# TypeScript 완벽 가이드

## 목차

1. [TypeScript란?](#typescript란)

2. [왜 TypeScript를 사용하는가?](#왜-typescript를-사용하는가)

3. [기본 타입](#기본-타입)

4. [고급 타입](#고급-타입)

5. [함수](#함수)

6. [클래스와 인터페이스](#클래스와-인터페이스)

7. [제네릭](#제네릭)

8. [모듈](#모듈)

9. [타입 가드](#타입-가드)

10. [실전 예제](#실전-예제)

---

## TypeScript란?

TypeScript는 Microsoft에서 개발한 \*\*정적 타입 시스템을 제공하는 JavaScript의 상위 집합(superset)\*\*입니다. TypeScript는 JavaScript로 컴파일되며, 모든 유효한 JavaScript 코드는 TypeScript 코드이기도 합니다.

### 주요 특징

- \*\*정적 타입 검사\*\*: 컴파일 시점에 타입 오류를 발견

- \*\*점진적 타입 적용\*\*: 기존 JavaScript 코드에 타입을 점진적으로 추가 가능

- \*\*엄격한 타입 추론\*\*: 자동으로 타입을 추론하여 개발자 경험 향상

- \*\*최신 ECMAScript 기능 지원\*\*: 향후 JavaScript 기능들을 미리 사용 가능

---

## 왜 TypeScript를 사용하는가?

### 1. 타입 안정성

```typescript

*// JavaScript (런타임 에러)*

function add(*a*, *b*) {

    return a + b;

}

add(1, 2);      *// 3*

add("1", "2");  *// "12" (의도하지 않은 결과!)*

add(null, 2);   *// 런타임 에러!*

*// TypeScript (컴파일 시점 에러)*

function add(*a*: number, *b*: number): number {

    return a + b;

}

add(1, 2);      *// 3*

add("1", "2");  *// ❌ 컴파일 에러!*

add(null, 2);   *// ❌ 컴파일 에러!*

```

### 2. 더 나은 IDE 지원

- 자동 완성

- 리팩토링 지원

- 인라인 문서

- 실시간 오류 감지

### 3. 자동 문서화

타입이 곧 문서화 역할을 하므로, 별도의 주석 없이도 코드를 이해할 수 있습니다.

### 4. 리팩토링 안정성

대규모 프로젝트에서 코드를 안전하게 변경할 수 있습니다.

---

## 기본 타입

### 원시 타입 (Primitive Types)

```typescript

*// 숫자*

let age: number = 27;

let pi: number = 3.14159;

*// 문자열*

let name: string = "홍길동";

let greeting: string = `안녕하세요, ${name}님!`;

*// 불리언*

let isActive: boolean = true;

let isComplete: boolean = false;

*// null과 undefined*

let nullValue: null = null;

let undefinedValue: undefined = undefined;

```

### 배열

```typescript

*// 두 가지 선언 방식*

let numbers: number[] = [1, 2, 3, 4];

let names: Array<string> = ["Alice", "Bob", "Charlie"];

*// 튜플 (고정 길이 배열)*

let tuple: [string, number] = ["홍길동", 27];

let [name, age] = tuple;

```

### 객체

```typescript

*// 명시적 타입*

let user: { name: string; age: number } = {

    name: "홍길동",

    age: 27

};

*// Optional 속성*

let user2: {

    name: string;

    age?: number  *// ?는 선택적 속성*

} = {

    name: "이순신"

};

*// Readonly 속성*

let config: {

    readonly apiUrl: string;

    timeout: number;

} = {

    apiUrl: "https://api.example.com",

    timeout: 5000

};

*// config.apiUrl = "new url";  // ❌ 에러!*

```

### any와 unknown

```typescript

*// any: 모든 타입 허용 (타입 체크 비활성화)*

let anyValue: any = "문자열";

anyValue = 42;

anyValue = true;

*// ⚠️ any 사용을 자제하고 unknown 사용 권장*

*// unknown: 안전한 any*

let unknownValue: unknown = "문자열";

*// unknownValue.toUpperCase();  // ❌ 에러! 타입 체크 필요*

if (typeof unknownValue === "string") {

    unknownValue.toUpperCase();  *// ✅ OK*

}

```

### void와 never

```typescript

*// void: 반환값이 없는 함수*

function log(*message*: string): void {

    console.log(message);

}

*// never: 절대 반환하지 않는 함수*

function throwError(*message*: string): never {

    throw new Error(message);

*// 함수가 끝나지 않으므로 never 반환*

}

function infiniteLoop(): never {

    while (true) {

*// 무한 루프*

    }

}

```

---

## 고급 타입

### 유니온 타입 (Union Types)

```typescript

*// 여러 타입 중 하나*

let id: string | number = "123";

id = 123;  *// ✅ OK*

function processId(*id*: string | number) {

    if (typeof id === "string") {

        return id.toUpperCase();

    } else {

        return id.toString();

    }

}

```

### 교차 타입 (Intersection Types)

```typescript

*// 여러 타입의 조합*

type Draggable = { x: number; y: number };

type Resizable = { width: number; height: number };

type Window = Draggable & Resizable;

let window: Window = {

    x: 10,

    y: 20,

    width: 100,

    height: 200

};

```

### 리터럴 타입

```typescript

*// 특정 값만 허용*

let direction: "left" | "right" | "up" | "down" = "left";

direction = "right";  *// ✅ OK*

*// direction = "diagonal";  // ❌ 에러!*

*// 숫자 리터럴*

type Dice = 1 | 2 | 3 | 4 | 5 | 6;

let roll: Dice = 4;

*// 불리언 리터럴*

type Success = true;

let isSuccess: Success = true;

*// let isFailure: Success = false;  // ❌ 에러!*

```

### 타입 별칭 (Type Alias)

```typescript

type User = {

    id: string;

    name: string;

    email: string;

    age?: number;

};

function getUser(*id*: string): User {

    return {

        id,

        name: "홍길동",

        email: "hong@example.com"

    };

}

*// 유니온 타입 별칭*

type Status = "pending" | "approved" | "rejected";

*// 제네릭 타입 별칭*

type Result<T> = { success: boolean; data: T };

type ApiResponse<T> = { status: number; body: T };

```

---

## 함수

### 함수 타입 선언

```typescript

*// 일반 함수*

function multiply(*x*: number, *y*: number): number {

    return x \* y;

}

*// 화살표 함수*

const divide = (*x*: number, *y*: number): number => {

    return x / y;

};

*// 함수 타입*

type Operation = (*x*: number, *y*: number) => number;

const add: Operation = (*x*, *y*) => x + y;

const subtract: Operation = (*x*, *y*) => x - y;

```

### 선택적 매개변수

```typescript

function greet(*name*: string, *greeting*?: string): string {

    return `${greeting || "안녕하세요"}, ${name}님!`;

}

greet("홍길동");              *// "안녕하세요, 홍길동님!"*

greet("홍길동", "안녕!");      *// "안녕!, 홍길동님!"*

*// 기본 매개변수*

function createUser(

*name*: string,

*age*: number = 0,

*isActive*: boolean = true

) {

    return { name, age, isActive };

}

```

### 나머지 매개변수

```typescript

function sum(...*numbers*: number[]): number {

    return numbers.reduce((*acc*, *num*) => acc + num, 0);

}

sum(1, 2, 3);           *// 6*

sum(1, 2, 3, 4, 5);     *// 15*

*// 고정 파라미터와 나머지 파라미터 결합*

function greetMany(*greeting*: string, ...*names*: string[]): string {

    return names.map(*name* => `${greeting}, ${name}!`).join(" ");

}

greetMany("안녕", "홍길동", "이순신", "김철수");

*// "안녕, 홍길동! 안녕, 이순신! 안녕, 김철수!"*

```

### 함수 오버로딩

```typescript

*// 오버로드 시그니처*

function process(*value*: string): string;

function process(*value*: number): number;

function process(*value*: boolean): boolean;

*// 구현 시그니처*

function process(*value*: string | number | boolean): string | number | boolean {

    if (typeof value === "string") {

        return value.toUpperCase();

    } else if (typeof value === "number") {

        return value \* 2;

    } else {

        return !value;

    }

}

let result1: string = process("hello");   *// "HELLO"*

let result2: number = process(5);         *// 10*

let result3: boolean = process(true);    *// false*

```

---

## 클래스와 인터페이스

### 클래스

```typescript

class Person {

*// public, private, protected 접근 제한자*

    private id: string;

    public name: string;

    protected age: number;

    constructor(*name*: string, *age*: number) {

        this.id = Math.random().toString(36).substr(2, 9);

        this.name = name;

        this.age = age;

    }

*// 메서드*

    introduce(): string {

        return `안녕하세요, 저는 ${this.name}입니다. 나이는 ${this.age}세입니다.`;

    }

*// getter*

    get getId(): string {

        return this.id;

    }

}

const person = new Person("홍길동", 27);

console.log(person.name);      *// "홍길동"*

*// console.log(person.age);    // ❌ 에러! protected는 외부 접근 불가*

*// console.log(person.id);     // ❌ 에러! private은 외부 접근 불가*

console.log(person.getId);     *// 랜덤 ID*

```

### 상속

```typescript

class Employee extends Person {

    private department: string;

    constructor(*name*: string, *age*: number, *department*: string) {

*super*(name, age);

        this.department = department;

    }

    introduce(): string {

*// super로 부모 메서드 호출*

        return `${*super*.introduce()} ${this.department} 부서에서 일하고 있습니다.`;

    }

}

const employee = new Employee("김철수", 30, "개발");

console.log(employee.introduce());

```

### 추상 클래스

```typescript

abstract class Animal {

    protected name: string;

    constructor(*name*: string) {

        this.name = name;

    }

*// 추상 메서드: 하위 클래스에서 반드시 구현*

    abstract makeSound(): string;

*// 일반 메서드*

    getName(): string {

        return this.name;

    }

}

class Dog extends Animal {

    makeSound(): string {

        return "멍멍!";

    }

}

class Cat extends Animal {

    makeSound(): string {

        return "야옹!";

    }

}

*// let animal = new Animal("동물");  // ❌ 에러! 추상 클래스는 인스턴스화 불가*

let dog = new Dog("바둑이");

console.log(dog.makeSound());  *// "멍멍!"*

```

### 인터페이스

```typescript

*// 인터페이스 정의*

interface User {

    id: string;

    name: string;

    email: string;

    age?: number;

    readonly createdAt: Date;

}

*// 인터페이스 구현*

class RegisteredUser implements User {

    id: string;

    name: string;

    email: string;

    age?: number;

    readonly createdAt: Date;

    constructor(*name*: string, *email*: string) {

        this.id = Math.random().toString(36).substr(2, 9);

        this.name = name;

        this.email = email;

        this.createdAt = new Date();

    }

}

*// 인터페이스를 타입으로 사용*

function displayUser(*user*: User): void {

    console.log(`ID: ${user.id}, 이름: ${user.name}, 이메일: ${user.email}`);

}

*// 함수 타입 인터페이스*

interface MathOperation {

    (*x*: number, *y*: number): number;

}

const add: MathOperation = (*x*, *y*) => x + y;

const multiply: MathOperation = (*x*, *y*) => x \* y;

*// 인덱스 시그니처*

interface Dictionary<T> {

    [*key*: string]: T;

}

let scoreDict: Dictionary<number> = {

    "홍길동": 95,

    "이순신": 88,

    "김철수": 92

};

*// 선택적 속성*

interface Config {

    apiUrl: string;

    timeout?: number;

    retry?: number;

}

let config: Config = {

    apiUrl: "https://api.example.com"

*// timeout과 retry는 선택적*

};

```

### 인터페이스 확장

```typescript

interface Flyable {

    fly(): void;

}

interface Swimmable {

    swim(): void;

}

interface Bird extends Flyable {

    name: string;

}

interface Duck extends Flyable, Swimmable {

    name: string;

}

class Eagle implements Bird {

    name: string;

    constructor(*name*: string) {

        this.name = name;

    }

    fly(): void {

        console.log(`${this.name}이(가) 날아갑니다!`);

    }

}

class Mallard implements Duck {

    name: string;

    constructor(*name*: string) {

        this.name = name;

    }

    fly(): void {

        console.log(`${this.name}이(가) 날아갑니다!`);

    }

    swim(): void {

        console.log(`${this.name}이(가) 헤엄칩니다!`);

    }

}

```

---

## 제네릭

### 제네릭 함수

```typescript

*// 제네릭 함수: 타입을 파라미터로 받음*

function identity<T>(*arg*: T): T {

    return arg;

}

let output1 = identity<string>("hello");  *// 타입 명시*

let output2 = identity("world");         *// 타입 추론*

*// 여러 타입 파라미터*

function pair<T, U>(*first*: T, *second*: U): [T, U] {

    return [first, second];

}

let result = pair<string, number>("홍길동", 27);

*// result는 [string, number] 타입*

*// 제네릭 제약*

interface Lengthwise {

    length: number;

}

function logLength<T extends Lengthwise>(*arg*: T): T {

    console.log(arg.length);

    return arg;

}

logLength({ length: 10, value: "hello" });

logLength("hello");

*// logLength(123);  // ❌ 에러! length 속성이 없음*

```

### 제네릭 클래스

```typescript

class Box<T> {

    private value: T;

    constructor(*value*: T) {

        this.value = value;

    }

    getValue(): T {

        return this.value;

    }

    setValue(*value*: T): void {

        this.value = value;

    }

}

let numberBox = new Box<number>(42);

console.log(numberBox.getValue());  *// 42*

let stringBox = new Box<string>("hello");

console.log(stringBox.getValue());  *// "hello"*

```

### 제네릭 인터페이스

```typescript

interface Repository<T> {

    save(*entity*: T): void;

    findById(*id*: string): T | null;

    findAll(): T[];

}

class UserRepository implements Repository<User> {

    private users: User[] = [];

    save(*user*: User): void {

        const index = this.users.findIndex(*u* => u.id === user.id);

        if (index !== -1) {

            this.users[index] = user;

        } else {

            this.users.push(user);

        }

    }

    findById(*id*: string): User | null {

        return this.users.find(*u* => u.id === id) || null;

    }

    findAll(): User[] {

        return [...this.users];

    }

}

```

### 제네릭 유틸리티 타입

```typescript

*// Partial: 모든 속성을 선택적으로 만들기*

interface User {

    id: string;

    name: string;

    email: string;

}

type PartialUser = Partial<User>;

*// { id?: string; name?: string; email?: string; }*

*// Required: 모든 속성을 필수로 만들기*

type RequiredUser = Required<PartialUser>;

*// Readonly: 모든 속성을 읽기 전용으로 만들기*

type ReadonlyUser = Readonly<User>;

*// Pick: 특정 속성만 선택*

type UserNameOnly = Pick<User, "name">;

*// { name: string; }*

*// Omit: 특정 속성 제외*

type UserWithoutEmail = Omit<User, "email">;

*// { id: string; name: string; }*

*// Record: 키-값 타입 매핑*

type UserRecord = Record<string, User>;

```

---

## 모듈

### 내보내기 (Export)

```typescript

*// named export*

export function add(*x*: number, *y*: number): number {

    return x + y;

}

export const multiply = (*x*: number, *y*: number): number => x \* y;

export class Calculator {

    subtract(*x*: number, *y*: number): number {

        return x - y;

    }

}

*// default export*

export default function hello(*name*: string): string {

    return `Hello, ${name}!`;

}

*// 모든 것을 한 번에 내보내기*

export { add, multiply, Calculator };

```

### 가져오기 (Import)

```typescript

*// named import*

import { add, multiply, Calculator } from "./math";

*// default import*

import hello from "./greeting";

*// 모든 것을 가져오기*

import \* as Math from "./math";

*// 별칭으로 가져오기*

import { add as sum } from "./math";

*// 타입만 가져오기*

import type { User, Config } from "./types";

*// 타입과 값 모두 가져오기*

import { type User, getUser } from "./user";

```

---

## 타입 가드

### typeof 타입 가드

```typescript

function process(*value*: string | number) {

    if (typeof value === "string") {

*// 이 블록에서 value는 string 타입*

        return value.toUpperCase();

    } else {

*// 이 블록에서 value는 number 타입*

        return value \* 2;

    }

}

```

### instanceof 타입 가드

```typescript

class Dog {

    bark(): void {

        console.log("멍멍!");

    }

}

class Cat {

    meow(): void {

        console.log("야옹!");

    }

}

function makeSound(*animal*: Dog | Cat) {

    if (animal instanceof Dog) {

        animal.bark();  *// 타입이 Dog로 좁혀짐*

    } else {

        animal.meow();  *// 타입이 Cat으로 좁혀짐*

    }

}

```

### in 연산자 타입 가드

```typescript

interface Car {

    drive(): void;

}

interface Boat {

    sail(): void;

}

function move(*vehicle*: Car | Boat) {

    if ("drive" in vehicle) {

        vehicle.drive();  *// Car 타입*

    } else {

        vehicle.sail();   *// Boat 타입*

    }

}

```

### 사용자 정의 타입 가드

```typescript

interface Fish {

    swim(): void;

}

interface Bird {

    fly(): void;

}

function isFish(*animal*: Fish | Bird): *animal* is Fish {

    return (animal as Fish).swim !== undefined;

}

function move(*animal*: Fish | Bird) {

    if (isFish(animal)) {

        animal.swim();  *// Fish 타입으로 좁혀짐*

    } else {

        animal.fly();   *// Bird 타입으로 좁혀짐*

    }

}

```

---

## 실전 예제

### API 클라이언트 예제

```typescript

*// 타입 정의*

interface ApiResponse<T> {

    success: boolean;

    data: T;

    message?: string;

}

interface User {

    id: string;

    name: string;

    email: string;

}

*// API 클라이언트*

class ApiClient {

    private baseUrl: string;

    constructor(*baseUrl*: string) {

        this.baseUrl = baseUrl;

    }

    async get<T>(*endpoint*: string): Promise<ApiResponse<T>> {

        const response = await fetch(`${this.baseUrl}${endpoint}`);

        return response.json();

    }

    async post<T>(*endpoint*: string, *data*: unknown): Promise<ApiResponse<T>> {

        const response = await fetch(`${this.baseUrl}${endpoint}`, {

            method: "POST",

            headers: { "Content-Type": "application/json" },

            body: JSON.stringify(data)

        });

        return response.json();

    }

}

*// 사용 예제*

const api = new ApiClient("https://api.example.com");

async function getUser(*id*: string) {

    const result = await api.get<User>(`/users/${id}`);

    if (result.success) {

        console.log(result.data.name);

    }

}

```

### React 컴포넌트 예제

```typescript

import React, { useState, useEffect } from "react";

interface User {

    id: number;

    name: string;

    email: string;

}

interface UserListProps {

    userId?: number;

}

const UserList: React.FC<UserListProps> = ({ *userId* }) => {

    const [users, setUsers] = useState<User[]>([]);

    const [loading, setLoading] = useState<boolean>(false);

    useEffect(() => {

        async function fetchUsers() {

            setLoading(true);

            try {

                const response = await fetch("/api/users");

                const data: User[] = await response.json();

                setUsers(data);

            } catch (error) {

                console.error("Failed to fetch users:", error);

            } finally {

                setLoading(false);

            }

        }

        fetchUsers();

    }, [userId]);

    if (loading) return <div>Loading...</div>;

    return (

        <ul>

            {*users*.*map*(*user* => (

                <li key={user.id}>

                    {*user*.*name*} ({user.email})

                </li>

            ))}

        </ul>

    );

};

export default UserList;

```

### 이벤트 시스템 예제

```typescript

*// 이벤트 타입 정의*

type EventMap = {

    "user:login": { userId: string; timestamp: Date };

    "user:logout": { userId: string };

    "data:update": { data: unknown };

};

*// 이벤트 시스템*

class EventEmitter<T extends Record<string, unknown>> {

    private listeners: { [K in keyof T]?: ((*data*: T[K]) => void)[] } = {};

    on<K extends keyof T>(*event*: K, callback: (*data*: T[K]) => void): void {

        if (!this.listeners[event]) {

            this.listeners[event] = [];

        }

        this.listeners[event]!.push(callback);

    }

    emit<K extends keyof T>(*event*: K, *data*: T[K]): void {

        const callbacks = this.listeners[event];

        if (callbacks) {

            callbacks.forEach(*callback* => callback(data));

        }

    }

}

*// 사용 예제*

const emitter = new EventEmitter<EventMap>();

emitter.on("user:login", (*data*) => {

    console.log(`User ${data.userId} logged in at ${data.timestamp}`);

});

emitter.emit("user:login", {

    userId: "123",

    timestamp: new Date()

});

```

---

## TypeScript 설정 (tsconfig.json)

```json

{

  "compilerOptions": {

*// 모듈 시스템*

    "target": "ES2020",

    "module": "ESNext",

    "lib": ["ES2020", "DOM"],

    "moduleResolution": "node",

*// 출력 설정*

    "outDir": "./dist",

    "rootDir": "./src",

    "sourceMap": true,

*// 엄격한 타입 체크*

    "strict": true,

    "noImplicitAny": true,

    "strictNullChecks": true,

    "strictFunctionTypes": true,

    "strictBindCallApply": true,

    "strictPropertyInitialization": true,

*// 추가 체크*

    "noUnusedLocals": true,

    "noUnusedParameters": true,

    "noImplicitReturns": true,

    "noFallthroughCasesInSwitch": true,

*// 모듈 해석*

    "esModuleInterop": true,

    "allowSyntheticDefaultImports": true,

    "resolveJsonModule": true,

*// 증분 컴파일*

    "incremental": true,

    "tsBuildInfoFile": "./.tsbuildinfo"

  },

  "include": ["src/\*\*/\*"],

  "exclude": ["node\_modules", "dist"]

}

```

---

## 결론

TypeScript는 JavaScript에 타입 안정성을 추가하여 대규모 애플리케이션 개발을 더 안전하고 효율적으로 만듭니다.

### TypeScript의 핵심 가치

1. \*\*타입 안정성\*\*: 컴파일 시점에 오류 발견

2. \*\*개발자 경험\*\*: 자동 완성, 리팩토링, 문서화

3. \*\*점진적 적용\*\*: 기존 JavaScript 프로젝트에 점진적 도입 가능

4. \*\*최신 기술\*\*: 향후 JavaScript 기능을 미리 사용

### 학습 순서 권장사항

1. 기본 타입과 변수 선언

2. 함수와 매개변수

3. 인터페이스와 클래스

4. 제네릭

5. 고급 타입 (유니온, 인터섹션, 타입 가드)

6. 모듈 시스템

7. 실전 프로젝트 경험

---

## 참고 자료

- [TypeScript 공식 문서](https://www.typescriptlang.org/)

- [TypeScript Deep Dive](https://basarat.gitbook.io/typescript/)

- [TypeScript Handbook](https://www.typescriptlang.org/docs/handbook/intro.html)

---

\*\*마지막 업데이트\*\*: 2024년