10/13 강의자료

TS코드 테스트 : https://www.typescriptlang.org/play

code runner에서 TS 실행하기

$ npm install -g ts-node

를 설치해줍니다.

* ts-node는 TS코드를 JS로 컴파일 하지 않고 직접 실행시키는 라이브러리 입니다.

typescript 설치하기

npm install -g typescript

1교시

1. 함수의 매개변수

▼ typescript 동작원리

먼저 JS, JAVA의 컴파일러들은 텍스트를 추상문법트리(abstract syntax tree, AST)라는 자료구조로 변환합니다.

그리고 AST를 바이트 코드로 변환하여 런타임 프로그램이 바이트 코드를 읽는 방식입니다.

하지만 타입스크립트는 코드를 AST로 만들고 먼저 타입검사기가 AST를 확인합니다. 이 과정에서 타입 에러를 확인하는 것입니다.

그리고 나서 타입스크립트 AST가 자바스크립트 소스를 만듭니다. 그 뒤에는 JS와 동일하게 작동하죠 이렇게 컴파일 과정에서 타입을 체크해 에러를 찾을 수 있기 때문에 더욱 안정적으로 사용할 수 있는 것입니다.

▼ 기본적인 타입 종류는?

TypeScript의 기본적인 타입 종류에 대해서 알아보겠습니다.

```
// Boolean
let isDone: boolean = false;
console.log(isDone);

// Number : 16진수, 10진수, 2진수, 8진수
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
console.log(decimal);
console.log(hex);
console.log(binary);
console.log(octal);

// String
let color: string = "blue";
color = "red";
console.log(color);

// 템플릿 문자열 사용
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${fullName}.
I'll be ${age + 1} years old next month.`;
console.log(sentence);

// Array
let list1: number[] = [1, 2, 3]; // 타입 뒤에 []
let list2: Array<number> = [1, 2, 3]; // 제네릭 배열 타입
console.log(list1);
console.log(list2);

// Tuple : 요소의 타입과 개수가 고정된 배열
let x: [string, number];
x = ["hello", 10]; // 성공
console.log(x[0].substring(1)); // 성공
// console.log(x[1].substring(1)); // 오류, 'number'에는 'substring' 이 없습니다.

// Enum
enum Color1 { // enum은 0부터 시작
    Red,
    Green,
    Blue,
```

```
}
let c1: Color1 = Color1.Green;
console.log(c1);

enum Color2 { // 모든 값을 수동으로 설정가능
    Red = 1,
    Green = 3,
    Blue,
}
let c2: Color2 = Color2.Red;
let c3: Color2 = Color2.Green;
let c4: Color2 = Color2.Blue;
console.log(c2);
console.log(c3);
console.log(c4);

// Any : 타입 검사를 하지 않고, 그 값들이 컴파일 시간에 검사를 통과
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false;

// Void :  어떤 타입도 존재할 수 없음 any 반대
function warnUser(): void {
    console.log("This is my warning message");
}
warnUser();

let unusable: void = undefined;
console.log(unusable);

// Null / Undefined : Null / undefined 이와 다른 값 안됨
let u: undefined = undefined;
let n: null = null;

// Never : 절대 발생할 수 없는 타입 any도 안됨
function infiniteLoop(): never {
    while (true) {}
}

// Object : 원시타입(number, string, boolean, bigint, symbol, null,undefined) 가 아닌 나머지
let userA: { name: string; age: number } = {
    name: "juyoung",
    age: 27,
};

// Type assertions (형변환)
// angle-bracket 방법
let someValue1: any = "this is a string";

let strLength1: number = (<string>someValue1).length;

// as 방법
let someValue2: any = "this is a string";

let strLength2: number = (someValue1 as string).length;
```

참고: https://typescript-kr.github.io/pages/basic-types.html

▼ 타입 명시 방법은?

직접 적어서 사용하는 방법, type을 사용하는 방법, interface를 사용하는 방법이 있습니다.

각 방법들에 대해서 실습을 통해 알아보겠습니다.

2교시

2. 클래스 만들기

▼ tsconfig.json 알아보기

디렉토리에 `tsconfig.json` 파일이 있다면 해당 디렉토리가 TypeScript 프로젝트의 루트가 됩니다.

`tsconfig.json` 파일은 프로젝트를 컴파일하는 데 필요한 루트 파일과 컴파일러 옵션을 지정합니다

compilerOptions안에

‘target’은 타입스크립트파일을 어떤 버전의 자바스크립트로 바꿔줄지 정하는 부분입니다.

es5로 셋팅해놓으면 es5 버전 자바스크립트로 컴파일(변환) 해줍니다.

‘module'은 자바스크립트 파일간 import 문법을 구현할 때 어떤 문법을 쓸지 정하는 곳입니다.

commonjs는 require 문법을 / es2015, esnext는 import 문법을 사용합니다.

‘strict’은 엄격 모드 on/off

‘esModuleInterop’은 CommonJS와 ES 모듈 간의 배출 상호 운용성을 지원

‘experimentalDecorators'은 모듈과 함께 작동하는 데코레이터에 대한 유형 메타데이터 방출여부

‘skipLibCheck’은 선언 파일 형식 확인 여부

‘forceConsistentCasingInFileNames’은 대소문자가 일치하지 않는 동일한 파일에 대한 참조를 허용 여부

다른 많은 설정값들 :
https://www.typescriptlang.org/tsconfig#allowSyntheticDefaultImports

3. 클래스 상속받기

▼ 상속의 장점은?

다른 클래스가 가지고 있는 멤버(필드와 메소드)들을 새로 작성할 필요 없이 클래스에서 직접 만들지 않고 가져와서 사용할 수 있습니다. 또한 상속의 가장 큰 목적은 코드의 재사용입니다. 더 빠르고 유지보수가 쉽고, 중복이 적고, 통일성이 있는 코드를 작성하기 위해 상속을 사용합니다.

상속의 장점

1. 적은 양의 코드로 새로운 클래스를 작성 가능
2. 코드를 공통적으로 관리하기 때문에 여러 코드의 추가 및 변경이 용이
3. 중복을 제거해서 생산성과 쉬운 유지보수성에 크게 기여

라고 정리할 수 있겠습니다.

이렇게 상속을 받을 때 부모의 멤버값을 가지고와 사용할 때 `super([arguments])`라는 메서드를 사용합니다.

참고 :
https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Operators/super

4. 추상 클래스

▼ 추상 클래스란?

먼저 **추상화**는 **객체 지향 프로그래밍(OOP:Object-Oriented Programming)**의 핵심 아이이디어 중 하나입니다. 복잡성을 최소화하고 고급 아키텍처 문제를 해결하는데 도움이 되는 기술이며, 하위 수준의 세부 사항을 미리 구현할 필요가 없습니다. 상위 수준에 집중하고 나중에 세부 사항을 구현합니다.

이러한 추상화를 하기위해서 추상 클래스라는 것이 생겨났는데요 구현은 추상 클래스를 상속받은 구체 클래스에게 넘기는 것이죠.

▼ 접근 지정자 (JAVA 기준)

TypeScript는 JAVA기준에서 Default가 없고 기본값이 public임

<<JAVA기준>>

- public : 오픈 마인드
- protected : 가족만 접근 가능 (상속받으면 다 가능)
- default : 일촌만 가능(같은 패키자 안에서 가능) → 기본값
- private : 쇄국정책

한정자	클래스 내부	동일 패키지	하위 클래스	그 외의 영역
public	●	●	●	●
protected	●	●	●	X
default	●	●	X	X
private	●	X	X	X

접근 영역 : public > protected > default > private

3교시

5. interface

▼ interface

인터페이스는 일반적으로 **타입 체크를 위해 사용되며 변수, 함수, 클래스에 사용**할 수 있습니다. 인터페이스는 여러가지 타입을 갖는 프로퍼티로 이루어진 새로운 타입을 정의하는 것과 유사합니다.

인터페이스는 프로퍼티와 메소드를 가질 수 있다는 점에서 클래스와 유사하나 직접 인스턴스를 생성할 수 없고 모든 메소드는 추상 메소드입니다.

타입으로도 쓰일 수 있고, 클래스에 인터페이스로도 사용될 수 있습니다.

▼ type과의 차이점

모든 것을 지정할 수 있는 type과는 다르게 interface는 obj 형태만을 사용할 수 있습니다.

하지만 obj형태에 최적화 시켜논 것이기 때문에 성능과 편림함은 interface가 더 좋습니다. 그래서 class나 object를 정의하고싶을 땐 interface를 쓰는것을 추천드립니다.

- Class나 Object의 모양을 정의하고 싶을땐 interface
- 나머지의 경우는 type 으로 정의

또한 type과 interface의 차이점으로는 확장방법에 있습니다.

Type은 선언적 확장이 불가능하여 overloading이 안되는 반면 interface는 선언적 확장이 가능하여 더 편리하게 사용할 수 있습니다.

- type 확장

```
type PeopleType = {
    name: string
    age: number
}

type StudentType = PeopleType & {
    school: string
}

// 선언적 확장 불가능
type OS = {
```

```
  title : string
}

type OS = {
  version : number
}
// Error남 이름 중복
```

- interface 확장

```
interface PeopleInterface {
  name: string
  age: number
}

interface StudentInterface extends PeopleInterface {
  school: string
}

// 선언적 확장 가능 (Type은 안됨)
interface OS{
  title : string
}

interface OS {
  version : number
}


const windoVersion : OS = {
  title : "윈도우",
  version: 10,
}
}
```

6. generic 함수

▼ generic 설명

C#과 Java 같은 언어에서, 재사용 가능한 컴포넌트를 생성하는 도구상자의 주요 도구 중 하나는 제네릭입니다. 즉, 단일 타입이 아닌 다양한 타입에서 작동하는 컴포넌트를 작성할 수 있습니다.

사용자는 제네릭을 통해 여러 타입의 컴포넌트나 자신만의 타입을 사용할 수 있습니다.

보충자료

인터페이스 super 사용법

선언 VS 명령

> *명령형(절차적) 프로그래밍은 당신이 어떤 일을 **어떻게** 할 것인가에 관한 것이고,*

> *선언적 프로그래밍은 당신이 **무엇을** 할 것인가에 관한 것입니다.*

즉, 의미를 해석해보면 명령형은 알고리즘은 명시하고 목표는 명시하지 않으며 선언형은 목표를 명시하고 알고리즘을 명시하지 않습니다.

이러한 선언형은 과정은 추상화의 영역으로 보내 어떻게 이러한 결과가 나오는지를 아는게 아니라 원하는 원하는 결과물이 나오도록 설계를 해놓는 것이죠. 여러분이 함수를 사용했을때 그 안에 로직을 모두 알고 사용하는게 아닌 메서드를 사용해서 나오는 결과를 알기때문에 바로 사용하실 수 있으셨죠? 그게 바로 선언형의 장점입니다. 명령형에 비해 읽기 쉽고 예측이 쉽다는 장점이 있죠

Q&A

박세민,손병진 레이서님 질문 :

```
type Parent = number | string | boolean;

1. function add<T extends Parent>(a: T, b: T): Parent {...};
2. function add<Parent>(a: Parent, b: Parent): Parent {...};
의 차이
```

A :

1번 같은 경우 Generic T에 Parent라는 타입만 들어 올 수 있도록 제약 조건을 걸어준다고 생각하시면 제네릭 변수 T에 들어올 수 있는 타입으로 Parent를 지정해준 것입니다.

TypeScript의 제네릭 함수는 **구현자**가 아니라 형식 매개변수를 지정하는 함수 **호출자** 입니다.

제네릭 타입을 지정해주실때 <> 안에 들어오는 값은 **타입**이아닌 제네릭 **변수**가 들어와야 합니다.

하지만 2번의 경우는 제네릭 변수가 들어온 것이 아닌 직접적인 타입이 들어온 경우로 이러한 경우는 제네릭을 사용하는 것이 아닌 아래와 같은 코드로 해야 올바르게 동작할 수 있습니다.

```
type Parent = number | string | boolean;

function add(a: Parent, b: Parent): Parent {
  if (typeof a === 'number' && typeof b === 'number') return a + b;
  else if (typeof a === 'string' && typeof b === 'string') return a + b;
  else return a || b;
}

console.log(add(13, 15));
console.log(add('hell', 'o'));
console.log(add(false, true));
```

참고 :

https://stackoverflow.com/questions/61648189/typescript-generic-type-parameters-t-vs-t-extends,

https://stackoverflow.com/questions/62623637/r-could-be-instantiated-with-an-arbitrary-type-which-could-be-unrelated-to-re

황규섭님 질문 :

```
type Parent = string | number | boolean;

function add<T extends Parent>(a: T, b: T): Parent {
    if (typeof a === "boolean" && typeof b === "boolean") return a || b;
    else return a + b;
}
// 순서를 바꾸면 왜 안되죠?
```

A :

TS에서는 유형 검사기에서 변수에 대한 유형을 예측할 때 a, b를 Parent 타입으로 인식하기 때문에 (string | number | boolean) + (string | number | boolean) 인식을 해서 오류를 표시합니다.

이러한 경우의 해결법으로는 내부 캐스팅을 통해서 any 타입으로 만들어주는 것입니다.

```
type Parent = string | number | boolean;

function add<T extends Parent>(a: T, b: T): Parent {
  if (typeof a === 'boolean' && typeof b === 'boolean') return a || b;
  else return <any>a + <any>b;
}
```

참고 : https://github.com/microsoft/TypeScript/issues/41449,

https://stackoverflow.com/questions/59980277/typescript-error-ts2365-operator-cannot-be-applied-to-types-string-numb,

https://github.com/microsoft/TypeScript/issues/41449