

11월 23일 (화)

1. 엘리스 SW 엔지니어 트랙 - TypeScript-

- 타입스크립트 예제 코드

```
type Box = {
  width: number;
  height: number;
  borderRadius: number;
}

let box: Box = {
  width: 200,
  height: 100,
  borderRadius: 4
}

//tuple
let x: [string, number];

x = ['x', 123];
x.push('asdf'); // 이걸 또 오류가 안난다..

// enum
enum Color {Red, Green, Blue} // 0, 1, 2 인덱스 할당
console.log(Color); // enum 은 객체, 배열의 속성을 모두 갖는다.

enum Color {Red = 1, Green, Blue} // 1, 2, 3

enum Color {Red = 1, Green = 4, Blue = 6} // 1, 4, 6

//Any

let notSure: any = () => { } // 그냥 JS와 같다. 모든 값을 넣을 수 있다. 많이 사용할 일은 없을 것

//void => 함수가 리턴하는 값이 없을 때 쓰는 것

function warmUser(temp: number): void { // 리턴값이 없는 함수
  console.log(temp);
  return temp; // 리턴을 쓸 경우 오류가 뜬다.
}
```

```

function temp(age: number): number { // 리턴값을 number로 받는 함수
    return age + 1;
}

let unusable: void = undefined; // 많이 안쓰지만 참고용

// null, undefined;

let u: undefined = undefined; // 많이 쓰이지 않음

let n: null = null; // 많이 쓰이지 않음 참고용

//never

function error(message: string): never { // 리턴값이 반환이 절대로 발생하지 않는 코드?
    throw new Error(message);
}

function infiniteLoop(): never { // 계속해서 반복하여 리턴 반환이 안되게
    while(true) {

    }
}

// 타입 별칭

let x: number = 10; // 타입 별칭 x
let xPosition: string = 'Hello';

type YesOrNo = string;
type YesOrNo = 'Y' | 'N'; // Y, N 만 받음

let answer: YesOrNo = 'Y';
let answer: YesOrNo = 'U'; // 예러발생

// 함수도 타입 별칭 설정가능
type FooFunction = () => string; // 인자도 없고 스트링을 반환하는 함수

let temp: FooFunction = ('age') => { // 인자를 넣으면 예러가 남
    return 'temp'
}

// interface 사용할 일 많다. 객체 정의시 사용

type Name = string; // 인터페이스 안에 타입별칭도 사용가능하다.

interface IUser {

```

```

    id: number;
    name: Name;
    email: string;
}

// 이렇게 타입으로만 하면 되는데 왜 인터페이스를 사용할까?
// => 인터페이스 같은 경우 값을 추가할 수 있다.
// => 타입은 값 추가가 안된다.

interface IUser { address : string };

type TUser = {
    id: number;
    name: Name;
    email: string;
}

let my: IUser = {
    id: 3,
    name: 'john',
    email: 'dfdsfasdf'
}

// Utility types

// keyof

interface User {
    id: number;
    name: string;
    age: number;
    gender: 'm' | 'f'
}

type UserKey = keyof User;
// 각 속성값인 'id' | 'name' | 'age' | 'gender' 이 출력해서 타입으로 사용

const uk: UserKey = 'id'; // 예러 x
const uk: UserKey = 'temp'; // 예러 o

// Partial<T> : 타입을 선택적으로 받고 싶을 때 사용

interface User {
    id: number; // require 속성
    name: string;
    age: number;
    gender?: 'm' | 'f'; // ? 하나만 붙여도 Partial 기능을 수행한다.
}

```

```

}

let admin: User = {
  name: 'temp', // id를 없애도 오류가 안난다.
  age: 0,
  gender: 'm'
}

let admin: Partial<User> = {
  name: 'temp', // id를 없애도 오류가 안난다.
  age: 0,
  gender: 'm'
}

// Readonly<T> : 읽기 전용

interface User {
  id: number; // require 속성
  name: string;
  age: number;
  gender: 'm'|'f';
}

let admin: Readonly<User> = {
  id: 0,
  name: 'temp',
  age: 0,
  gender: 'm'
}

admin.id = 3; // 바꿀 수 없다는 오류가 난다.

// 축약형?
interface User {
  id: number; // require 속성
  name: string;
  age: number;
  readonly gender: 'm'|'f'; // ? 와 같이 readonly 를 사용할 수 있다.
}

let admin: User = {
  id: 0,
  name: 'temp',
  age: 0,
  gender: 'm'
}

admin.id = 3; // 바꿀 수 없다는 오류가 난다.

```

```

// Record<K,T> K는 key 키값과 타입값 리턴값 모두를 제어한다.

type Grade = '1' | '2' | '3' | '4';
type Score = 'A' | 'B' | 'C' | 'D';

const score: Record<Grade, Score> = { // Score 대신 string 을 넣으면 모든 스트링 입력 가능
  1: 'A',
  2: 'B',
  3: 'C',
  4: 'D' // 'E' 를 넣을 경우 에러
}

interface User {
  id: number;
  name: string;
  age: number;
}

function isValid(user: User) {
  const result: Record<keyof User, boolean> = {
    id: user.id > 0,
    name: user.name !== '',
    age: user.age > 0
  }

  return result;
}

// Pick<T,K>

interface User {
  id: number;
  name: string;
  age: number;
  gender: 'm' | 'f';
}

const admin: Pick<User, 'id' | 'name'> = { // id, name 만 뽑아서 쓰는 것
  id: 0,
  name: 'Bob',
  // age: 3   뽑지 않았기 때문에 오류가 난다.
}

// Omit<T,K>. Pick의 반대 개념

interface User {
  id: number;
  name: string;
  age: number;
  gender: 'm' | 'f';
}

```

```

}

const admin: Omit<User, 'age'|'gender'> = { // age, gender 만 생략하고 쓰는 것
  id: 0,
  name: 'Bob',
  // age: 3    뽑지 않았기 때문에 오류가 난다.
}

// Exclude<T1, U>   타입을 제외시키는 유틸리티

type T1 = string | number | boolean;
type T2 = Exclude<T1, number | string>

let isNum: T2 = 3; // 에러가 난다.
let isBool: T2 = true; // 에러가 안 난다.


// NonNullable<T>   // 타입에서 null과 undefined를 제거하는 유틸리티

type T1 = string | null | undefined | void;

type T2 = NonNullable<T1>; // null 과 undefined 가 빠진 타입이 된다.

let empty: T2 = null; // 에러가 난다.
let product: T2 = 'STR'; // 잘 작동


// Parameters<T> 함수로 지정한 타입을 인자로 받아서 튜플 타입으로 리턴

type T0 = Parameters<() => string>; // 인자값이 없다. 빈 배열을 반환 []
type T1 = Parameters<(s: string) => void>; // [string] 리턴

let T1Arr: T1 = [1]; // 에러
let T1Arr: T1 = ['1']; // 에러x

type temp = (s: string, i: number)=> number;

type T2 = Parameters<temp>; // [string, number] 반환

let T2Arr: T2 = ['1', 1];


// ReturnType<T> // 리턴 타입은 튜플배열과 같은 것을 반환하지 않는다. 리턴은 하나만 리턴값

type T0 = ReturnType<() => string>; // string

let str: T0 = ''; // 에러 x

```

```

type T7 = ReturnType<string>; // 에러가 뜬다.
type T8 = ReturnType<any>; // 에러가 안 뜬다. // any 리턴
type T9 = ReturnType<never>; // 에러가 안 뜬다. // boolean은 안되는 any

let temp: T9 = true;

// Required<T> 모든 속성을 필수값으로 바꾼다.

interface User {
  id: number; // 디폴트로 모두 Required
  name?: string; // Required X
}

let admin: Required<User> = {
  id: 1,
  name: 'kim' // name 또한 Required 된다.
}

```

• 수업중 질문



void랑 never타입의 차이가 뭔가요?



@이지윤 void랑 never타입의 차이가 뭔가요?



[코치] 김병철 오늘 오후 2:14

void: 반환값이 없는 함수입니다. (그냥 출력하거나, 상태를 바꿀 때 자주 씁니다.)
never: 함수가 종료되지 않습니다. (무한루프를 돌거나, Error를 띄울 때 사용합니다.)



김영우 오늘 오후 2:26

lastName 에 undefined 대신 null 를 넣으면 오류가 생기는 이유는 무엇인가요?!
감사합니다



@안영우

lastName 에 undefined 대신 null 를 넣으면 오류가 생기는 이유는 무엇인가요?!

[코치] 김병철 오늘 오후 2:31

undefined가 넘어가면 lastName이 없다고 판단되지만, null이 들어가면 lastName이 있다고 판단을 합니다.
(undefined는 값 자체가 없다고 판단하나, null은 값은 있는데 의미없는 특별한 값이 들어있다고 보시면 좋을 것 같아요.)
+ undefined는 미리 선언된 global variable이나, null은 키워드입니다.

- python3 - deque 사용법

list와 비슷한 deque

deque의 사용법을 잠시 살펴보자.

```
>>> from collections import deque
>>> d = deque([1,2,3,4,5])
>>> d.append(6)
>>> d
deque([1, 2, 3, 4, 5, 6])
>>> d.appendleft(0)
>>> d
deque([0, 1, 2, 3, 4, 5, 6])
>>> d.pop()
6
>>> d
deque([0, 1, 2, 3, 4, 5])
>>> d.popleft()
0
>>> d
deque([1, 2, 3, 4, 5])
>>>
```