
STACK
LEARNER

HM Nayem

Your Instructor 😎

Today's Agenda

- `type` vs `Interfaces`
- Handling `any` and `unknown` safely
- Understand Type Guard
- Understand `never` type
- Const & Satisfy
- Convert JS Codes into TypeScript
- Utility types: `Partial`, `Pick`, `Omit`, `Record`, `Readonly`.

Let’s Begin . . .

type **VS** interface

Type vs Interface

Both type and interface are used to define the shape of objects in TypeScript. While they overlap in functionality, they have key differences that impact how and when to use them.

Comparison of `type` & `interface`

Feature	`type`	`interface`
Extensibility	Closed after definition. Use `&` to combine.	Open for augmentation (can be extended).
Primitive Types	Can define primitives (`string`, `number`, etc.).	Cannot define primitives.
Union and Intersection Types	Supports unions and intersections.	Does not support unions directly.
Callable/Constructable Types	Can define function and constructor types.	Can define callable and constructor types.
Duplicate Definitions	Duplicate definitions are not allowed.	Can have duplicate declarations (merged).
Compile-Time Performance	Faster for complex types.	Slightly slower for complex cases.

When to Use What?

Use `interface`

- For defining objects, especially when you need extensibility (e.g., libraries or large projects).
- When augmenting types in existing code or third-party libraries.

Use `type`

- For primitives, unions, intersections, and complex function signatures.
- When dealing with utility types and advanced TypeScript features.

Handling any **VS** unknown

Any

What is `any`?

- For defining objects, especially when you need extensibility *(e.g. libraries or large projects)*.
- When augmenting types in existing code or third-party libraries.

Why Avoid `any`?

- **No Type Safety:** Using `any` bypasses TypeScript's type system.
- **Hard to Debug:** Errors show up at runtime rather than compile-time.

Unknown

What is `unknown`?

The `unknown` type is safer than `any` because it ensures the developer explicitly performs type checks before using the value.

Best Practices of `unknown`

- Avoid `any` unless absolutely necessary *(e.g. rapid prototyping)*.
- **Use `unknown` when:**
  - You're dealing with APIs or third-party data with uncertain types.
  - You want to enforce type checks for safety.
- **Always narrow down `unknown` before using it, either with:**
  - Type guards.
  - Type assertions *(use sparingly)*.
- Write custom type guards for complex objects to ensure type safety.

Comparison of `any` & `unknown`

Feature	`any`	`unknown`
Type Checking	No type checking; unsafe.	Requires explicit checks before usage.
Flexibility	Extremely flexible (no restrictions).	Flexible but controlled by type checks.
Safety	Unsafe: runtime errors are common.	Safe: enforces type checks.
Common Use Case	Temporary debugging, legacy code.	Safe handling of unknown inputs.
Example Usage	`let data: any;`	`let data: unknown;`

Understand Type Guard

Type Guard

What is Type Guard?

A type guard is a runtime check that ensures a value matches a specific type. Type guards are particularly useful when working with union types or unknown types, as they allow you to safely narrow down the type of a variable during runtime.

Common Type Guard Syntax

- `typeof` **checks:** For primitive types like `string`, `number`, etc.
- `instanceof` **checks:** For class-based objects.
- **Custom Type Guards:** Using a function that returns a boolean and asserts the type.

Advantages of Type Guard

- **Type Safety:** Ensures your code handles types correctly.
- **Error Prevention:** Reduces runtime errors by validating types before use.
- **Improved Intellisense:** Helps TypeScript narrow down the type automatically after the guard.
- **Maintainable Code:** Keeps validation logic modular and reusable.

Best Practices for Type Guards

- **Use Custom Type Guards for Complex Types:**

  Validate objects with multiple fields to ensure their structure matches the expected type.

- **Use Built-in Guards for Simple Checks:**

  Use `typeof` for primitive types and `instanceof` for class-based objects.

- **Always Validate External Data:**

  For API responses or JSON parsing, validate the data structure before using it.

never **Type**

Never

What is `never`?

The `never` type in TypeScript represents a value that **never occurs**. It is the return type for functions or expressions that:

1. Never return a value
2. Always throw an error
3. Have infinite loops

It is primarily used for exhaustive type checking and to catch programming errors at compile time.

Key Characteristics of `never`

- It is a **subtype** of every type, but no type is a subtype of `never` except `never` itself.
- It cannot have any value.
- It is often used for error handling and ensuring type safety in exhaustive checks.

Real Use Cases of
const & satisfies

Const

What is `const` Assertion?

The `const` assertion tells TypeScript to narrow down the type of an object, array, or literal to the most specific possible type and prevents it from being widened.

Key Benefits

- ➔ Prevents mutation of values.
- ➔ Preserves literal types in objects or arrays, ensuring type precision.

Satisfies

What is `const` Assertion?

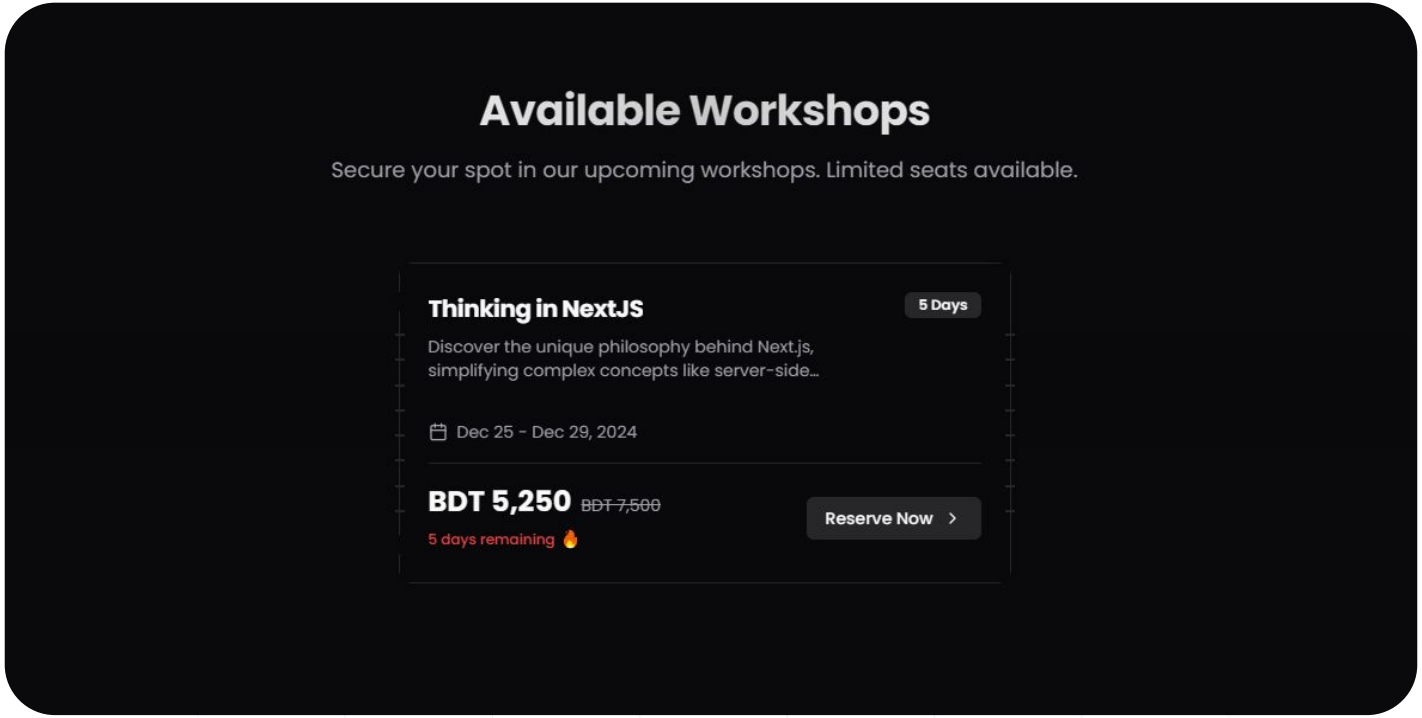
The `satisfies` operator, introduced in TypeScript 4.9, allows developers to validate that an object matches a specific type without fully defining it as that type. It works as a middle ground between type checking and preserving excess properties.

Key Benefits

- ➔ Ensures a value conforms to a type.
- ➔ Allows excess properties without triggering type errors.
- ➔ Retains type inference for specific values.

Utility Types

Our Upcoming Workshop



Thank You!