Day 8 — TypeScript Fundamentals & Types

Covered topics:

- Type inference
- Strict mode essentials (strict, noImplicitAny)
- Union & intersection types
- Literal types
- Type narrowing (typeof / in / instanceof)
- Enums vs union types

---

1. Type Inference

Type inference means TypeScript can figure out the type automatically from the value.

You do **not** always need to write `: type` explicitly — TS often infers it.

1.1 Basic Inference

```
let count = 5;
// TypeScript infers: count is a number

count = 10;      // ✅ OK
// count = "hello"; // ❌ Error: Type 'string' is not assignable to type
'number'
```

1.2 Function Return Type Inference

```
function double(x: number) {
  return x * 2;
}

// TypeScript infers: double(x: number) => number
```

When to rely on inference

- For simple variables: `let x = 1;`
- For obvious returns: `return x * 2;`

When to explicitly write the type

- Public APIs (exported functions)

- Complex objects / arrays / generics
- When clarity matters more than brevity

---

2. Strict Mode Essentials

`strict` mode enables a set of strict type-checking rules that catch more bugs early.

In `tsconfig.json`:

```
{
  "compilerOptions": {
    "strict": true
  }
}
```

This automatically turns on options like:

- `noImplicitAny`
- `strictNullChecks`
- and more.

2.1 noImplicitAny

`noImplicitAny` means TypeScript will **not** allow a parameter or variable to have type `any` implicitly.

Example (Error):

```
function log(message) {
  console.log(message);
}
// ❌ Error in strict mode: 'message' has an implicit 'any' type
```

Fixed:

```
function log(message: string) {
  console.log(message);
}
```

Why strict mode is important

- Catches missing types early
- Reduces runtime bugs

- Forces you to be explicit
- Great for large or long-lived codebases

---

3. Union Types

A **union type** allows a value to be **one of several types**.

Use the `|` operator:

```
let id: string | number;

id = 42;        // ✅
id = "user-1"; // ✅
// id = true;  // ❌ Error
```

3.1 Union with custom types

```
type ApiResult = { data: string } | { error: string };

function handleResult(result: ApiResult) {
  // We'll use narrowing later to distinguish the cases
}
```

When to use unions

- When a value can reasonably be **multiple types**
- e.g. `string | null`
- `number | string`
- success vs error shapes: `{ data } | { error }`

---

4. Intersection Types

An **intersection type** combines multiple types into one using `&`.

> The value must satisfy **all** the types in the intersection.

```
type WithId = { id: number };
type WithName = { name: string };

type User = WithId & WithName;
```

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

When to use intersections

- To build complex types from smaller pieces
- To compose features: e.g. `Draggable & Resizable`

---

5. Literal Types

A **literal type** represents a specific value, not just a general type.

```ts
let direction: "left" | "right";

direction = "left";   // ✅
direction = "right";  // ✅
// direction = "up";  // ❌ Error
```

Here:

- `"left"` and `"right"` are **literal types**
- `direction` can only be one of those exact strings

5.1 Literal union for statuses

```ts
type Status = "idle" | "loading" | "success" | "error";

let status: Status = "idle";
status = "loading";   // ✅
// status = "finished"; // ❌ not allowed
```

When to use literal types

- For fixed sets of allowed values
- Statuses: `"idle" | "loading" | ...`
- Themes: `"light" | "dark"`
- Directions: `"up" | "down" | ...`
- Great for preventing typos and invalid states

---

6. Type Narrowing

Narrowing means: using runtime checks (like `typeof`, `in`, or `instanceof`) so TypeScript can refine a union type inside a block.

6.1 Narrowing with typeof

```
function printValue(v: string | number) {
  if (typeof v === "string") {
    // Here v is a string
    console.log(v.toUpperCase());
  } else {
    // Here v is a number
    console.log(v.toFixed(2));
  }
}
```

6.2 Narrowing with `in`

```
type User = { name: string; age: number };
type ApiError = { message: string; code: number };

type Result = User | ApiError;

function handle(result: Result) {
  if ("name" in result) {
    // result is a User
    console.log("User:", result.name);
  } else {
    // result is an ApiError
    console.log("Error:", result.message);
  }
}
```

6.3 Narrowing with instanceof

```
function logError(err: unknown) {
  if (err instanceof Error) {
    console.log("Error message:", err.message);
  } else {
    console.log("Unknown error:", err);
  }
}
```

When narrowing is useful

- Whenever you have a union type
- And you need to safely access properties/methods that only exist on **some** of the union members

---

7. Enums vs Union Types

7.1 Enums

An **enum** is a TypeScript feature that defines a named set of values.

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}

let d: Direction = Direction.Up;
```

This produces JavaScript at runtime:

```
{
  0: "Up",
  1: "Down",
  2: "Left",
  3: "Right",
  Up: 0,
  Down: 1,
  Left: 2,
  Right: 3
}
```

You can also use string enums:

```
enum Status {
  Idle = "idle",
  Loading = "loading",
  Success = "success",
  Error = "error",
}
```

```
let s: Status = Status.Loading;
```

7.2 Union of literal types (modern alternative)

```
type DirectionLiteral = "up" | "down" | "left" | "right";

let d2: DirectionLiteral = "up";
```

This does **not** generate extra JavaScript; it exists only in TypeScript's type system.

7.3 When to use enums

- When you need a value at **runtime** (e.g. pass around `Status.Idle`)
- When integrating with existing codebases that already use enums

7.4 When to prefer literal unions

- Most modern codebases prefer unions of literals:
- Lighter (no runtime JS output)
- Simpler
- Great for modeling finite states

Example:

```
type StatusLiteral = "idle" | "loading" | "success" | "error";

let s2: StatusLiteral = "loading";
```

---

8. Putting It All Together — Example

```
// 1) Define a literal union type for status
export type Status = "idle" | "loading" | "success" | "error";

// 2) Define a User type
export type User = {
  id: number;
  name: string;
  role?: "admin" | "user"; // optional literal union
};

// 3) A function that uses unions and narrowing
export function formatResult(input: User | { error: string }): string {
```

```
  if ("error" in input) {
    // Narrowed to error shape
    return `Error: ${input.error}`;
  }

  // Here input is a User
  const role = input.role ?? "user";
  return `User #${input.id}: ${input.name} (${role})`;
}

// 4) Inferred types
let currentStatus = "idle" as Status;

function setStatus(newStatus: Status) {
  currentStatus = newStatus;
}

setStatus("loading");
// setStatus("done"); // ❌ not allowed
```

---

9. Comparison Summary

9.1 Union vs Intersection

- **Union (``****)**: value can be **A or B**
- Example: `string | number`
- **Intersection (``****)**: value must satisfy **A and B**
- Example: `{ id: number } & { name: string }`

9.2 Literal Types vs Normal Types

- **Normal type**: `string`, `number`, etc. (any value of that kind)
- **Literal type**: specific values like `"up" | "down"`

Literal types are great for fixed sets of allowed options.

9.3 Enums vs Literal Unions

Feature	Enums	Literal Unions
Syntax	`enum Status { Idle, ... }`	`type Status = "idle" | "loading" | ...`

Feature	Enums	Literal Unions
Runtime output	Yes (extra JS object)	No (type system only)
Performance	Slight overhead	Very lightweight
Use cases	When you need runtime enum	Most modern TS codebases
Recommended	Older / special use cases	**Preferred in most new projects**

9.4 When to use what (quick guide)

- **Type inference**: Use it for simple cases, but be explicit for public APIs.
- **Strict mode**: Always enable it for serious projects.
- **Unions**: When a value can be multiple types.
- **Intersections**: When a value must combine multiple shapes.
- **Literal types**: For fixed sets of string/number options.
- **Narrowing**: Whenever you have unions and need safe property access.
- **Enums**: Only if you really need runtime enum objects; otherwise use literal unions.