Day 9 — Objects, Functions & Generics (TypeScript)

Covered topics:

- Interfaces vs type aliases
- Function types
- Optional & readonly properties
- Generics (e.g. `List<T>`)
- Structural typing
- Utility types: `Partial`, `Pick`, `Record`

---

1. Interfaces vs Type Aliases

Both **interfaces** and **type aliases** allow you to describe the shape of objects in TypeScript.

1.1 Interface

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

const u: User = {
  id: 1,
  name: "AbdelSalam",
};
```

1.2 Type Alias

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

const u: User = {
  id: 1,
  name: "AbdelSalam",
};
```

In this case they behave the same.

1.3 When to use `interface`

- When you describe the shape of **objects and classes**.
- When you want to allow **extension/merging**.

Example: extend another interface

```
interface Person {
  name: string;
}

interface Employee extends Person {
  id: number;
}

const e: Employee = {
  id: 123,
  name: "Sara",
};
```

Interfaces can also be **merged** across declarations, which is useful for library authors.

1.4 When to use `type`

- When you need **unions or intersections**.
- When the type is not just an object (e.g. primitive, function, tuple).

```
type Id = number | string;

type Point = [number, number]; // tuple

// function type
type Formatter = (value: string) => string;
```

1.5 Simple rule of thumb

- Use `` for most object shapes.
- Use `` when you need unions, intersections, or more advanced type tricks.

---

2. Function Types

Function types describe the parameters and return type of a function.

2.1 Using a type alias

```ts
type Adder = (a: number, b: number) => number;

const add: Adder = (x, y) => x + y;
```

2.2 Using an interface

```ts
interface Adder {
  (a: number, b: number): number;
}

const add: Adder = (x, y) => x + y;
```

Both are valid. `type` is often more common for function types.

2.3 When & why

- Reuse the same function shape in multiple places.
- Give functions **names** so the code is easier to read.
- Let TypeScript check arguments and returned values.

---

3. Optional & readonly Properties

3.1 Optional Properties (`?`)

Optional properties are not required when creating an object.

```ts
interface User {
  id: number;
  name: string;
  email?: string; // optional
}

const u1: User = { id: 1, name: "Bruce" }; // ok without email
const u2: User = { id: 2, name: "Sara", email: "sara@example.com" }; // also ok
```

When to use: when a property may or may not exist (e.g. `middleName`, `avatarUrl`, `description`).

3.2 readonly Properties

`readonly` prevents re-assignment of a property after the object is created.

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

const u: User = { id: 1, name: "Ali" };

u.name = "Omar";  // ✅ allowed
// u.id = 2;      // ❌ Error: Cannot assign to 'id' because it is a read-only property
```

When to use:

- For IDs
- Creation timestamps
- Data that should not change after initialization

---

4. Generics (e.g. `List<T>`)

Generics allow you to define **reusable, type-safe components** that work with multiple types.

Think of `T` as a placeholder type that the caller decides.

4.1 Simple Generic Interface

```
interface List<T> {
  items: T[];
  add(item: T): void;
  get(index: number): T | undefined;
}

const numberList: List<number> = {
  items: [],
  add(item) {
    this.items.push(item);
  },
  get(index) {
    return this.items[index];
  },
};
```

```
numberList.add(10);      // ok
// numberList.add("hi"); // ❌ Error
```

Here:

- `T` is `number` in `numberList`
- TypeScript enforces that only numbers are added.

4.2 Generic Function

```
function wrapInArray<T>(value: T): T[] {
  return [value];
}

const nums = wrapInArray(5);         // T = number, nums: number[]
const names = wrapInArray("Alice"); // T = string, names: string[]
```

When to use generics:

- When you want to write functions/components that work with **many types** but stay type-safe.
- Collections, helpers, reusable utilities.

---

5. Structural Typing

TypeScript uses **structural typing**, also called "duck typing":

> If it looks like the type (has the same structure), it **is** the type.

5.1 Example

```
interface Point {
  x: number;
  y: number;
}

function logPoint(p: Point) {
  console.log(p.x, p.y);
}

const p1 = { x: 10, y: 20 };
const p2 = { x: 5, y: 7, z: 100 };
```

```
logPoint(p1); // ✅ OK
logPoint(p2); // ✅ Also OK (extra property is allowed)
```

Even though `p2` has an extra `z`, it still matches `Point` because `Point` only requires `x` and `y`.

5.2 Why structural typing matters

- You don't have to use `class` or `implements` explicitly.
- As long as the object "fits" the interface, it's accepted.
- Makes TypeScript flexible and easy to integrate with plain JavaScript objects.

---

6. Utility Types: Partial, Pick, Record

TypeScript provides **utility types** to transform existing types.

6.1 `Partial<T>`

`Partial<T>` makes **all properties optional**.

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

// All fields are optional now
type UserUpdate = Partial<User>;

const patch: UserUpdate = {
  name: "New Name", // ok, others can be omitted
};
```

When to use: updates/patches, forms, `updateUser(id, changes)` style functions.

6.2 `Pick<T, K>`

`Pick<T, K>` creates a new type with **only** some properties from `T`.

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

```
  isAdmin: boolean;
}

// Only keep "id" and "name"
type UserSummary = Pick<User, "id" | "name">;

const summary: UserSummary = {
  id: 1,
  name: "Ali",
};
```

When to use:

- Creating lightweight views of objects (e.g. list item summaries)
- Controlling which fields are exposed through an API

6.3 `Record<K, T>`

`Record<K, T>` describes an object whose **keys are of type ** `and whose **values are of type` `\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***`.

```
type Role = "admin" | "user" | "guest";

// Each role maps to a number (e.g. permission level)
const roleLevels: Record<Role, number> = {
  admin: 3,
  user: 2,
  guest: 1,
};
```

Another example:

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

// A lookup table from user id to User
const usersById: Record<number, User> = {
  1: { id: 1, name: "Ali" },
  2: { id: 2, name: "Sara" },
};
```

When to use:

- Maps / dictionaries / lookup tables
- Config objects indexed by known keys

---

7. Hands-on Idea: generic Result + safe map/filter helpers

7.1 Generic Result

A common pattern is a `Result<T>` type to represent either **success** or **error**.

```
interface OkResult<T> {
  ok: true;
  value: T;
}

interface ErrResult {
  ok: false;
  error: string;
}

export type Result<T> = OkResult<T> | ErrResult;

export function ok<T>(value: T): Result<T> {
  return { ok: true, value };
}

export function err<T = never>(message: string): Result<T> {
  return { ok: false, error: message };
}

// Example usage:
const r1: Result<number> = ok(42);
const r2: Result<number> = err("Something went wrong");
```

This uses:

- Generics (`Result<T>`, `ok<T>`)
- Union types
- Narrowing via `ok` boolean flag.

7.2 Safe map helper

```ts
export function safeMap<T, U>(
  items: T[] | null | undefined,
  fn: (item: T) => U
): U[] {
  if (!items || items.length === 0) return [];
  return items.map(fn);
}

// Example:
const names = safeMap([
  { id: 1, name: "Ali" },
  { id: 2, name: "Sara" },
], (u) => u.name);
// names: string[]
```

7.3 Safe filter helper

```ts
export function safeFilter<T>(
  items: T[] | null | undefined,
  predicate: (item: T) => boolean
): T[] {
  if (!items || items.length === 0) return [];
  return items.filter(predicate);
}

// Example:
const admins = safeFilter([
  { id: 1, isAdmin: true },
  { id: 2, isAdmin: false },
], (u) => u.isAdmin);
// admins: { id: number; isAdmin: boolean }[]
```

These helpers show how generics and function types make utilities reusable and safe.

---

8. Summary

- **Interfaces vs type aliases**: both define shapes; interfaces are great for objects/classes and extension, types are more general and support unions, intersections, tuples, etc.
- **Function types**: describe parameter and return types for reusable and type-safe functions.
- **Optional / readonly**: model properties that may be missing or must never change.
- **Generics**: let you write reusable components that work with many types while staying type-safe.
- **Structural typing**: if an object has the right shape, it is accepted, regardless of where it came from.

- **Utility types (Partial, Pick, Record)**: powerful helpers to transform existing types instead of rewriting them.