2025.01.13.화

생성일	@2026년 1월 13일 오전 9:11
태그	

목차

▼ 1) TypeScript 기초

- 변수 초기값, 반환값, 연산 결과를 보고 타입을 자동으로 결정
  - 잘못된 타입은 컴파일 에러 발생 → 안정적인 실행, 유지보수 비용 감소
- let은 값이 바뀔 수 있어 보통 넓게 추록, const는 더 좁게(리터럴 쪽) 추론되는 경향
  - let a = “test” → String
  - const a = “test” → ‘test’라는 리터럴 타입
- 함수는 매개변수 타입이 없다면 기본적으로 any, ts를 안쓰겠다는 의미 (타입 추론x)
  - 잠재적 위험이 있어서 보통 명시하거나 제네릭으로 유도
- Interface
  - 객체/클래스 형태 정의 설계, implements, extends로 확장 쉬움
  - 동일 이름 선언이 합쳐지는 특성: 선언 병합
  - 컴파일 후 제거

```
interface User {
    name: string
    age: number
    email?: string
}

let userA: User = {
    name: Kim
    age: 28
    email: sang@exmaple.com
}
```

- Type
  - 유니온/인터섹션, 리터럴, 튜플, 조건부 타입 등 표현력이 넓음
  - 객체 뿐 아니라 모든 타입 표현식 가능
  - 선업 병합 불가
  - 객체 타입이면 implements, extends 가능

```
type User {
    name: string
    age: number
    email?: string
}

// 객체 외 타입 선언
type ID = number | string;
```

```
type Point = [number, number];
type Callback = (value: string) => void;
```

- 두 타입의 합집합 타입 선언: `|` (vertical bar) → 하나만 만족
- 두 타입의 교집합 타입 선언: `&` (ampercent) → 모두 만족

```
type A = {
  name: string;
  age: number;
};

type B = {
  name: string;
  email: string;
};

let user1: A | B = {
  name: 'kim',
  age: 30,
};
let user2: A & B = {
  name: 'lee',
  age: 20,
  email: 'lee@example.com',
};
```

- 리터럴 타입: 값 하나를 그대로 타입으로 고정, 안정성 향상

```
type OnlyKim = "kim";

let a : OnlyKim = "kim"; // OK
a = "KIM" // compile error
```

- never: 공집합으로 가질 수 있는 값이 없음을 의미
  - T|nerver → T, T&never → never
- 할당 가능성: 좁은 타입이 넓은 타입에 들어감

```
let wide: string;
let narrow: "kim" = "kim";

wide = narrow; // wide의 값은 "kim", 타입은 string
```

- Extract<T, U>: T타입에서 U에 할당 가능한 것만 남김

```
type A = "a" | "b" | "c";
type B = "b" | "d";

type R = Extract<A, B>; // "b"
```

- Control-flow Analysis
  - 타입 스크립트는 현재 분기에서 가능한 값으로 계속 갱신된다
  - 코드 흐름에 따라 타입을 좁힘
  - 타입 좁히기: if/else, switch, return'throw, &&/||
  - 타입 가드: typeof, instanceof, in, 리터럴 판별

```
function f(x: string | number) {
    if (typeof x === "string") {
        return x.toUpperCase();
    }
    return x.toFixed(2);
}
```

- unknown 타입: 어떤 값이든 받을 수 있지만 타입 좁히기/검증 후에만 사용 가능
    - any는 검증 없이 사용 가능

▼ 2) Generic

- 타입을 매개변수로 받아 재사용하는 방식
- 특정 타입에 고정되지 않고 "같은 T가 흐른다"는 일반화 제약
- 타입 안정성 확보

```
// 타입 정의 없음 (any)
function first(arr: any[]) {
  return arr[0];
}

// 제네릭
function second<T>(arr: T[]): T | undefined {
    return arr[0];
}

// <T>라는 제네릭 파라미터, arr는 T라는 타입으로 이루어짐, 반환 타입은 T
```

- 조건부 타입 (Conditional Types)

```
type StringToLength<T> = T extends string ? number : T
// T가 string에 할당 가능하면 number 반환, 아니면 T
```

- 매핑 타입 (Mapped Types)

```
type Nullable<T> = { K in keyof T] : T[K] | null };

type User = { id: number; name: string };
type UserOrNull = Nullable<User>;
```

▼ 3) Utility Types

- Partial<T>
    - 모든 프로퍼티를 선택으로 바꾸는 유틸 → 타입 변수 뒤에 ? 붙이는 거랑 같음
    - 얕은 변환으로 객체 내부까지 바꾸진 않음

```
type Post = {
    id: number;
    title: string;
    body: string;
};

const post1: Post = { id: 1 }; // compile error, title missing, body missing
const draft: Partial<Post> = {};
draft.title = "임시 제목";
```

- Readonly<T>
    - 읽기 전용으로 바꿔 수정 불가를 보장하는 타입

    - 얕은 변환으로 중첩 객체 내부는 수정 가능

```
type User = {
    id: number
    name: string
};

const base: User = { id: 1, name: "kim" };
const ro: Readonly<User> = base;

ro.name = "park"; // compile error

base.name = "Lee"; // OK, mutable 참조
console.log(ro.name); // Lee
```

- Pick<T, k>: 객체 타입에서 지정한 키 K만 골라서 새로운 타입을 만든다
- Omit<T, k>: 객체 타입에서 지정한 키만 제거된 새로운 타입을 만든다
- Exclude<T, U>: 유니온 타입 T 중에서 U에 해당하는 타입만 추출
- Extract<T, U>: 유니온 타입 T 중에서 U에 해당하는 타입만 추출