

기본 타입 (Basic Types)

- 원시 타입: `string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`
- 배열: `number[]` 또는 `Array<number>`
- 튜플 (Tuple): 고정 길이 및 타입 배열

```
let user: [number, string] = [1, "Alice"];
let optionalTuple: [string, number?] = ["Bob"];
let restTuple: [string, ...number[]] = ["Scores", 90, 85];
```
- 열거형 (Enum):

```
enum Direction { Up = 1, Down, Left, Right }
const enum FastStatus { Ready, Done } // 인라인 컴파일
```
- 특수 타입:
 - `any`: 모든 타입 허용 (타입 검사 비활성화)
 - `unknown`: 알 수 없는 타입 (타입 좁히기 전까지 연산 불가)
 - `never`: 절대 발생하지 않는 값의 타입 (예러 던짐, 무한 루프)
 - `void`: 반환값이 없는 함수의 반환 타입

인터페이스와 타입 별칭 (Interface vs Type)

Interface (확장 및 객체 정의)

```
interface User {
  readonly id: number;
  name: string;
  age?: number; // 선택적 속성
}
```

```
// 상속 및 병합(Declaration Merging)
interface Admin extends User {
  role: string;
}
```

Type Alias (유니온, 원시 타입, 튜플)

```
type ID = string | number;
type Point = { x: number; y: number };
type Callback = (data: string) => void;
type Status = "idle" | "loading" | "success" | "error";
```

유니온 및 인터섹션 (Union & Intersection)

- Union (|)**: 여러 타입 중 하나

```
function printId(id: string | number) { ... }
```
- Intersection (&)**: 여러 타입을 하나로 결합

```
type Person = { name: string };
type Employee = Person & { employeeId: number };
```

타입 좁히기 (Type Narrowing)

런타임 조건문을 통해 더 구체적인 타입으로 추론합니다.

- `typeof` 검사: `typeof x === "string"`
- `instanceof` 검사: `err instanceof CustomError`
- `in` 연산자: `"role" in account`
- 구별된 유니온 (Discriminated Unions):

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; size: number };
```

```
function getArea(s: Shape): number {
  switch (s.kind) {
    case "circle": return Math.PI * s.radius ** 2;
    case "square": return s.size * s.size;
  }
}
```

- 사용자 정의 타입 가드:

```
function isString(val: unknown): val is string {
  return typeof val === "string";
}
```

제네릭 (Generics)

재사용 가능한 컴포넌트와 함수를 위한 타입 매개변수.

```
function identity<T>(arg: T): T {
  return arg;
}
```

```
// 제네릭 인터페이스 & 기본값
interface ApiResponse<T = any> {
  code: number;
}
```

```
data: T;
}
```

```
// 제네릭 제약조건 (Constraints)
interface Lengthwise { length: number; }
function logLength<T extends Lengthwise>(item: T): number {
  return item.length;
}
```

유틸리티 타입 (Utility Types)

TypeScript에 내장된 강력한 변환 도구 모음.

- `Partial<T>`: 모든 속성을 선택적(?)으로 변경
- `Required<T>`: 모든 속성을 필수로 변경
- `Readonly<T>`: 모든 속성을 읽기 전용으로 변경
- `Record<K, T>`: 키 K와 값 T로 구성된 객체 타입 생성
- `Pick<T, K>`: T에서 특정 키 K만 추출
- `Omit<T, K>`: T에서 특정 키 K를 제외
- `Exclude<T, U>`: 유니온 T에서 U에 할당 가능한 타입 제외
- `Extract<T, U>`: 유니온 T에서 U에 할당 가능한 타입 추출
- `Nullable<T>`: `null`과 `undefined` 제거
- `ReturnType<T>`: 함수 타입 T의 반환 타입 추출
- `Parameters<T>`: 함수 타입 T의 매개변수 튜플 추출
- `Awaited<T>`: `Promise<T>`의 해결된 값 타입 추출

고급 타입 및 연산자

- `keyof` 연산자: 객체 타입의 모든 키 유니온 반환

```
type UserKeys = keyof User; // "id" | "name" | "age"
```
- 인덱스 접근 타입 (Indexed Access):

```
type UserName = User["name"]; // string
```
- 조건부 타입 (Conditional Types):

```
type IsString<T> = T extends string ? true : false;
```
- 템플릿 리터럴 타입:

```
type Event = `on${"Click" | "Hover"}`; // "onClick" | "onHover"
```
- `satisfies` 연산자: 타입 검사를 수행하되 더 구체적인 추론 유지

```
const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
}
```

```
} satisfies Record<string, string | number[]>;
```

클래스 및 접근 제어자

```
class Animal {
  public name: string;
  protected age: number;
  #internalId: string; // ES Private 필드
}
```

```
// 생성자 단축 프로퍼티 선언
constructor(public readonly breed: string, age: number) {
  this.name = breed;
  this.age = age;
  this.#internalId = Math.random().toString();
}
```

tsconfig.json 핵심 설정

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```