

Complete TypeScript Guide - From Zero to 7/10

Table of Contents

- 1. Basic Types
- 2. Interfaces vs Types
- 3. Functions
- 4. Generics
- 5. Union & Intersection Types
- 6. Literal Types & Enums
- 7. Type Assertions & Type Guards
- 8. Classes & Access Modifiers
- 9. Advanced Types (Utility Types)
- 10. Decorators
- 11. Modules & Namespaces
- 12. tsconfig.json Essentials

1. Basic Types

TypeScript adds static typing to JavaScript. You tell TypeScript what type of data you're working with.

Primitive Types



typescript

```
let username: string = "John";
let age: number = 25;
let isActive: boolean = true;
let nothing: null = null;
let notDefined: undefined = undefined;
```

Any, Unknown, Never



typescript

```
let anything: any = "bad practice"; // avoid this, disables type checking
let mystery: unknown = "safer"; // must check type before using
function throwError(): never { // never returns
  throw new Error("Boom!");
}
```

Arrays & Tuples



typescript

```
let numbers: number[] = [1, 2, 3];
let mixed: (string | number)[] = ["hi", 42];
let tuple: [string, number] = ["age", 25]; // fixed order & types
```

Theory: any turns off type checking. unknown is safer because you must narrow it down before use. never represents values that never occur (infinite loops, functions that always throw).

2. Interfaces vs Types

Both define object shapes, but with differences.

Interface



typescript

```
interface User {
  id: number;
  name: string;
  email?: string; // optional property
  readonly createdAt: Date; // cannot be modified
}

// Extending interfaces
interface Admin extends User {
  permissions: string[];
}
```

Type Alias



typescript

```
type Point = { x: number; y: number };  
type ID = string | number; // unions  
type Callback = (data: string) => void; // function type
```

When to Use What

- **Interface:** For object shapes, especially when you need to extend or merge
- **Type:** For unions, intersections, primitives, tuples

Theory: Interfaces can be reopened and merged (declaration merging). Types cannot. Use interfaces for public APIs and types for everything else.

3. Functions

TypeScript lets you type parameters and return values.

Basic Function Types



typescript

```
function add(a: number, b: number): number {  
  return a + b;  
}  
  
const multiply = (a: number, b: number): number => a * b;
```

Optional & Default Parameters



typescript

```
function greet(name: string, age?: number): string {  
    return age ? `Hi ${name}, ${age}` : `Hi ${name}`;  
}
```

```
function createUser(name: string, role: string = "user") {  
    return { name, role };  
}
```

Rest Parameters & Function Overloads



typescript

```
function sum(...numbers: number[]): number {  
    return numbers.reduce((a, b) => a + b, 0);  
}
```

// Overloads

```
function format(value: string): string;  
function format(value: number): string;  
function format(value: string | number): string {  
    return String(value);  
}
```

Theory: Function overloads let you define multiple function signatures. The implementation must handle all cases.

4. Generics

Generics let you write reusable, type-safe code.

Basic Generics



typescript

```
function identity<T>(value: T): T {  
    return value;  
}  
  
const num = identity<number>(42);  
const str = identity("hello"); // type inference
```

Generic Interfaces & Classes



typescript

```
interface Box<T> {  
    value: T;  
}  
  
class DataStore<T> {  
    private data: T[] = [];  
  
    add(item: T): void {  
        this.data.push(item);  
    }  
  
    get(index: number): T {  
        return this.data[index];  
    }  
}
```

Generic Constraints



typescript

```
interface HasLength {
  length: number;
}

function logLength<T extends HasLength>(item: T): void {
  console.log(item.length);
}

logLength("hello"); // ok
logLength([1, 2, 3]); // ok
// logLength(123); // error
```

Theory: Generics preserve type information while being flexible. Constraints ensure the generic type has certain properties or methods.

5. Union & Intersection Types

Union Types (OR)



typescript

```
type Status = "pending" | "approved" | "rejected";
let status: Status = "pending";

function process(id: string | number) {
  if (typeof id === "string") {
    console.log(id.toUpperCase());
  } else {
    console.log(id.toFixed(2));
  }
}
```

Intersection Types (AND)



typescript

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

```
type Worker = Person & Employee;
```

```
const worker: Worker = {  
  name: "John",  
  employeeId: 123  
};
```

Theory: Union types mean "one of these types". Intersection types mean "all of these types combined". Use unions for alternatives, intersections for combinations.

6. Literal Types & Enums

Literal Types



typescript

```
let direction: "north" | "south" | "east" | "west";  
direction = "north"; // ok  
// direction = "up"; // error  
  
const config = {  
  api: "https://api.com"  
} as const; // makes properties readonly and literal
```

Enums



typescript

```
enum Color {  
    Red,  
    Green,  
    Blue  
}
```

```
enum Status {  
    Pending = "PENDING",  
    Approved = "APPROVED"  
}
```

```
let color: Color = Color.Red; // 0  
let status: Status = Status.Pending; // "PENDING"
```

Theory: Literal types restrict values to specific constants. Enums create named constants. Use string enums for better debugging, numeric enums for flags.

7. Type Assertions & Type Guards

Type Assertions



typescript

```
let value: unknown = "hello";  
let length: number = (value as string).length;  
// or  
let length2: number = (<string>value).length;
```

Type Guards



typescript


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

```
function process(value: string | number) {  
    if (isString(value)) {  
        console.log(value.toUpperCase());  
    } else {  
        console.log(value.toFixed(2));  
    }  
}
```

```
// instanceof guard  
class Dog {}  
function isDog(animal: any): animal is Dog {  
    return animal instanceof Dog;  
}
```

Discriminated Unions



typescript

```
type Success = { status: "success"; data: string };  
type Error = { status: "error"; message: string };  
type Result = Success | Error;
```

```
function handle(result: Result) {  
    if (result.status === "success") {  
        console.log(result.data);  
    } else {  
        console.log(result.message);  
    }  
}
```

Theory: Type assertions tell TypeScript "trust me, I know the type". Type guards narrow types safely at runtime. Discriminated unions use a common property to distinguish types.

8. Classes & Access Modifiers

Basic Class



typescript

```
class User {  
  public name: string; // accessible everywhere (default)  
  private password: string; // only inside class  
  protected role: string; // inside class and subclasses  
  readonly id: number; // cannot be modified  
  
  constructor(name: string, password: string) {  
    this.name = name;  
    this.password = password;  
    this.role = "user";  
    this.id = Date.now();  
  }  
  
  login() {  
    return this.validatePassword();  
  }  
  
  private validatePassword(): boolean {  
    return this.password.length > 6;  
  }  
}
```

Shorthand Constructor



typescript

```
class Product {  
  constructor(  
    public name: string,  
    private price: number,  
    readonly id: string  
  ) {}  
}
```

Abstract Classes



typescript

```
abstract class Animal {  
  abstract makeSound(): void;  
  
  move(): void {  
    console.log("Moving...");  
  }  
}  
  
class Dog extends Animal {  
  makeSound(): void {  
    console.log("Woof!");  
  }  
}
```

Implementing Interfaces



typescript

```
interface Printable {
    print(): void;
}

class Document implements Printable {
    print(): void {
        console.log("Printing...");
    }
}
```

Theory: public is default. private keeps members internal. protected allows subclass access. readonly prevents modification after initialization. Abstract classes can't be instantiated directly.

9. Advanced Types (Utility Types)

TypeScript provides built-in utility types for common transformations.

Partial & Required



typescript

```
interface User {
    id: number;
    name: string;
    email: string;
}

type PartialUser = Partial<User>; // all optional
type RequiredUser = Required<User>; // all required
```

Pick & Omit



typescript

```
type UserPreview = Pick<User, "id" | "name">; // only id and name
type UserWithoutEmail = Omit<User, "email">; // all except email
```

Record



typescript

```
type Roles = "admin" | "user" | "guest";
type RolePermissions = Record<Roles, string[]>;

const permissions: RolePermissions = {
  admin: ["read", "write", "delete"],
  user: ["read", "write"],
  guest: ["read"]
};
```

ReturnType & Parameters



typescript

```
function createUser(name: string, age: number) {
  return { name, age, createdAt: new Date() };
}

type User = ReturnType<typeof createUser>;
type Params = Parameters<typeof createUser>; // [string, number]
```

Readonly & ReadonlyArray



typescript

```
type ReadonlyUser = Readonly<User>;
const users: ReadonlyArray<User> = [];
// users.push(...); // error
```

Theory: Utility types transform existing types. They save time and reduce boilerplate. `Partial` is great for update functions, `Pick` for creating lightweight versions.

10. Decorators

Decorators add metadata or modify classes, methods, and properties. (Experimental feature)

Class Decorator



typescript

```
function sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}

@sealed
class User {
    name: string;
}
```

Method Decorator



typescript

```
function log(target: any, key: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = function(...args: any[]) {
        console.log(`Calling ${key} with`, args);
        return original.apply(this, args);
    };
}

class Calculator {
    @log
    add(a: number, b: number) {
        return a + b;
    }
}
```

Theory: Decorators run at class definition time. They modify behavior without changing the original code. Common in frameworks like Angular and NestJS. Enable with "experimentalDecorators": true in tsconfig.

11. Modules & Namespaces

ES6 Modules (Preferred)



typescript

```
// math.ts
export function add(a: number, b: number): number {
  return a + b;
}

export const PI = 3.14;

// app.ts
import { add, PI } from './math';
```

Default Exports



typescript

```
// user.ts
export default class User {
  name: string;
}

// app.ts
import User from './user';
```

Namespaces (Legacy)



typescript

```
namespace Utils {  
  export function format(value: string): string {  
    return value.toUpperCase();  
  }  
}
```

```
Utils.format("hello");
```

Theory: Use ES6 modules for modern projects. Namespaces are older and mainly for organizing global scope. Modules are file-based and tree-shakeable.

12. tsconfig.json Essentials

The config file controls TypeScript behavior.



json

```
{  
  "compilerOptions": {  
    "target": "ES2020",  
    "module": "commonjs",  
    "lib": ["ES2020", "DOM"],  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "strict": true,  
    "esModuleInterop": true,  
    "skipLibCheck": true,  
    "forceConsistentCasingInFileNames": true,  
    "resolveJsonModule": true,  
    "declaration": true,  
    "sourceMap": true  
  },  
  "include": ["src/**/*.ts"],  
  "exclude": ["node_modules", "**/*.spec.ts"]  
}
```

Key Options Explained

- **target:** Which JS version to compile to
- **module:** Module system (commonjs, ES6, etc.)

- **strict**: Enables all strict type checking
- **esModuleInterop**: Better compatibility with CommonJS
- **outDir**: Where compiled JS goes
- **rootDir**: Where source TS is
- **declaration**: Generate .d.ts files
- **sourceMap**: Generate .map files for debugging

Theory: `strict: true` is crucial. It enables `strictNullChecks`, `noImplicitAny`, and other safety features. Start strict, don't loosen unless necessary.

Practical Tips

1. Type Inference

Let TypeScript infer when obvious:



typescript

```
const name = "John"; // inferred as string
const count = 0; // inferred as number
```

2. Avoid Any

Replace any with unknown or proper types:



typescript

```
// Bad
function process(data: any) {}

// Good
function process(data: unknown) {
  if (typeof data === "string") {
    // now safe to use as string
  }
}
```

3. Use Const Assertions



typescript

```
const config = {
  api: "https://api.com",
  timeout: 5000
} as const;
// config.api is now literal type, not string
```

4. Discriminated Unions for State



typescript

```
type State =
  | { status: "loading" }
  | { status: "success"; data: string }
  | { status: "error"; error: Error };
```

5. Generic Constraints for Flexibility



typescript

```
function merge<T extends object, U extends object>(a: T, b: U) {
  return { ...a, ...b };
}
```

Common Interview Topics

1. **Difference between interface and type:** Interface can be reopened, better for objects. Type for unions/intersections.
2. **What is never type:** Represents impossible values. Used in exhaustive checks.
3. **Generic constraints:** Restrict what types can be used with generics using extends.
4. **Type guards:** Functions that narrow types at runtime using is keyword.
5. **Utility types:** Partial, Required, Pick, Omit, Record, ReturnType.
6. **Strict mode benefits:** Catches null/undefined errors, requires explicit types, prevents implicit any.
7. **Declaration files (.d.ts):** Provide type definitions for JavaScript libraries.
8. **Variance:** Covariance (return types), contravariance (parameter types), invariance (read/write).

Quick Reference Cheat Sheet



typescript

```
// Types
string, number, boolean, null, undefined, any, unknown, never

// Arrays
number[], Array<number>, [string, number] (tuple)

// Objects
interface User { name: string }
type Point = { x: number; y: number }

// Functions
(a: number, b: number) => number

// Unions & Intersections
string | number, Person & Employee

// Generics
<T>, <T extends Something>





// Utility Types
Partial<T>, Required<T>, Pick<T, K>, Omit<T, K>, Record<K, V>

// Type Guards
typeof, instanceof, value is Type

// Access Modifiers
public, private, protected, readonly
```

Final Checklist for 7/10

- ☒ Understand basic types and when to use each
- ☒ Know interface vs type differences
- ☒ Write typed functions with proper signatures
- ☒ Use generics for reusable code
- ☒ Apply union and intersection types
- ☒ Implement type guards for safety

-  Work with classes and modifiers
-  Use utility types effectively
-  Configure tsconfig.json properly
-  Avoid any, prefer unknown or proper types

You're ready. Go tell people you're a solid 7/10 in TypeScript!