 다음과 같이 신경써서 만들어야 함
//그렇다 하더라도 개발자 B에게는 코드 자동완성 기능으로 나타나지 않음
const add = (x, y) => {
  if (typeof(x)!=="number" || typeof(y)!=="number") {
     throw new Error("x,y는 숫자만 전달해야 합니다");
  }
  return x+y;
}
```

1. Typescript 소개

- Typescript 를 사용하면?
  - 협업이 용이해짐 --〉 디버깅 시간 단축 --〉 생산성 향상

```
//개발자 A가 함수를 작성하는 것이 좀더 편해짐
const add = (x:number, y:number) : number => {
    return x+y;
}

//개발자B가 A가 만든 add 함수를 다음과 같이 이용하면 명백하게 에러 발생
add("hello", "world");
```

  - 코드 자동 완성 기능으로 다음과 같이 타입을 명시적으로 알려줌

```
1   const add = (x:number, y:number) : number => {
2       return x+y;
3   }
4
5   add()
```

```
add(x: number, y: number): number
```

2. Typescript 환경 설정

❖Typescript 컴파일러

- 컴파일러(트랜스파일러) : tsc
  - npm install -D typescript

❖ 환경 설정

- mkdir typescript-test
- cd typescript-test
- npm install react react-dom
- npm install -D typescript rimraf @types/react @types/react-dom
- VSCode 실행 후 '보기'-'터미널' 을 열고 다음 명령어 실행
  - npx tsc --int
  - 결과 : tsconfig.json 파일 생성 --〉 기본값 확인
- package.json에 script 러너 추가, type을 module로 지정

```
"scripts": {
    "build": "rimraf ./build && tsc",
},
"type": "module",
```

2. Typescript 환경 설정

❖ tsconfig.json 작성

- Typescript 컴파일러가 컴파일할 때의 기본 설정값 지정
- 자세한 설정 내용은 다음 문서 참조
  - https://typescript-kr.github.io/pages/tsconfig.json.html
- 프로젝트 디렉토리에 생성된 tsconfig.json 파일을 확인하고 다음과 같이 변경

```
{
  "compilerOptions": {
    "outDir": "./build/",
    "allowJs": true,
    "esModuleInterop": true,
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "ESNext",
    "moduleResolution": "node",
    "target": "ESNext",
    "jsx": "react"
  },
  "include": ["./src/**/*"]
}
```

2. Typescript 환경 설정

❖tsconfig.json 의 주요 설정 옵션

- 특히 Type Checking 관련 설정 옵션에 유의
  - "strict" : true

- compilerOptions.outDir: 트랜스파일한 최종 결과물이 저장되는 경로를 지정합니다.
- compilerOptions.allowJs: 트랜스파일할 대상에 .js, .jsx와 같은 파일도 포함합니다.
- compilerOptions.esModuleInterop: ES6가 아닌 commonJS는 모듈을 임포트하고 익스포트하는 방법이 다릅니다. ES6는 import문을 사용하지만, commonJS는 require문을 사용하는 등의 차이가 있습니다. 따라서 commonJS로 작성한 모듈을 ES6에서 임포트할 때 약간의 문제를 일으키기도 하는데, 이를 해소하기 위해 이 속성을 true로 지정합니다.
- compilerOptions.resolveJsonModule: 이 속성을 true로 지정하면 .json 텍스트 파일을 자바스크립트 객체로 임포트할 수 있습니다.
- compilerOptions.sourceMap: 트랜스파일한 코드와 함께 디버깅을 위한 .js.map과 같은 소스 맵 파일을 생성합니다.
- compilerOptions.noImplicitAny: 타입스크립트는 데이터 형식이 지정되지 않으면 암시적으로 any 데이터 형식을 사용합니다. 타입스크립트를 사용하는 이유는 암시적 데이터 형식을 사용하지 않도록 하는 것이므로 필수적으로 이 속성을 true로 지정하세요.
- compiler.module: 컴파일된 결과물이 사용하게 될 모듈 시스템 방식을 지정합니다. target 속성이 "es5"이면 "commonjs"를 주로 지정하고, target 속성이 "es6" 혹은 그 이상의 버전을 사용하면 "ES6", "ES2015", "ESNext" 등을 지정합니다.
- compilerOptions.target: 트랜스파일한 최종 결과물의 형태를 지정합니다. 여기서는 es5로 지정했으므로 ES6 문법으로 작성한 코드는 모두 ES5로 트랜스파일됩니다.
- compilerOptions.jsx: 이 책에서는 타입스크립트를 이용해 리액트 애플리케이션을 개발하는 내용을 다룹니다. jsx는 리액트에서 주로 사용하는 HTML 마크업 형태의 자바스크립트 확장 문법입니다. 이 속성을 "react"로 지정하면 jsx가 사용된 부분을 React.createElement( ) 함수의 호출 형태로 트랜스파일합니다.
- include: 파일 패턴을 지정해서 트랜스파일할 대상 파일을 지정합니다.
- exclude: 트랜스파일할 때 배제 대상을 지정합니다. 이 예제에서는 node_modules 디렉터리의 모든 파일과 파일명이 .spec.ts로 끝나는 파일은 트랜스파일하지 않습니다.

2. Typescript 환경 설정

❖환경 설정 테스트

- src/App.tsx, src/index.tsx

```
import React from 'react';

const App = () => {
    return (
        <div>Hello</div>
    );
};

export default App;
```

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'

ReactDOM.createRoot(document.getElementById('root')!).render
(
    <App />
)
```

- 트랜스파일 테스트
  - npm run build
  - outDir (build 디렉토리) 확인

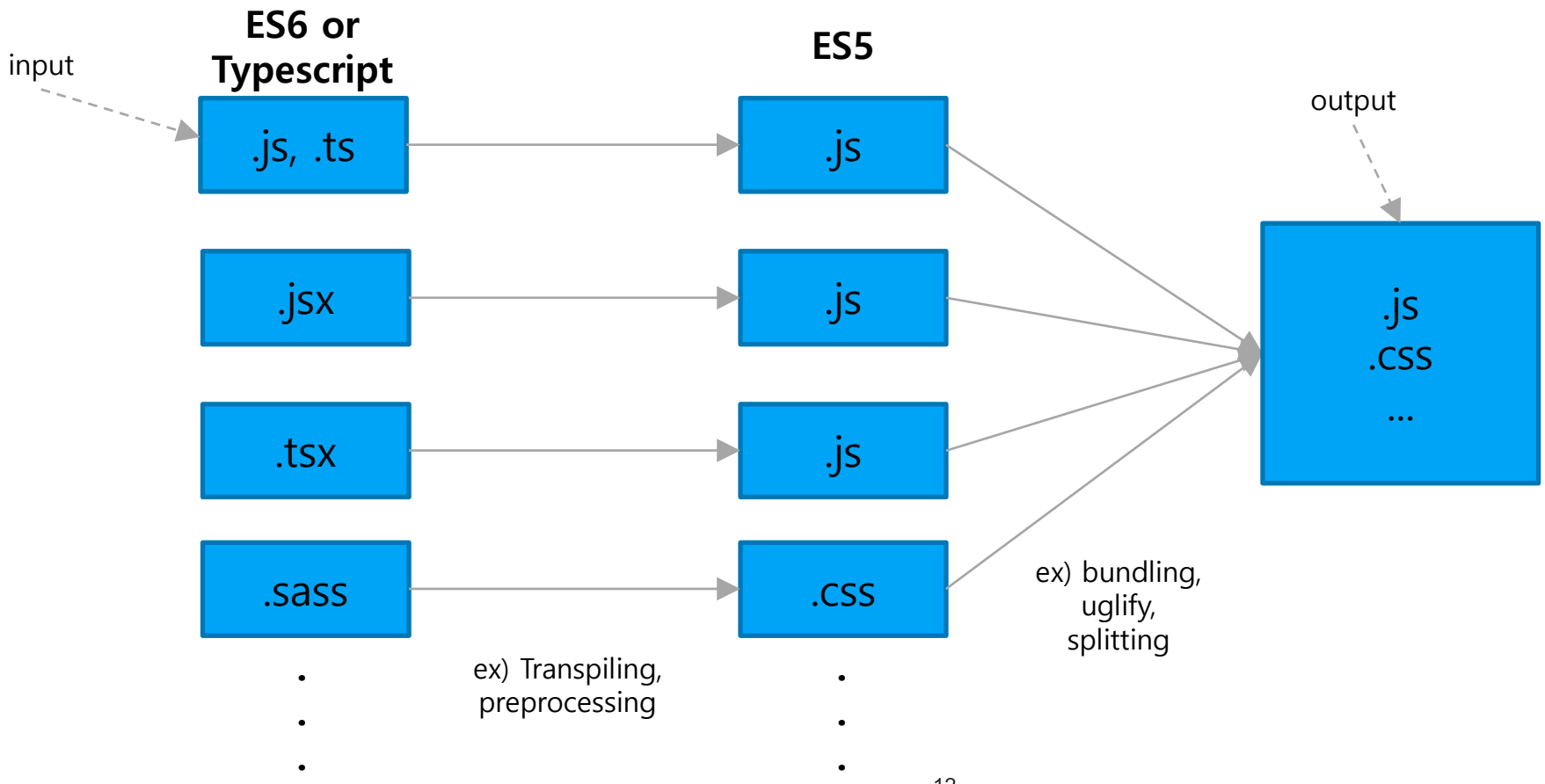
❖별도의 설치 없이 Typescript 코드 테스트

- https://www.typescriptlang.org/play

3. Typescript + React

❖아키텍처

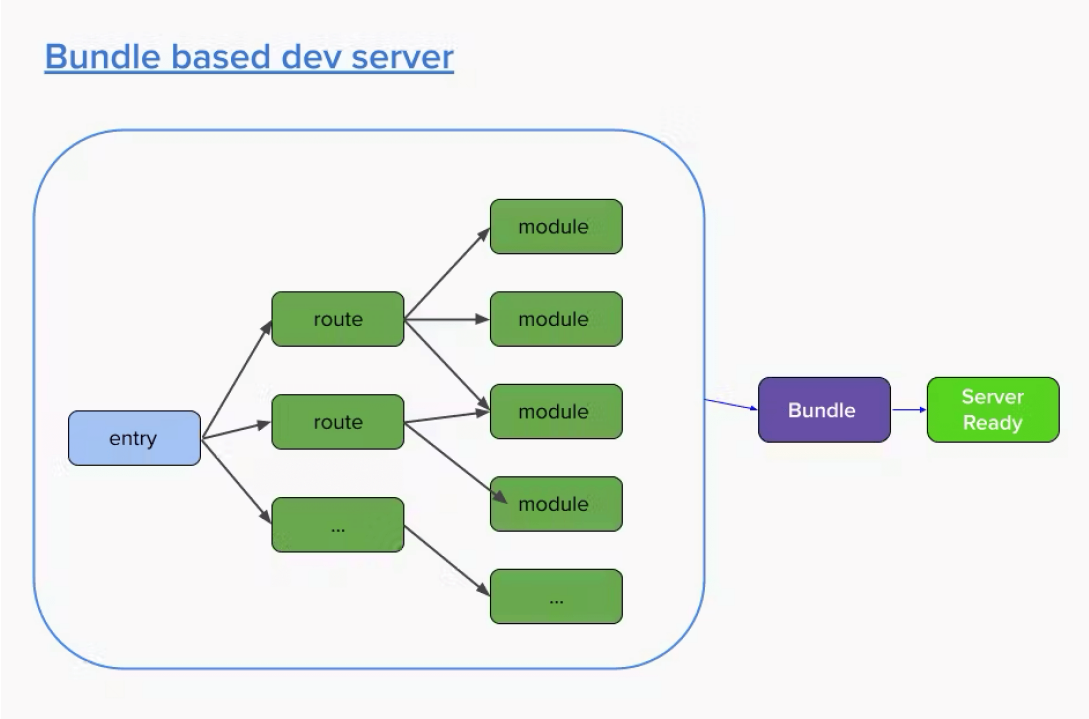
- 하지만 이런 개발 환경을 직접 설정하는 것은 대단히 고통스러운 일
- 그렇기 때문에 Vite, CRA를 사용하는 것임



3. Typescript + React

❖Typescript 기반의 리액트 프로젝트 생성하기

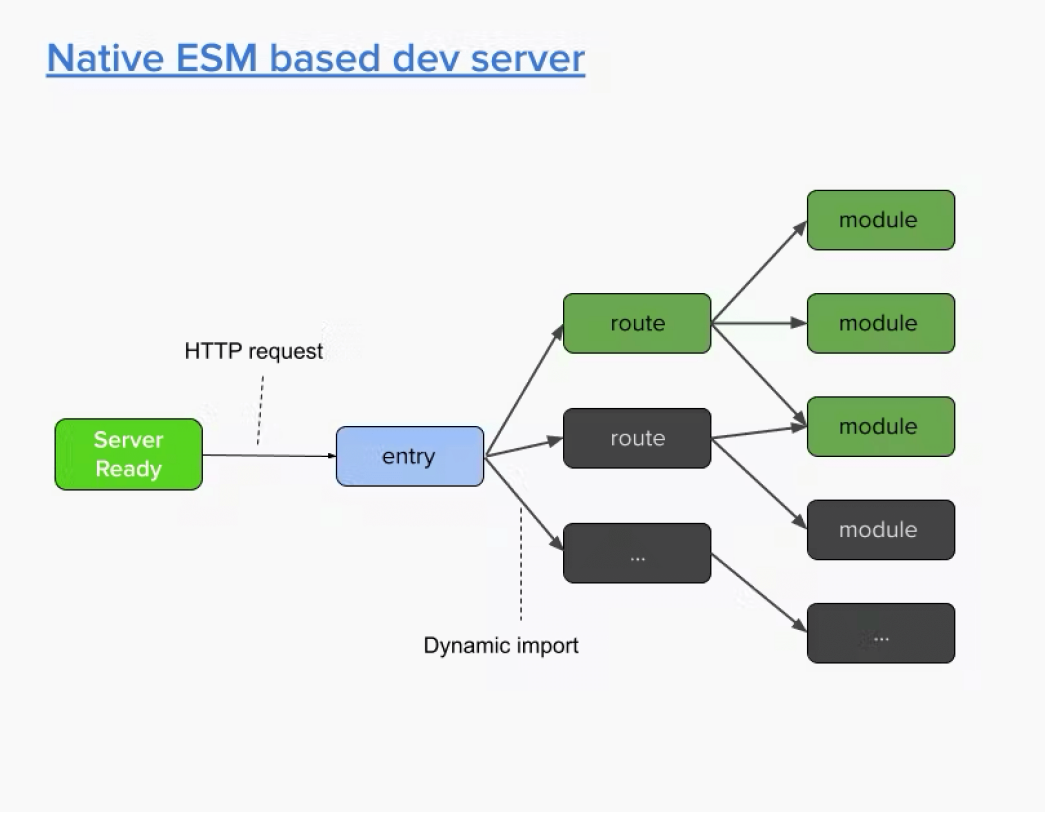
- CRA
  - npx create-react-app 프로젝트디렉토리명 --template typescript
  - 모듈 번들러 : Webpack
  - 개발용 웹서버 : webpack-dev-server



출처 : https://refine.dev/blog/what-is-vite-vs-webpack/#vite-vs-webpack

3. Typescript + React

- Vite
  - npm init vite 프로젝트디렉토리명 -- --template react-ts
  - 모듈 번들러 : Rollup
  - 개발용 웹서버 : vite-dev-server, ESM 기반의 개발 서버



출처 : https://refine.dev/blog/what-is-vite-vs-webpack/#vite-vs-webpack

4. type vs interface

- ❖type
- ❖interface
- ❖type과 interface의 비교

4.1 type

❖ type

- 타입에 대한 별칭을 지정하는 방법을 제공
  - 기본 타입(Primitive Type)에 대한 별칭을 지정할 수 있음
  - 복잡한 타입에 대한 별칭을 지정할 수 있음

```
let name1 : string = "홍길동";

//단순 타입(string)에 대한 별칭
type MyType = string;
let name2 : MyType = "이몽룡";

console.log(name1);
console.log(name2);

//합성 타입을 재사용하려면 type 별칭을 지정함
let person11 : { name:string; age:number; } = { name:"홍길동", age:20 };
console.log(person11);

type PersonType1 = { name:string; age:number; };
let person12 : PersonType1 = { name:"이몽룡", age:21 };
console.log(person12);
```

4.1 type

❖ 선택적 필드

- ? : 해당 필드는 선택적으로 존재할 수 있음

```
type PersonType2 = { name:string; age?:number; email:string; };
//p21, p22 모두 정상
let p21: PersonType2 = {name:"홍길동", email:"gdhong@test.com" };
let p22: PersonType2 = {name:"이몽룡", age:21, email:"mrlee@test.com" };

console.log(p21);
console.log(p22);
```

❖읽기 전용 필드

- readonly : 해당 필드에 값이 한번 주어지면 변경이 불가능함

```
type PersonType3 = { name:string; age?:number; readonly email:string; };

let p3: PersonType3 = {name:"홍길동", email:"gdhong@test.com" };
console.log(p3);

//에러 발생
p3.email = "gdhong@gmail.com";
```

4.1 type

❖ type을 활용해 다른 type을 선언할 수 있음

```
type ContactType = {
   name:string;
   mobile:string;
   email?: string;
};

type ContactListType = {
   pageNo: number;
   pageSize: number;
   contacts : ContactType[];
};

const contactList : ContactListType = {
    pageNo: 1,
    pageSize: 10,
    contacts: [
        { name:"홍길동", mobile:"010-1111-1111" },
        { name:"박문수", mobile:"010-1111-1112", email:"mspark@gmail.com" },
        { name:"이몽룡", mobile:"010-1111-1113" },
    ]
}

console.log(contactList);
```

4.2 interface

❖interface란?

- 객체, 함수, 배열, 클래스 등에 대한 규칙을 정의할 수 있는 기능
- type과 유사함

```
//type과 비교
// type PersonType2 = {
//     name:string;
//     age?:number;
//     email:string;
// };

//선택적 필드, readonly 필드 모두 사용 가능
interface IPerson {
    name:string;
    age?:number;
    email:string;
};

//p31, p32 모두 정상
let p31: IPerson = {name:"홍길동", email:"gdhong@test.com" };
let p32: IPerson = {name:"이몽룡", age:21, email:"mrlee@test.com" };

console.log(p31);
console.log(p32);
```

4.2 interface

❖함수에 대한 타입, 인터페이스 지정

```
type FunctionType1 = (x:number, y:number)=> number

interface FunctionInterface1 {
    (x:number, y:number):number;
}

const add : FunctionType1 = (x:number,y:number)=>{
    return x+y;
}

const multiply : FunctionInterface1 = (x:number,y:number)=>{
    return x*y;
}

console.log(add(3,4))
console.log(multiply(3,4))
```

4.2 interface

❖클래스에 대한 타입 지정

```
// type 정의
type TodoType = { id: number; todo: string; done: boolean; }
interface ITodo { id: number; todo: string; done: boolean; };

class TodoItem1 implements TodoType {
    constructor(
        public id: number,
        public todo: string,
        public done: boolean) { }
}
class TodoItem2 implements TodoType {
    constructor(
        public id: number,
        public todo: string,
        public done: boolean) { }
}


const todoItem1 = new TodoItem1(1001, '타입스크립트 학습', true);
const todoItem2 = new TodoItem2(1002, 'Zustand', false);
console.log(todoItem1);
console.log(todoItem2);
```

4.2 interface

❖Duck Typing

- 값, 객체의 모양에 촛점을 맞춘 Type 검사 기법
  - 동일한 이름, 형상의 속성, 메서드가 있다면 같은 타입!!
- 비유를 들자면?
  - 사냥꾼 : "너 오리 맞어?"
  - 까마귀 : "네. 저는 오리입니다."
  - 사냥꾼 : "니가 오리라는 것을 증명해봐"
  - 까마귀 : "네. 저는 '꽥꽥()' 이라는 메서드를 가지고 있습니다."
  - 사냥꾼 : "너 오리 맞구나!!"

❖ Typescript에서는 왜 DuckTyping을 지원하는거야?

- typescript의 정적 타입 : 빌드(컴파일)할 때 타입 기반 검증
- duck typing : 런타임시에 타입 검증 방법
  - 동적 타입의 유산
- typescript는 둘다 지원함

```
interface AnimalType {
    유형:string;
    짖어라: ()=>void;
};

class Dog implements AnimalType {
    유형 = "개";
    짖어라 = ()=>{
        console.log("멍멍!");
    }
}
class Cat {
    유형 = "고양이";
    짖어라 = ()=>{
        console.log("야옹!");
    }
}

let dog1:AnimalType = new Dog();
let dog2:AnimalType = new Cat();
dog1.짖어라();
dog2.짖어라();
```

4.3 type VS interface

❖이전까지의 내용을 토대로

- 무슨 차이가 있지?
- 무엇을 사용할까?

❖차이점

- interface
  - extends 로 확장
  - 선언 병합(확장)이 가능 : type은 불가능
- type
  - intersection으로 확장
  - union : interface는 불가능
  - computed property name : interface는 불가능
  - tuple : interface는 불가능

4.3.1 확장

❖확장 방법

- type : intersection (&)

```
type PersonType1 = { name:string; tel:string; };
type AdditionType1 = { email:string; };
type EmployeeType1 = PersonType1 & AdditionType1;
//정상
const e11: EmployeeType1 =  { name:"홍길동", tel:"010-1111-1111", email:"gdhong@gmail.com" };
//에러
const e12: EmployeeType1 =  { name:"홍길동", tel:"010-1111-1111" };
```

- interface : extends 키워드

```
interface PersonType2 {
    name:string;
    tel:string;
};

interface EmployeeType2 extends PersonType {
    email: string;
}

const e2: EmployeeType2 =  { name:"홍길동", tel:"010-1111-1111", email:"gdhong@gmail.com" };
```

4.3.1 확장

❖ 선언 확장 : interface만 가능함

```
interface EmployeeType3 {
    name:string;
    tel:string;
};

interface EmployeeType3 {
    email: string;
}

const e31: EmployeeType3 =  { name:"홍길동", tel:"010-1111-1111", email:"gdhong@gmail.com" };
//오류
const e32: EmployeeType3 =  { name:"홍길동", tel:"010-1111-1111" };
```

4.3.2 Union

❖interface는 union 을 지원하지 않음

```
//string 타입이거나  number 타입인 경우
type YourType = string | number;

let a1 : YourType = 100;                    //ok
let a2 : YourType = "hello";                //ok
```

❖union이 반드시 필요한 상황

- **"no, name 속성은 반드시 존재하고 email, tel 중 하나의 속성을 포함하는 객체의 타입"은?**
- interface를 사용하며 선택적 속성을 사용하면 어떨까? --〉 X

```
//선택적 속성을 생각해볼 수 있지만.....
type PersonType3 = { no:number, name:string, email?: string, tel?:string };

//아래 두 케이스를 모두 허용해버림
let a31 : PersonType3 = { no:1001, name:"홍길동" };
let a32 : PersonType3 = { no:1001, name:"홍길동", email: "...", tel: "..." };
```

4.3.2 Union

❖ union을 사용하여 해결

```
//두개의 Type을 Union 하여 문제 해결
type PersonType41 = { no:number; name:string; email:string };
type PersonType42 = { no:number; name:string; tel:string };
type PersonTypeUnion = PersonType41 | PersonType42;

let a41 : PersonTypeUnion = { no:1001, name:"홍길동", email:"gdhong@test.com" };
let a42 : PersonTypeUnion = { no:1001, name:"홍길동", tel:"010-1111-1111" };
```

4.3.2 Union

❖Union Type을 사용할 때 주의사항

- 오른쪽 그림의 오류
  - 원인은 무엇인가?

```
1   type Student = {
2     name: string;
3     studentid: string;
4   }
5
6   type Employee {
7     name: string;
8     employeeid: string;
9   }
10
11  const viewPerson = (person:Studer
12      console.log(`이름 : ${person.
13      console.log(`학번 : ${person.studentid}`);
14  }
15
16  viewPerson({name:"홍길동", studentid:"s1001010"})
```

```
Property 'studentid' does not exist on type 'Student | Employee'.
  Property 'studentid' does not exist on type 'Employee'. (2339)

any

View Problem (Alt+F8)    No quick fixes available
```

```
type Student = {
  name: string;
  studentid: string;
}

type Employee {
  name: string;
  employeeid: string;
}

const viewPerson = (person:Student | Employee) => {
    console.log(`이름 : ${person.name}`);
    console.log(`학번 : ${person.studentid}`);
}

viewPerson({name:"홍길동", studentid:"s1001010"})
```

4.3.2 Union

❖이전 페이지 오류의 원인은 무엇인가?

- Student, Employee 타입중 어떤 타입의 속성이 전달될지 알 수 없으므로 두 타입이 모두 가지고 있는 공통적인 속성에 대해서만 정상적이라고 판단함
  - 타입스크립트 컴파일러가 오류라고 판단함
- 하지만 정상적으로 실행됨.
  - 런타임에는 문제 없음
- 이런 상황이 꽤 많이 발생됨
- 이 문제의 해결을 위해서는?
  - 개발자가 직접 type guard를 작성해야 함

❖type guard

- union type의 값에 대해 타입을 검사하는 기능을 수행하는 코드
- typeof 연산자로는 사용자 정의 타입에 대한 검사를 수행할 수 없음,
- 따라서 특정한 속성이 있는지 여부로 판단해야 함
- in 연산자
  - "속성명" in obj

4.3.2 Union

❖type guard

```
type Student = {
  name: string;
  studentid: string;
}

type Employee {
  name: string;
  employeeid: string;
}

const viewPerson = (person:Student | Employee) => {
    if ("studentid" in person){
        console.log(`이름 : ${person.name}`);
        console.log(`학번 : ${person.studentid}`);
    } else {
        console.log(`이름 : ${person.name}`);
        console.log(`사번 : ${person.employeeid}`);
    }
}

viewPerson({name:"홍길동", studentid:"s1001010"})
viewPerson({name:"이몽룡", employeeid:"e123456"})
```

```
type Student = {
  name: string;
  studentid: string;
}

type Employee {
  name: string;
  employeeid: string;
}

const viewPerson = (person:Student | Employee) => {
    if ("studentid" in person){
        console.log(`이름 : ${person.name}`);
        console.log(`학번 : ${person.studentid}`);
    } else {
        console.log(`이름 : ${person.name}`);
        console.log(`사번 : ${person.employeeid}`);
    }
}

viewPerson({name:"홍길동", studentid:"s1001010"})
viewPerson({name:"이몽룡", employeeid:"e123456"})
```

4.3.2 Union

❖type guard, union, intersection 예제

- ts-react-test 예제 참조
- 핵심 코드 : src/TestComponent.tsx

```
import Error from "./Error";

type PropsCommon = { children: string | JSX.Element | JSX.Element[]; };
type Props1 = PropsCommon & { name: string; };
type Props2 = PropsCommon & { nick: string; };
type Props = Props1 | Props2;

const TestComponent = (props: Props) => {
    if ("name" in props || "nick" in props ) {
        return props.children
    }
    return <Error />;
};

export default TestComponent;
```

4.3.3 Tuple

❖Tuple이란?

- 길이가 고정되고 각 요소의 타입이 지정된 배열

```
type TupleType = [ string, number, boolean ];

let t1 : TupleType = [ "hello", 1004, true ];
```

- useState 훅이 Tuple 타입을 사용함

```
const [visible, setVisible] = useState<boolean>(false);
```

```
(alias) useState<boolean>(initialState: boolean | (() => boolean)): [boolean, React.Dispatch<React.SetStateAction<boolean>>] (+1
overload)
import useState
```

Returns a stateful value, and a function to update it.

@version — 16.8.0

@see — https://react.dev/reference/react/useState

```
useState<boolean>(false);
```

4.3.4 열거형과 상수

❖열거형(Enum : Enumeration)

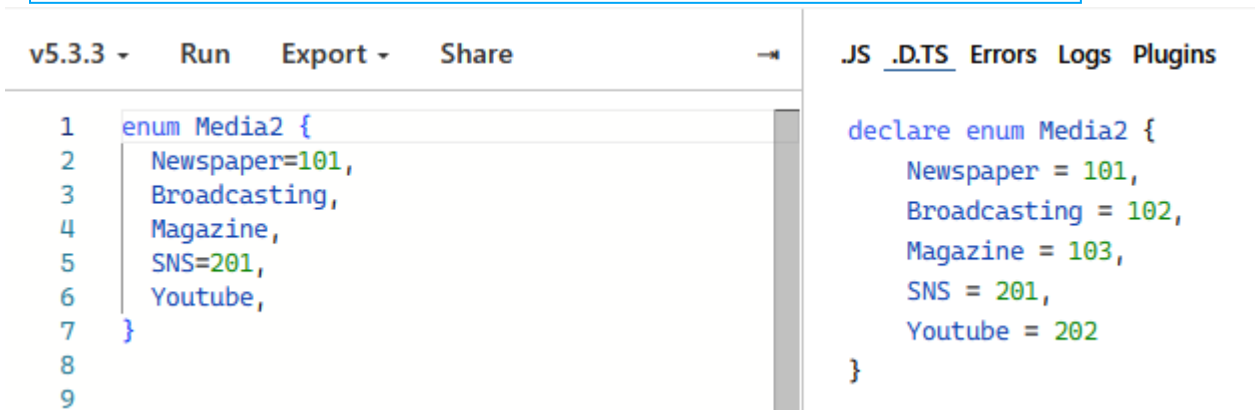
- 정해진 값을 가지는 집합을 표현함.
  - 가독성을 위해 서로 관련된 값들을 하나의 namespace에 묶어서 관리함
- 값을 직접 지정할 수 있음 : 문자, 숫자 등

```
enum Media1 {
  Newspaper = "신문",
  Broadcasting = "방송",
  SNS = "SNS",
  Magazine = "잡지",
  Youtube = "유튜브",
}

let media1 :Media1 = Media1.Youtube;
console.log(media1); //"유튜브" 출력
```

```
enum Media2 {
  Newspaper=101,        //101
  Broadcasting,           //102
  Magazine,              //103
  SNS=201,               //201
  Youtube,               //202
}

let media2 :Media2 = Media2.Youtube;
console.log(media2); //202가 출력
```

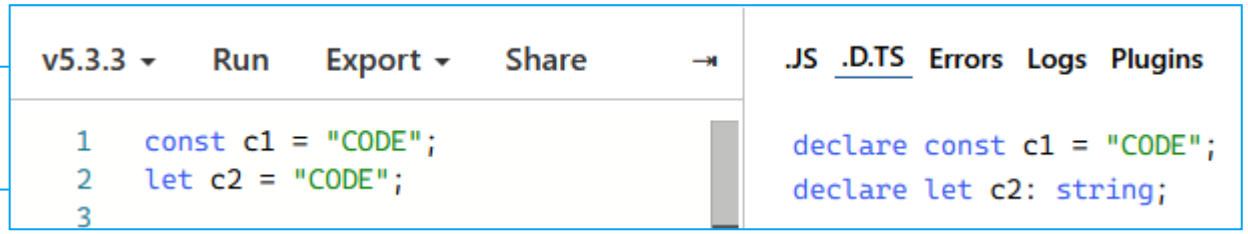


4.3.4 열거형과 상수

❖ 상수(const: Constant)와 열거형 비교

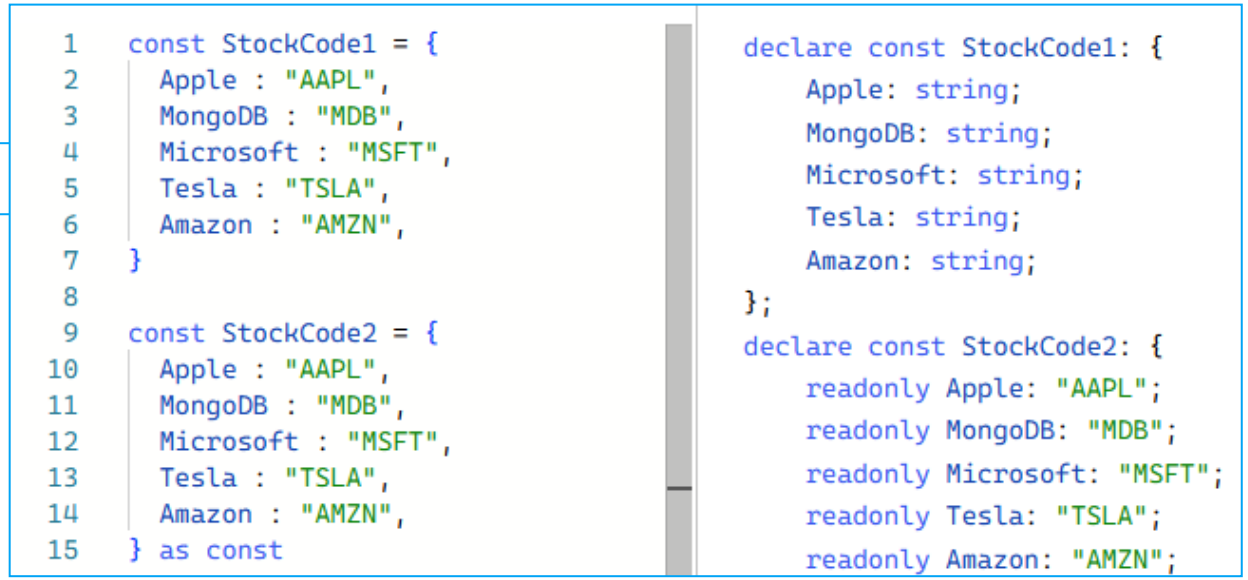
- 타입 추론(inference)

```
const c1 = "CODE";          //"CODE"로 타입 추론
let c2 = "CODE";            //string 타입으로 추론
```



- 상수 단언(as const : const assertion)
  - 추론의 범위를 좁히고 값의 재할당을 막아줌

```
let c3 = "CODE" as const;   //"CODE"로 타입 추론
```



4.3.4 열거형과 상수

❖목적

- enum의 목적
  - 서로 관련된 값들을 하나의 네임스페이스로 묶어 관리하기 위해 사용함
  - 역방향 매핑 지원 : 일반적인 경우라면 필요하지 않음
    - key ---〉 value, value--〉 key
  - const enum 추천
    - 역방향 매핑 하지 않음
    - 트랜스파일할 때 불필요한 코드를 생성하지 않음
- as const
  - 타입 추론의 범위를 좁히고 값의 재할당을 막기 위해 사용함
  - 객체 내부의 필드가 모두 readonly로 바뀜

❖결론

- 목적에 맞게 사용하자
- const enum 괜찮다.

4.4 type, interface 정리

❖type vs interferface

- 일반적으로는 interface 사용
- union이 필요하거나 tuple 타입이 필요할 때는 type 사용

❖enum 과 as const는 용도에 맞게 사용

Q&A