The TypeScript Handbook

Predrag Milanović

TypeScript for Developers

This handbook gives you a concise, practical TypeScript reference without the fluff.

I assume you're already familiar with programming fundamentals in at least one other language.
I'll move quickly through the core TypeScript concepts and patterns.

You'll get: - Minimal, example-focused explanations

Not included: - Framework specifics (React/Vue/etc.) - Deep dives into build tooling

---

Want the complete TypeScript guide?

For comprehensive documentation and in-depth coverage, visit the official TypeScript documentation.

Basic Types

TypeScript has static types. This TypeScript code:

```
const bootupMessage: string = "Starting support.ai servers...";
```

is equivalent to this JavaScript code:

```
const bootupMessage = "Starting support.ai servers...";
```

The `: string` annotation specifies the variable's type. The same pattern works for other primitive types like `number`, `boolean`, `null`, and `undefined`:

```
const bootupMessage: string = "Starting support.ai servers...";
const port: number = 3000;
const isServerOnline: boolean = true;
const noValue: null = null;
const notDefined: undefined = undefined;
```

If a value doesn't match its annotated type, TypeScript will report an error at compile time. For example, this is invalid:

```
const bootupMessage: string = 123;
// Error: Type 'number' is not assignable to type 'string'.
```

Notes: - TypeScript will often infer types if you omit annotations (e.g., `const x = 1` infers `number`). - Use annotations when you want an explicit contract or when the inferred type is too broad.

Type Inference

If you find typing long annotations tedious, good news — TypeScript is excellent at inferring types for you.

Instead of explicitly declaring the type:

```
const bootupLog: string = "Starting support.ai servers...";
```

you can rely on inference and write the shorter equivalent:

```
const bootupLog = "Starting support.ai servers...";
```

TypeScript is incredible at type inference. TypeScript infers the type of `bootupLog` as `string`, so the shorter form is just as safe (and less error-prone) than an explicit annotation. Prefer inference for local constants and variables unless you need a specific, broader, or intentionally different type.

Any

When you compile plain JavaScript code using `tsc`, your codebase is full of any types.

The any type is exactly what it sounds like - a type that can be anything. The purpose of types, really, is to narrow down the possible values that a variable can hold. From that perspective, `any` is the most useless type because it doesn't narrow anything down at all! But it's important because it allows you to opt out of type-checking for a variable.

The any type is super useful when you migrate an existing JavaScript codebase to TypeScript. The (very simplified) process is:

- Change file extensions from `.js` to `.ts`,
- Get `tsc` running without errors (often works out of the box, due to `any`),
- Slowly over time, replace `any`'s with more specific types.

Types in TypeScript

This section covers the fundamentals of TypeScript's type system, from basic type annotations to type inference and the escape hatch of `any`.

What is TypeScript?

TypeScript is a superset of JavaScript developed by Microsoft, meaning all JavaScript code is valid TypeScript. TypeScript adds **static typing** to JavaScript, enabling compile-time error checking for safer, more maintainable code.

The TypeScript compiler (`tsc`) validates your code and compiles it to JavaScript before execution. This catch bugs early and makes refactoring easier.

Why TypeScript?

- **Catches bugs at compile time** before they reach production
- **Improves code readability** with explicit type contracts
- **Enables safer refactoring** with type-aware tooling
- **Better developer experience** with IDE autocomplete and documentation

Topics

01. Basic Types

Learn TypeScript's fundamental type system. Covers: - Type annotations: `: string`, `: number`, `: boolean`, etc. - Primitive types: `string`, `number`, `boolean`, `null`, `undefined` - Type mismatch errors - When to use explicit annotations

Key takeaway: Annotate types to create explicit contracts; TypeScript enforces them at compile time.

02. Type Inference

Understand how TypeScript automatically infers types. Covers: - Implicit type inference from variable values - When inference is sufficient (local variables) - When explicit annotations help (APIs, complex logic) - Best practices for balancing inference and clarity

Key takeaway: Prefer inference for internal code; use annotations for public APIs and when you need an explicit contract.

03. Any

Learn about `any`, TypeScript's type escape hatch. Covers: - What `any` means and why it exists - Migration strategy: `.js` → `.ts` with `any` - Gradually replacing `any` with specific types - Why `any` is useful but should be minimized

Key takeaway: Use `any` during migrations to get TypeScript running; replace it with specific types over time.

Quick Tips

1. **Use annotations for clarity**, not just to satisfy the compiler.
2. **Trust inference** for local variables and straightforward assignments.
3. **Annotate function parameters and return types** to document your API contract.
4. **Avoid `any`** in new code; use it strategically during migrations.
5. **Type errors are your friend** — they catch bugs before runtime.

Type System Overview

Concept	Example	Use Case
Annotation	`const x: number = 5`	Explicit type contract
Inference	`const x = 5`	Local variables, clear intent
Any	`const x: any = ...`	Migrations, unknown types
Union	`string | number`	Multiple possible types
Literal	`"north" | "south"`	Restricted set of values

Function Type Syntax

One of the most useful places for explicit types is a function signature. Each parameter can have a type annotation, and the function itself can declare a return type. Example with a regular function:

```
function createMessage(name: string, a: number, b: number): string {
  return `${name} scored ${a + b}`;
}
```

Here: - `name: string`, `a: number`, and `b: number` annotate each parameter's type. - The `: string` after the parameter list declares the return type.

Arrow functions use the same syntax for parameters and return type:

```
const createMessage = (name: string, a: number, b: number): string => {
  return `${name} scored ${a + b}`;
};
```

Notes and best practices: - Prefer letting TypeScript infer the return type for short functions when the implementation is obvious; explicit return types are valuable for public APIs and when you want to enforce a contract. - Use parameter annotations to make intent clear and to surface mistakes early during development.

- For more complex signatures consider using type aliases or interfaces for readability, e.g. `type Score = (name: string, a: number, b: number) => string;`.

Inferred Return Types

Much like preferring `const x = 1` over `const x: number = 1`, it's usually fine — and often preferable — to let TypeScript infer a function's return type instead of annotating it explicitly.

Simple example:

```
function divide(a: number, b: number) {
    return a / b; // inferred as number
}

const result = divide(10, 2); // result: number
```

Arrow functions work the same way:

```
const toUpper = (s: string) => s.toUpperCase(); // inferred return: string
```

Async functions and promises:

```
async function fetchCount(): Promise<number> {
    return 42;
}

// If you omit the return annotation, TypeScript still infers Promise<number>
async function fetchCountInferred() {
    return 42; // inferred as Promise<number>
}
```

When inference can be problematic - Complex conditional returns — if different branches produce values that require manual narrowing, an explicit return type documents intent and catches accidental regressions.

```
function parseAmount(src: string) {
    if (src === '') return null; // inferred: string | null
    return Number(src);
}

// If you wanted to enforce always returning number | null, you might annotate it:
function parseAmountExplicit(src: string): number | null {
    if (src === '') return null;
    return Number(src);
}
```

- Any `any`-producing operations (e.g., `JSON.parse`) — you may want explicit annotations to avoid leaking `any` into your API.

Generic helpers and callbacks — return types are often inferred and remain useful without explicit annotations:

```
function identity<T>(v: T) {
    return v; // inferred return: T
}

const mapped = [1, 2, 3].map(n => n * 2); // inferred number[]
```

When to prefer explicit return types - Public APIs, library code, or exported functions where the declared contract matters. - When inference would produce an overly broad type (e.g., `any`) or hide complexity. -

When using function overloads — the overload signatures are the public contract and the implementation may have a broader parameter/return shape.

In practice, prefer inference for small, internal helpers and use explicit return types for exported or critical functions. This keeps code concise while preserving clear API boundaries.

Void

The TypeScript `void` type represents functions that intentionally return nothing.

```
function logMessage(message: string): void {
    console.log(message);
    // explicitly returns nothing
}
```

In JavaScript a function with no `return` statement yields `undefined` at runtime, which is a bit vague. In TypeScript, `void` communicates intent: the function is not meant to produce a meaningful value.

When to use `void`: - For event handlers, logging, or other side-effect-only functions. - On function types when you want to signal "no useful return value" to callers, e.g. `(s: string) => void`.

Examples:

```
// event handler type
type ClickHandler = (event: MouseEvent) => void;

// callback that doesn't return a value
function withLogging(cb: (msg: string) => void) {
    console.log('before');
    cb('hi');
    console.log('after');
}

// async functions that don't return meaningful data: inferred Promise<void>
async function save(): Promise<void> {
    await db.write();
}
```

Notes: - Prefer `void` when you really intend "no meaningful return". Don't use it to hide useful return values. - `void` is different from `undefined` and from `any`; it's a signal about intent, not a full runtime guarantee.

```
# Function Types

Functions are values in JavaScript - and TypeScript - so they have types too. But function types aren't

## Syntax

A function type is written as:
```

(param1: type1, param2: type2, …) => returnType

```
For example, a function that takes two numbers and returns a number:

```ts
(a: number, b: number) => number
```
```

Both of these functions match that type:

```
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
```

Using Function Types

Name function types with a `type` alias for reuse:

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

const multiply: MathOp = (x, y) => x * y;
const divide: MathOp = (x, y) => x / y;
```

Assign function types to variables or parameters:

```
// variable holding a function of type (name: string) => string
const greet: (name: string) => string = (n) => `Hello, ${n}!`;

// callback parameter
function execute(cb: (x: number) => void) {
  cb(42);
}
```

Function types in higher-order functions (functions that take or return functions):

```
type Mapper<T, U> = (item: T) => U;

function map<T, U>(arr: T[], transform: Mapper<T, U>): U[] {
  return arr.map(transform);
}

const nums = [1, 2, 3];
const strs = map(nums, (n) => n.toString()); // inferred: string[]
```

Optional and Default Parameters in Function Types

```
// optional param: ? suffix
type Logger = (msg: string, level?: string) => void;

// default param
type Multiply = (a: number, b?: number) => number;
```

Key Takeaway

Function types capture the shape of a function: its parameter types and return type. Use them to annotate callbacks, higher-order functions, and to enforce contracts across your codebase.

Type Alias

Long, complex types can clutter your code. Type aliases let you name and reuse them, improving readability and maintainability.

The Problem

Without type aliases, function signatures can become verbose and hard to read:

```
function setLoggerTimeout(
  loggerCallback: (s1: string, s2: string) => string,
  delay: number,
) {
  // do something
}
```

The Solution

Use the `type` keyword to create a type alias:

```
type LoggerCallback = (s1: string, s2: string) => string;
```

Now you can use `LoggerCallback` anywhere you need this specific function type:

```
function setLoggerTimeout(loggerCallback: LoggerCallback, delay: number) {
  // do something
}
```

Much cleaner and easier to read!

Benefits

- **Readability**: Named types communicate intent better than inline annotations.
- **Reusability**: Define once, use everywhere.
- **Maintainability**: Change the type definition in one place, and all usages update automatically — no risk of copy-paste errors.

More Examples

Object types:

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

function greetUser(user: User): string {
  return `Hello, ${user.name}!`;
}
```

Union types:

```
type Status = 'idle' | 'loading' | 'success' | 'error';

function setStatus(status: Status) {
  // ...
}
```

Generic type aliases:

```
type Result<T> = { ok: true; value: T } | { ok: false; error: string };

function fetchData(): Result<number> {
  // ...
}
```

When to Use Type Aliases

- When a type is used in multiple places.
- When a type is complex or verbose.
- When you want to give a meaningful name to a type for clarity.

Prefer type aliases over repeating the same inline type annotations. Your future self (and your teammates) will thank you.

Importing Types

When you need to use types from another module, TypeScript provides a dedicated syntax for importing types only — without importing runtime values.

Basic Syntax

The `import type` syntax signals to TypeScript that you're importing only types:

```
import type { User, Post } from "./models";
```

Compare with a regular import (which might include runtime values):

```
import { User, Post } from "./models"; // may include runtime code
```

Why Use `import type`?

TypeScript will strip out type-only imports during compilation, so they don't generate any JavaScript code. This keeps your bundle smaller and makes the intent clear: these are compile-time types, not runtime values.

Alternative Syntax

You can also mix types and values in a single import using the `type` keyword on individual items:

```
import { type User, type Post } from "./models";
import { fetchUser } from "./models"; // runtime value
```

Or combined:

```
import { type User, fetchUser } from "./models";
```

Recommendation

Prefer `import type { ... }` for all type-only imports. It's: - **Concise**: all types in one dedicated import statement. - **Clear**: explicitly signals "types only" to readers. - **Organized**: separates type imports from value imports.

Example

```
// models.ts
export type User = {
  id: number;
  name: string;
};

export function fetchUser(id: number): Promise<User> {
  // ...
}
```

```ts
// app.ts
import type { User } from "./models";
import { fetchUser } from "./models";

async function loadUser(id: number): Promise<User> {
  return fetchUser(id);
}
```

Best Practice

Always use `import type` for types. It makes your code clearer, keeps bundles lean, and helps avoid circular dependency issues.

Functions in TypeScript

This section covers how to annotate and work with function types in TypeScript, from basic parameter and return type syntax to advanced patterns.

Overview

Functions are first-class values in TypeScript, which means they have types too. Understanding function types helps you write safer, more maintainable code with clear contracts between callers and implementations.

Topics

01. Function Type Syntax

Learn how to annotate function parameters and return types. Covers: - Basic parameter and return type annotations - Regular functions vs arrow functions - Optional, default, and rest parameters - Function overloads - Generic functions

Key takeaway: Annotate parameters to clarify intent; consider inferring return types for short functions.

02. Inferred Return Types

Understand when and how TypeScript infers function return types. Covers: - Simple inference (e.g., `const divide = (a, b) => a / b`) - Async functions and `Promise<T>` inference - When to prefer explicit return types (public APIs, complex logic) - Generic helpers

Key takeaway: Prefer inference for internal helpers; use explicit return types for exported/critical functions.

03. Void

Explore the `void` type for functions that don't return meaningful values. Covers: - Using `void` for side-effect-only functions - Event handlers and callbacks - Async functions with `Promise<void>` - The difference between `void`, `undefined`, and `any`

Key takeaway: Use `void` to signal intent that a function is not meant to produce a meaningful value.

04. Function Types

Master function types as values. Covers: - Function type syntax: `(param: type) => returnType` - Type aliases for function types - Assigning function types to variables - Higher-order functions with generics - Optional and default parameters in function types

Key takeaway: Use type aliases to name complex function signatures for clarity and reusability.

05. Type Alias

Learn how to name and reuse complex types using `type`. Covers: - Simplifying verbose function signatures - Object types, union types, and generic aliases - Benefits: readability, reusability, maintainability - When to use type aliases

Key takeaway: Always use type aliases for types that appear multiple times or are complex.

06. Importing Types

Understand how to import types cleanly and efficiently. Covers: - `import type { ... }` syntax - Why it's better (smaller bundles, clearer intent) - Alternative inline syntax (`import { type User, ... }`) - Best practices

Key takeaway: Always use `import type` for types to keep bundles lean and intent clear.

Unions

Union types let you specify that a value can be one of several types. Use the pipe symbol (`|`) to separate types:

```
// userId is a string OR a number
let userId: string | number;
userId = "user_42";  // valid
userId = 42;         // valid
userId = true;       // error
```

Type Narrowing

One powerful feature of TypeScript is **type narrowing**: when you check a value's type, TypeScript automatically refines the type in that branch. This makes it safe to access type-specific properties or call type-specific methods.

Example with `typeof`:

```
function safeSquare(val: string | number): number {
  if (typeof val === "string") {
    // Inside this block, val is narrowed to string
    val = parseInt(val, 10);
  }
  // Now val is only a number
  return val * val;
}

let result = safeSquare("5");
console.log(result);  // 25

result = safeSquare(5);
console.log(result);  // 25
```

More Narrowing Examples

Using `instanceof` for objects:

```
class Dog {
  bark() { console.log("Woof!"); }
}
```

```
class Cat {
  meow() { console.log("Meow!"); }
}

function makeSound(pet: Dog | Cat) {
  if (pet instanceof Dog) {
    pet.bark();  // safe: pet is narrowed to Dog
  } else {
    pet.meow();  // safe: pet is narrowed to Cat
  }
}
```

Checking for property existence:

```
type Bird = { fly: () => void };
type Fish = { swim: () => void };

function move(animal: Bird | Fish) {
  if ("fly" in animal) {
    animal.fly();
  } else {
    animal.swim();
  }
}
```

Use Cases

- **API responses**: A field might be `User | Error`
- **Configuration**: A setting might accept `string | number | boolean`
- **Flexible parameters**: Functions that accept multiple input types
- **Optional values**: `T | null` or `T | undefined`

Key Takeaway

Unions let you express that a value can be multiple types. Use type narrowing (conditionals) to safely access type-specific operations without casts or assertions.

Optional Parameters

Function parameters can be marked optional with a question mark (`?`) after the parameter name. When a caller omits an optional parameter, it receives the value `undefined`.

Basic Example

```
function greet(name: string, title?: string): string {
  if (title) {
    return `Hello, ${title} ${name}!`;
  }
  return `Hello, ${name}!`;
}

greet("Gandalf");           // "Hello, Gandalf!"
greet("Gandalf", "Wizard"); // "Hello, Wizard Gandalf!"
```

Key Rules

1. Optional parameters must come after required parameters.

This won't compile:

```
// Error: Required parameter cannot follow optional parameter
function greet(title?: string, name: string): string {
  // ...
}
```

2. Optional parameters have `undefined` automatically unioned with their type.

Inside the function, an optional parameter has type `Type | undefined`:

```
function greet(name: string, title?: string): string {
  // Inside the function, title has type: string | undefined

  if (title) {
    // Here, title is narrowed to string
    return `Hello, ${title} ${name}!`;
  }
  // Here, title is narrowed to undefined
  return `Hello, ${name}!`;
}
```

Optional vs Default Parameters

Optional parameters default to `undefined`, but you can provide an explicit default value:

```
// Optional: defaults to undefined
function logMessage(msg: string, level?: string) {
  console.log(`[${level || "INFO"}] ${msg}`);
}

// Default value: defaults to "INFO"
function logMessageWithDefault(msg: string, level: string = "INFO") {
  console.log(`[${level}] ${msg}`);
}

logMessage("Error occurred");           // "[INFO] Error occurred"
logMessageWithDefault("Error occurred"); // "[INFO] Error occurred"
```

Type Narrowing with Optional Parameters

Always check for `undefined` before using an optional parameter:

```
function formatDate(date: Date, format?: string): string {
  if (format === undefined) {
    return date.toDateString();
  }
  // Safe to use format here - it's narrowed to string
  return date.toLocaleString("en-US", { dateStyle: format });
}
```

Key Takeaway

Use `?` for optional parameters; they automatically union with `undefined`. Always place optional parameters after required ones, and use type narrowing to safely access them.

Default Parameters

Default parameters provide fallback values for function arguments when the caller doesn't supply them.

Basic Example

```
function newCharacter(name: string, role: string = "warrior"): string {
  return `${name} is a ${role}`;
}

console.log(newCharacter("Gandalf"));
// Gandalf is a warrior
console.log(newCharacter("Gandalf", "wizard"));
// Gandalf is a wizard
```

Type Inference

When you use a default value, you don't need to mark the parameter as optional with `?`, and TypeScript can infer the parameter's type automatically:

```
function countdown(start = 10): void {
  // start is inferred as number (not number | undefined)
  console.log(`Counting down from ${start}...`);
}

countdown();    // uses default 10
countdown(5);   // uses provided 5
```

Notice that `start` has type `number`, not `number | undefined` — because a default value is always provided (either by the caller or by the function).

Default vs Optional

Feature	Optional (`?`)	Default Value
Parameter type inside function	`Type | undefined`	`Type`
Requires type annotation	No	No (inferred)
Must come after required params	Yes	Yes
Always has a value	No (can be `undefined`)	Yes

```
// Optional: can be undefined
function greet(name: string, title?: string) {
  // title: string | undefined
}

// Default: never undefined inside function
function greetWithDefault(name: string, title: string = "Friend") {
  // title: string
}
```

Type Widening

Sometimes you want a default value of one type but accept a wider type as parameter. Use an explicit type annotation:

```
// Without annotation: inferred as number (literal 0)
function add(a: number, b = 0) {
  return a + b;  // b: number
}

// With annotation: explicitly widen to accept string or number
function addWide(a: number | string, b: number | string = 0) {
  // b can be number or string
  return a + b;
}
```

Key Takeaway

Use default parameters to provide sensible fallbacks. TypeScript infers the parameter type from the default value, so you usually don't need explicit annotations — unless you want to widen the type beyond what the default suggests.

Literal Types

Literal types allow you to specify that a variable can only be a specific literal value. This is a powerful feature for constraining what values are accepted, making your code more type-safe.

The Problem

Consider a `move` function that accepts a direction:

```
function move(direction: string) {
  // Implementation...
}
```

This accepts *any* string — "north", "south", "invalid", "anything" — which is too permissive. We need to narrow the possible values.

The Solution: Literal Types

Use a literal value as a type to restrict to that exact value:

```
function move(direction: "north") {
  // Implementation...
}

move("north");  // valid
move("south");  // error
```

Now `direction` can only be the string `"north"`.

Combining Literals with Unions

Literal types become powerful when combined with unions to restrict a variable to a specific set of values:

```
function move(direction: "north" | "south" | "east" | "west"): void {
  console.log(`Moving ${direction}`);
```

```
}

move("north");   // valid
move("south");   // valid
move("left");    // error
```

Literal Type Examples

String literals:

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

function setStatus(status: Status): void {
  // ...
}

setStatus("loading");  // valid
setStatus("pending");  // error
```

Number literals:

```
function selectCard(cardNumber: 1 | 2 | 3 | 4): void {
  // ...
}

selectCard(2);  // valid
selectCard(5);  // error
```

Boolean literals:

```
type Mode = true | false;  // equivalent to just boolean, but useful when paired with other types
```

With Type Aliases

Use `type` aliases to name common literal unions:

```
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";
type LogLevel = "debug" | "info" | "warn" | "error";

function request(method: HttpMethod, path: string): void {
  // ...
}

function log(level: LogLevel, message: string): void {
  // ...
}
```

Type Narrowing with Literals

Literal types work with type narrowing:

```
function handleEvent(type: "click" | "hover" | "focus"): void {
  if (type === "click") {
    // type is narrowed to "click"
    console.log("clicked");
  } else if (type === "hover") {
    // type is narrowed to "hover"
    console.log("hovered");
```

```
  }
}
```

vs Enums

While TypeScript has `enum`, literal type unions are lightweight and often preferable:

```
// Literal union (preferred for simple cases)
type Direction = "north" | "south" | "east" | "west";

// Enum (more verbose, but useful for complex scenarios)
enum DirectionEnum {
  North = "north",
  South = "south",
  East = "east",
  West = "west",
}
```

Key Takeaway

Literal types restrict variables to specific exact values. Combine them with unions to create powerful, type-safe APIs that only accept intended values. They're a lightweight alternative to enums for many use cases.

Value Unions

Value unions combine literal types with union types to create precise, constrained APIs. They're particularly useful for parameters that should only accept a specific set of string or numeric values.

From Literal to Union

Start with a simple literal type:

```
function move(direction: "north") {
  // Implementation...
}
```

This function only accepts one value, which isn't very useful.

Expanding with Unions

Combine literal types using unions to allow multiple specific values:

```
function move(direction: "north" | "south" | "east" | "west") {
  // Implementation...
}
```

Now the function accepts exactly four directions, nothing more, nothing less.

Type Alias for Reusability

Extract the union into a type alias for better code organization:

```
type Direction = "north" | "south" | "east" | "west";

function move(direction: Direction) {
  // Implementation...
}
```

Benefits

Type Safety: TypeScript catches invalid values at compile time:

```
move("north");     // valid
move("northeast"); // Error: not assignable to type 'Direction'
```

Autocomplete: Editors provide intelligent suggestions for valid values.

Self-Documenting: The type clearly communicates what values are acceptable.

Refactoring: Changing the type definition updates all usages automatically.

Real-World Patterns

Status Values

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

function updateStatus(status: Status) {
  // Implementation...
}
```

HTTP Methods

```
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";

function request(method: HttpMethod, url: string) {
  // Implementation...
}
```

Numeric Unions

```
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;

function rollDice(): DiceRoll {
  return (Math.floor(Math.random() * 6) + 1) as DiceRoll;
}
```

When to Use

- Function parameters with a fixed set of options
- Configuration values with predefined choices
- State machines with known states
- API responses with specific status codes

Value unions create contracts between different parts of your code, making it impossible to pass invalid values while keeping the code readable and maintainable.

Template Literal Types

Template literal types are one of TypeScript's most powerful features for creating string-based type constraints. They use JavaScript's template literal syntax to generate unions of string literal types programmatically.

Basic Usage

Start with a simple union type:

```
type Class = "wizard" | "warrior" | "rogue";
```

Use template literal syntax to create derived types:

```
type Hero = `elf ${Class}`;
```

TypeScript automatically expands the template to all possible combinations:

```
// Equivalent to:
type Hero = "elf wizard" | "elf warrior" | "elf rogue";
```

Combining Multiple Unions

Template literals can combine multiple union types, creating a Cartesian product of all possibilities:

```
type Class = "wizard" | "warrior" | "rogue";
type Race = "elf" | "human" | "dwarf";
type Hero = `${Race} ${Class}`;
```

This generates all nine combinations:

```
// "elf wizard" | "elf warrior" | "elf rogue"
// | "human wizard" | "human warrior" | "human rogue"
// | "dwarf wizard" | "dwarf warrior" | "dwarf rogue"
```

Pattern Matching with Primitives

Use primitive types in templates to enforce string patterns:

```
type LogRecord = `${string}: ${number}`;
```

This creates a type that matches any string following the pattern "text: number":

```
const criticalErr: LogRecord = "CRITICAL: 69";      // valid
const warningMsg: LogRecord = "WARNING: 404";       // valid

const invalid1: LogRecord = "CRITICAL 92";          // Error: missing colon
const invalid2: LogRecord = "CRITICAL: 92a";        // Error: not a number
const invalid3: LogRecord = "92: CRITICAL";         // Error: number before string
```

Practical Applications

Event Names

```
type Events = "click" | "focus" | "blur";
type EventHandler = `on${Capitalize<Events>}`;
// "onClick" | "onFocus" | "onBlur"
```

CSS Properties

```
type Color = "red" | "blue" | "green";
type Shade = "light" | "dark";
type CssColor = `${Shade}-${Color}`;
// "light-red" | "light-blue" | "light-green" | "dark-red" | "dark-blue" | "dark-green"
```

API Endpoints

```
type Method = "GET" | "POST" | "PUT" | "DELETE";
type Resource = "users" | "posts" | "comments";
type Endpoint = `/${Resource}`;
type Route = `${Method} ${Endpoint}`;
// "GET /users" | "POST /users" | "PUT /users" | ...
```

Version Strings

```
type Version = `${number}.${number}.${number}`;

const validVersion: Version = "1.2.3";   // valid
const invalidVersion: Version = "1.2";   // Error: doesn't match pattern
```

Performance Considerations

Be cautious when combining large unions—the number of generated types grows exponentially:

```
// This creates 3 * 3 * 3 = 27 combinations
type A = "a" | "b" | "c";
type B = "x" | "y" | "z";
type C = "1" | "2" | "3";
type Combined = `${A}${B}${C}`;
```

For very large unions, consider alternative approaches to avoid performance issues.

When to Use

- Creating string enums with systematic naming patterns
- Enforcing string format constraints
- Building type-safe routing or event systems
- Generating related types from base unions
- Validating structured string formats like logs or IDs

Template literal types combine TypeScript's type system with string manipulation, enabling compile-time validation of string patterns that would otherwise require runtime checks.

Giant Unions

Complex union types can explode in size faster than expected. Understanding these limitations helps you design practical type systems.

The Problem

Consider building a `MoveMessage` type for a game:

```
type Distance = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
type Class =
  | "Warrior"
  | "Rogue"
  | "Mage"
  | "Cleric"
  | "Paladin"
  | "Druid"
  | "Hunter"
  | "Shaman";
```

```
type MoveMessage =
  `The ${Class} moves ${Distance}, ${Distance}, ${Distance}, ${Distance}, then ${Distance}.`;

const message: MoveMessage = "The Warrior moves 6, 2, 5, 4, then 7.";
```

This will likely produce an error:

```
Error: Expression produces a union type that is too complex to represent.
```

Why It Fails

The union above creates hundreds of thousands of combinations:

- 8 classes × 9^5 distances = **472,392 possible strings**

Even a simpler version with three distance values:

```
type MoveMessage =
  `The ${Class} moves ${Distance}, ${Distance}, then ${Distance}.`;
```

Still generates over **5,800 combinations** (8 × 9^3).

TypeScript has internal limits to prevent performance degradation. Extremely large unions:

- Slow down editor responsiveness
- Increase compilation time dramatically
- Consume excessive memory
- Make type-checking impractical

The Combinatorial Explosion

Template literal types with multiple unions create a Cartesian product:

```
// 2 × 2 = 4 combinations
type Small = `${("a" | "b")} ${("x" | "y")}`;

// 3 × 3 × 3 = 27 combinations
type Medium = `${("a" | "b" | "c")} ${("x" | "y" | "z")} ${("1" | "2" | "3")}`;

// 10 × 10 × 10 × 10 = 10,000 combinations
type Large = `${TenOptions} ${TenOptions} ${TenOptions} ${TenOptions}`;
```

Growth is exponential, not linear.

Practical Solutions

Use Simpler Types

Sometimes a basic `string` is the right choice:

```
type MoveMessage = string;

// Or add minimal constraints
type MoveMessage = `The ${string} moves ${string}`;
```

Runtime Validation

Move validation from compile-time to runtime when types become impractical:

```
type MoveMessage = string;
```

```
function validateMoveMessage(msg: MoveMessage): boolean {
  const pattern = /^The (Warrior|Rogue|Mage|...) moves \d, \d, \d, \d, then \d\.$/;
  return pattern.test(msg);
}
```

Builder Patterns

Use functions to construct valid values instead of exhaustive type unions:

```
type Class = "Warrior" | "Rogue" | "Mage";
type Distance = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

function createMoveMessage(
  characterClass: Class,
  distances: Distance[]
): string {
  return `The ${characterClass} moves ${distances.join(", then ")}.`;
}
```

Branded Types

Use branded types for nominal typing without exhaustive unions:

```
type MoveMessage = string & { readonly __brand: "MoveMessage" };

function createMoveMessage(
  characterClass: Class,
  distances: Distance[]
): MoveMessage {
  return `The ${characterClass} moves ${distances.join(", then ")}.` as MoveMessage;
}
```

Guidelines

Keep unions manageable: Aim for dozens or hundreds of combinations, not thousands.

Consider alternatives: If your union would be huge, there's usually a better approach.

Balance precision with practicality: Overly specific types can harm developer experience.

Test compilation speed: If types slow down your editor, simplify them.

When to Stop

If you encounter "union type too complex" errors, it's a signal to reconsider your approach. TypeScript's type system is powerful, but it has pragmatic limits designed to maintain usability.

Not every constraint needs compile-time enforcement. Sometimes runtime validation or simpler types serve your codebase better than exhaustive type unions.

Unions and Literal Types

This section covers TypeScript's union types and literal types — powerful features for creating flexible yet type-safe APIs. Learn how to combine types, constrain values, and leverage advanced patterns like template literal types.

Overview

Union types allow values to be one of several types, while literal types restrict values to specific constants. Together, they enable precise type definitions that catch errors at compile time while maintaining code flexibility.

Topics Covered

01. Unions

Introduction to union types using the pipe (`|`) operator to allow multiple type possibilities. Covers type narrowing with `typeof` checks to safely work with different types in the same variable.

Key Concepts: - Union syntax: `string | number` - Type narrowing with `typeof` - Safely handling multiple types

02. Optional Parameters

Function parameters marked with `?` that can be omitted by callers, receiving `undefined` when not provided. Essential for creating flexible function signatures.

Key Concepts: - Optional parameter syntax: `param?` - Must follow required parameters - Equivalent to `param: Type | undefined`

03. Default Parameters

Provide fallback values for function parameters when callers don't supply them. TypeScript automatically infers types from default values.

Key Concepts: - Default value syntax: `param = defaultValue` - Automatic type inference - No need for `?` when defaults are provided

04. Literal Types

Constrain variables to specific literal values instead of broad types. Use exact strings, numbers, or booleans as types for maximum precision.

Key Concepts: - String literals: `"north"` - Numeric literals: `42` - Boolean literals: `true` - `const` declarations create literal types

05. Value Unions

Combine literal types with unions to create APIs accepting only specific values. Use type aliases for reusability and self-documenting code.

Key Concepts: - Combining literals: `"north" | "south" | "east" | "west"` - Type aliases for value sets - Autocomplete and compile-time validation - Real-world patterns: status values, HTTP methods

06. Template Literal Types

Generate unions of string types programmatically using template literal syntax. Automatically expand combinations and enforce string patterns.

Key Concepts: - Template syntax: `` `prefix ${Union}` `` - Cartesian products of multiple unions - Pattern matching with primitives - Event names, CSS properties, API endpoints

07. Giant Unions

Understand the limitations of complex union types. Learn when unions become impractical and how to use alternative approaches.

Key Concepts: - Combinatorial explosion in template literals - "Union type too complex" errors - Performance considerations - Practical alternatives: runtime validation, builder patterns, branded types

Arrays

Arrays in TypeScript are typed collections of elements. Use bracket notation to specify the element type: `string[]`, `number[]`, `boolean[]`, etc.

Basic Syntax

The simplest way to declare an array type is with bracket notation following the element type:

```
let colors: string[] = ["red", "green", "blue"];
let scores: number[] = [95, 87, 92];
let active: boolean[] = [true, true, false];
```

TypeScript automatically infers the array type from values:

```
let colors = ["red", "green"];  // inferred as string[]
let scores = [95, 87];           // inferred as number[]
```

Working with Arrays

Iterate over arrays using standard JavaScript patterns:

```
function logProducts(products: string[]) {
  for (let product of products) {
    console.log(`In stock: ${product}`);
  }
}

logProducts(["Laptop", "Mouse", "Keyboard"]);
// In stock: Laptop
// In stock: Mouse
// In stock: Keyboard
```

Use array methods with full type safety:

```
const temperatures: number[] = [72, 68, 75, 71, 76];

// map preserves type safety
const celsius = temperatures.map(f => (f - 32) * 5/9);
// celsius is inferred as number[]

// filter also maintains types
const hot = temperatures.filter(t => t > 73);
// hot is number[]
```

Array Methods

TypeScript provides type-safe access to all standard array methods:

```ts
const languages: string[] = ["TypeScript", "Python", "Rust"];

languages.push("Go");              // valid
languages.push(2024);              // Error: type mismatch

const first = languages[0];        // string
const total = languages.length;    // number

const list = languages.join(" → "); // "TypeScript → Python → Rust → Go"
```

Union Arrays

Arrays can contain union types when elements might be multiple types:

```ts
let mixed: (string | number)[] = ["hello", 42, "world", 100];

// Proper syntax: parentheses required
let mixed: (string | number)[];   // correct

// Without parentheses
let wrong: string | number[];     // wrong! means string OR number[]
```

Readonly Arrays

Prevent accidental modifications to arrays using `readonly`:

```ts
function displayTags(tags: readonly string[]) {
  // tags.push("new-tag");  // Error: push not allowed on readonly array
  console.log(tags[0]);      // valid
}

const config: readonly string[] = ["dev", "test", "prod"];
// config[0] = "staging";  // Error: cannot modify readonly array
```

Array of Objects

Arrays commonly hold objects with typed properties:

```ts
interface Book {
  title: string;
  pages: number;
}

function summarizeBooks(books: Book[]) {
  for (const book of books) {
    console.log(`${book.title} has ${book.pages} pages`);
  }
}

summarizeBooks([
  { title: "Clean Code", pages: 464 },
  { title: "Design Patterns", pages: 416 },
]);
```

Generic Array Type

TypeScript also supports the generic `Array<T>` syntax (less common):

```
let cities: Array<string> = ["Tokyo", "Paris"];
let populations: Array<number> = [37400068, 2161000];

// Equivalent to:
let cities: string[] = ["Tokyo", "Paris"];
let populations: number[] = [37400068, 2161000];
```

The bracket notation `T[]` is preferred for clarity and readability.

Common Mistakes

Wrong union syntax without parentheses:

```
// This means: string OR array of numbers
let wrong: string | number[];

// This means: array of strings or numbers (correct)
let correct: (string | number)[];
```

Forgetting readonly for immutable arrays:

```
// This allows mutations
let numbers: number[] = [1, 2, 3];
numbers.push(4); // allowed

// This prevents mutations
let immutable: readonly number[] = [1, 2, 3];
// immutable.push(4); // Error
```

Empty Arrays

Empty array literals need type hints:

```
let empty: string[] = [];     // valid
let empty = [];               // inferred as never[] (too permissive)

// Better: be explicit about intended type
let items: string[] = [];
```

Nested Arrays

Arrays can contain other arrays:

```
let board: number[][] = [
  [1, 0, 1],
  [0, 1, 0],
  [1, 1, 0],
];

// Or with generic syntax
let schedule: Array<Array<string>> = [
  ["Mon", "Wed", "Fri"],
  ["Tue", "Thu"],
];
```

Type Parameters

TypeScript offers two syntaxes for declaring array types: the bracket notation `T[]` and the generic type parameter syntax `Array<T>`. Both are equivalent and you'll see both in real-world code.

Two Equivalent Syntaxes

The bracket notation and generic syntax are interchangeable:

```
// Using bracket notation
function assignColors(name: string, colors: string[]): void {
  // Implementation...
}

// Using generic type parameter syntax
function assignColors(name: string, colors: Array<string>): void {
  // Implementation...
}
```

Both function declarations are identical in behavior and type safety.

Variable Declarations

Use either syntax when declaring variables:

```
// Bracket notation (preferred)
const colors: string[] = [
  "blue",
  "green",
  "purple",
  "red",
  "orange",
  "white",
  "black",
];

// Generic syntax
const measurements: Array<number> = [
  1000, 5000, 12000, 20000, 27000, 40000,
];
```

Both declarations are equally valid. Choose whichever feels more natural to your team.

When to Use Each Syntax

Bracket Notation (`T[]`)

Advantages: - Visually resembles an array with square brackets - Faster to type - More concise and readable - Preferred in most TypeScript codebases

```
const scores: number[] = [95, 87, 92];
const users: User[] = [...];
const flags: boolean[] = [true, false];
```

Generic Syntax (`Array<T>`)

Advantages: - Consistent with other generic types - Useful when working with complex generic types - May feel more familiar if coming from languages like Java or C#

```
const data: Array<Record<string, unknown>> = [...];
const promises: Array<Promise<string>> = [...];
const callbacks: Array<(x: number) => string> = [...];
```

Complex Types

For complex element types, either syntax works, but choose based on readability:

```
// Union types
const mixed1: (string | number)[] = [];
const mixed2: Array<string | number> = [];

// Function types
const handlers1: ((event: Event) => void)[] = [];
const handlers2: Array<(event: Event) => void> = [];

// Object types
const users1: { name: string; id: number }[] = [];
const users2: Array<{ name: string; id: number }> = [];
```

Nested Arrays

Both syntaxes can represent nested arrays:

```
// Bracket notation
let matrix1: number[][] = [[1, 2], [3, 4]];
let matrix2: string[][] = [["a", "b"], ["c", "d"]];

// Generic syntax
let matrix3: Array<Array<number>> = [[1, 2], [3, 4]];
let matrix4: Array<Array<string>> = [["a", "b"], ["c", "d"]];

// Mixed (avoid - confusing)
let matrix5: Array<number[]> = [[1, 2], [3, 4]];
```

Best Practices

Prefer `T[]` notation because: - It's more concise and readable - It's the dominant style in TypeScript communities - The square brackets visually represent an array - It's slightly faster to type

Use `Array<T>` when: - You're defining complex generic types (we'll cover this later) - Your team has established a standard for complex types - You need consistency with other generic syntax in your codebase

Real-World Examples

```
// API response handling
const apiData: string[] = [];
const users: Array<User> = [];

// Event handlers
const listeners: ((event: Event) => void)[] = [];
const promises: Array<Promise<Response>> = [];

// Configuration
const ports: number[] = [3000, 3001, 3002];
const environments: Array<"dev" | "test" | "prod"> = [];
```

Summary

- `T[]` and `Array<T>` are equivalent
- `T[]` is preferred for its simplicity and readability
- `Array<T>` becomes more useful when working with advanced generics
- Choose consistency within your codebase

Both syntaxes are valid. The choice is largely about consistency and readability rather than correctness.

Heterogeneous Arrays

Heterogeneous arrays contain elements of different types. TypeScript handles this elegantly by using union types, allowing you to create arrays that mix multiple types safely.

What Are Heterogeneous Arrays?

A heterogeneous array contains values of different types. Unlike languages that require all array elements to be the same type, TypeScript allows mixed-type arrays through union types:

```
// TypeScript infers the type as (string | number)[]
let mixedValues = [1, 2, "hello", "world"];

// Explicit type annotation
let data: (string | number)[] = [42, "text", 100, "more"];
```

Type Inference

TypeScript automatically infers union types from mixed-type array literals:

```
// Inferred as (string | number)[]
const values = [1, "two", 3, "four"];

// Inferred as (string | boolean)[]
const flags = ["active", true, "inactive", false];

// Inferred as (number | null)[]
const numbers = [1, 2, null, 4];
```

Working with Heterogeneous Arrays

When iterating over a heterogeneous array, you need to handle all possible types:

```
function process(item: string | number): void {
  if (typeof item === "string") {
    console.log(`Text: ${item.toUpperCase()}`);
  } else {
    console.log(`Number: ${item * 2}`);
  }
}

const data: (string | number)[] = [10, "hello", 20, "world"];
data.forEach(process);
// Number: 20
// Text: HELLO
// Number: 40
// Text: WORLD
```

Real-World Scenarios

API Response Data

```
// Mixed response data from an API
const responseData: (string | number | boolean)[] = [
  "success",
  200,
  true,
  "request_id_123",
  true,
];
```

Configuration Arrays

```
// Configuration with mixed types
const config: (string | number | boolean)[] = [
  "development",
  3000,
  true,
  "localhost",
  false,
];

function applyConfig(value: string | number | boolean): void {
  console.log(`Applying: ${value}`);
}

config.forEach(applyConfig);
```

Event Data

```
// Events with variable data
const logEntry: (string | number | Date)[] = [
  "error",
  500,
  new Date("2025-12-12"),
];

function logEvent(entry: string | number | Date): void {
  if (typeof entry === "string") {
    console.log(`Level: ${entry}`);
  } else if (typeof entry === "number") {
    console.log(`Code: ${entry}`);
  } else {
    console.log(`Time: ${entry.toISOString()}`);
  }
}

logEntry.forEach(logEvent);
```

Multiple Type Handling

For arrays with more than two types, ensure you handle all cases:

```typescript
// Three-type union
const mixed: (string | number | boolean)[] = [
  "text",
  42,
  true,
  "more text",
  100,
];

function handleMixed(value: string | number | boolean): void {
  if (typeof value === "string") {
    console.log(`String: ${value.length} chars`);
  } else if (typeof value === "number") {
    console.log(`Number: ${value}`);
  } else {
    console.log(`Boolean: ${value ? "true" : "false"}`);
  }
}

mixed.forEach(handleMixed);
```

Common Patterns

Tuple Alternative

For a fixed number of known types, consider tuples instead:

```typescript
// Heterogeneous array (variable length, mixed types)
const array: (string | number)[] = ["a", 1, "b", 2];

// Tuple (fixed length, specific types per position)
const tuple: [string, number, string] = ["a", 1, "b"];
```

Type Guards

Use type guards for safer type narrowing:

```typescript
function isSting(value: string | number): value is string {
  return typeof value === "string";
}

const data: (string | number)[] = ["hello", 42, "world"];

data.forEach((item) => {
  if (isString(item)) {
    console.log(item.toUpperCase());
  } else {
    console.log(item * 2);
  }
});
```

Best Practices

Be Explicit: Always annotate heterogeneous arrays explicitly:

```typescript
// Good - clear intent
const mixed: (string | number)[] = [1, "two", 3];
```

```
// Avoid - too permissive
const vague: any[] = [1, "two", 3];
```

Minimize Union Size: Keep unions manageable:

```
// Good - two types
const pair: (string | number)[] = [];

// Challenging - five types (hard to handle all cases)
const complex: (string | number | boolean | null | undefined)[] = [];
```

Use Type Narrowing: Always check types before using type-specific operations:

```
function process(value: string | number) {
  // error: value might be number
  // console.log(value.toUpperCase());

  // correct: check type first
  if (typeof value === "string") {
    console.log(value.toUpperCase());
  }
}
```

When to Use Heterogeneous Arrays

Good use cases: - Processing API responses with mixed data - Configuration arrays with various settings - Event data with different property types - Collections where element types vary logically

Avoid when: - You can use typed objects or interfaces instead - You have a fixed structure (use tuples) - Types are too numerous (simplify or use classes)

Summary

- Heterogeneous arrays use union types: `(T | U | V)[]`
- TypeScript infers union types from mixed-type literals
- Always use type guards to safely handle mixed types
- Consider tuples for fixed-length, fixed-type structures
- Keep unions simple and manageable for code clarity

Heterogeneous arrays give you the flexibility of dynamic languages while maintaining TypeScript's type safety.

Rest Parameters

Rest parameters allow a function to accept an indefinite number of trailing arguments and collect them into an array. They are denoted by three dots `...` before the parameter name.

Basic Syntax

```
function gatherParty(partyName: string, ...adventurers: string[]): string {
    return `${partyName} consists of: ${adventurers.join(", ")}`;
}

const msg = gatherParty("The Fellowship", "Frodo", "Sam", "Gandalf");
console.log(msg);
// "The Fellowship consists of: Frodo, Sam, Gandalf"
```

- `partyName`: regular parameter
- `...adventurers`: rest parameter, typed as `string[]`

Type Safety with Rest

Rest parameters are fully typed, so array operations are type-safe:

```
function sum(...nums: number[]): number {
    return nums.reduce((acc, n) => acc + n, 0);
}

sum(1, 2, 3);      // 6
// sum("4");      // Error: argument must be number
```

You can mix regular and rest parameters, but rest must be last:

```
function tag(label: string, ...values: (string | number)[]): string {
    return `${label}: ${values.join(", ")}`;
}

tag("ids", 10, 20, 30);
tag("names", "Aragorn", "Legolas");
```

Using Spread at Call Sites

Rest parameters are often paired with spread syntax when calling:

```
const party = ["Frodo", "Sam", "Gandalf"] as const;

gatherParty("The Fellowship", ...party);
```

Note: spread (`...array`) is used at call sites; rest (`...param`) is used in function definitions.

Optional + Rest

Optional parameters must appear before a rest parameter:

```
function log(prefix?: string, ...messages: string[]): void {
    const p = prefix ?? "LOG";
    messages.forEach(m => console.log(`${p}: ${m}`));
}

log(undefined, "started", "running");
log("WARN", "disk low");
```

Tuples with Rest

Rest parameters align well with tuple types for structured variadic functions:

```
type Command = [name: string, ...args: (string | number)[]];

function execute(...cmd: Command): void {
    const [name, ...args] = cmd;
    console.log(`Executing ${name} with`, args);
}

execute("upload", "file.txt", 3);
```

Common Pitfalls

- Rest parameter must be last: `function f(a: string, ...rest: number[], b: string)` // Error
- Rest parameter is always an array: treat it as `T[]`, not a single `T`
- Avoid `any[]` for rest—prefer specific unions like `(string | number)[]`

Built-ins Using Rest

You've likely used rest parameters without noticing. For example, `console.log` accepts rest arguments:

```
// Declaration style
declare function consoleLog(...data: unknown[]): void;
consoleLog("Users", 3, { active: true });
```

Summary

- Define variable-length parameters with `...param: T[]`
- Rest must be the final parameter
- Combine with unions and tuples for flexible, type-safe APIs
- Spread at call sites complements rest in definitions

Don't confuse rest parameters with spread syntax: rest collects into an array in parameter lists; spread expands arrays/iterables at call sites.

Evolving Any

When you create a new empty array, TypeScript infers it as `any[]`. As you push values into it, the array's element type evolves to include the types of the pushed values.

Basic Behavior

```
let inventory = [];
// inventory: any[]

inventory.push(42);
// inventory: number[]

inventory.push("robe");
// inventory: (number | string)[]
```

If you explicitly type the array, TypeScript enforces that constraint:

```
let numbers: number[] = [];
numbers.push("robe");
// Error: Argument of type 'string' is not assignable to parameter of type 'number'
```

What Is "Evolving Any"?

"Evolving any" is a special type inference behavior for empty array literals. The array starts as `any[]`, then narrows as values are added, accumulating a union of all inserted element types.

- Starts as `any[]`
- After first push of a `T`: becomes `T[]`
- After subsequent pushes of different types: becomes `(T | U | ...)[]`

This is convenient when building arrays progressively, but it doesn't help restrict inputs within the immediate scope of construction.

Useful Pattern: Build Internally, Enforce Externally

Evolving any shines when you construct an array inside a function, then return it. Outside the function, the resulting type is fixed, so further mutations must respect the inferred union.

```
function getConfig() {
  let config = [];
  // config: any[]
  config.push("api-key");
  // config: string[]
  config.push(8080);
  // config: (string | number)[]
  return config;
}

const cfg = getConfig();
// cfg: (string | number)[]

cfg.push(false);
// Error: Argument of type 'boolean' is not assignable to parameter of type 'string | number'
```

Inside `getConfig`, you can push various types freely, and TypeScript evolves the type. Once returned, the element type is locked to the inferred union, preventing incompatible pushes.

Best Practices

- Prefer explicit types for public APIs to avoid surprises:

  ```
  const items: Array<string | number> = [];
  ```

- Use evolving any for local array construction where flexibility helps, then return a well-defined union.
- Avoid starting arrays as `any[]` if you already know the intended element type — annotate directly for clarity.

Common Pitfalls

- Starting with `any[]` may hide mistakes during construction. If the array is meant to be homogeneous, annotate it:

  ```
  const ids: number[] = [];
  ```

- Evolving any applies specifically to empty array literals. If initialized with values, TypeScript infers from those values instead:

  ```
  const mixed = [1, "two"]; // inferred as (number | string)[]
  ```

Summary

- Empty arrays start as `any[]` and evolve as values are pushed.
- Explicit annotations enforce element types immediately.
- Construct flexibly inside functions, but rely on the inferred union outside to maintain type safety.

Arrays

Typed arrays are foundational in TypeScript. This section covers array syntax, generic type parameters, mixing types safely, variadic functions with rest parameters, and the special “evolving any” inference.

Topics

- 01. Arrays: Basic `T[]` syntax, inference, methods, unions, readonly arrays, arrays of objects, generic `Array<T>` alternative, nested arrays, and common mistakes.
- 02. Type Parameters: `T[]` vs `Array<T>`—equivalent syntaxes, when to use each, complex and nested types, and best practices.
- 03. Heterogeneous Arrays: Mix different element types with unions, type guards, tuple alternatives, real-world scenarios, and pitfalls.
- 04. Rest Parameters: Variadic functions using `...param: T[]`, mixing with unions/optionals, spread at call sites, tuples with rest, and common errors.
- 05. Evolving Any: How empty arrays infer `any[]` and evolve as you push values; patterns to build internally and enforce types externally.

Quick Reference

- `T[]`: Preferred array type syntax (concise, readable).
- `Array<T>`: Generic alternative, useful near other generics.
- `(A | B)[]`: Array of union element types; parentheses required.
- `readonly T[]`: Prevent mutations (no `push`, no index assignment).
- `...args: T[]`: Rest parameter collects trailing arguments into an array.
- Empty `[]`: Starts as `any[]`, evolves as values are pushed.

Common Patterns

```
// Arrays of objects
interface User { name: string; age: number }
const users: User[] = [{ name: "Ada", age: 36 }];

// Union element arrays
const values: (string | number)[] = ["id", 42];

// Readonly arrays
function logLevels(levels: readonly string[]) {
  console.log(levels.join(", "));
}

// Rest + spread
function sum(...nums: number[]): number {
  return nums.reduce((a, n) => a + n, 0);
}
const parts = [1, 2, 3] as const;
sum(...parts);

// Evolving any inside a builder function
function build(): (string | number)[] {
  const acc = [];
  acc.push("token");
  acc.push(8080);
  return acc; // inferred as (string | number)[] outside
}
```

Best Practices

- Prefer `T[]` for readability; use `Array<T>` near other generics.
- Be explicit with union element arrays: `(A | B)[]` (use parentheses).

- Use `readonly` to signal immutability and prevent accidental mutation.
- Rest must be the last parameter; type it as `T[]`.
- Avoid `any[]` unless intentionally building a union—otherwise annotate the element type.
- Consider tuples for fixed-length, position-specific heterogeneous data.

Arrays in TypeScript combine JavaScript flexibility with compile-time safety. Use the right syntax and patterns to keep your collections clear, efficient, and maintainable.

Object Literal Types

Object literal types describe the shape and structure of objects. They are one of TypeScript's most powerful upgrades from JavaScript, catching property typos and enforcing correct shapes at compile time.

Basic Object Type Syntax

Define an object type inline using literal syntax:

```
function displayCar(car: { brand: string; year: number }) {
  console.log(`${car.brand} from ${car.year}`);
}

displayCar({ brand: "Tesla", year: 2024 });
// Tesla from 2024

displayCar({ brand: "BMW", model: "M3" });
// Error: Object literal may only specify known properties, and 'model' does not exist
```

Defining Type Aliases

For reusability and clarity, extract object types into type aliases:

```
type Movie = {
  title: string;
  releaseYear: number;
};

function describeMovie(movie: Movie) {
  console.log(`${movie.title} (${movie.releaseYear})`);
}

describeMovie({ title: "Inception", releaseYear: 2010 });
```

This approach keeps your code organized and makes the type reusable across functions.

Property Types

Object properties can be any valid TypeScript type:

```
type Article = {
  id: number;
  title: string;
  published: boolean;
  tags: string[];
  timestamps: { created: Date; modified: Date };
};
```

```
const article: Article = {
  id: 42,
  title: "Learning TypeScript",
  published: true,
  tags: ["typescript", "tutorial"],
  timestamps: {
    created: new Date("2025-01-01"),
    modified: new Date(),
  },
};
```

Optional Properties

Mark properties as optional with `?`:

```
type Restaurant = {
  id: number;
  name: string;
  cuisine?: string;       // optional
  rating: number;
  website?: string;       // optional
};

const pizzeria: Restaurant = {
  id: 1,
  name: "Mario's Pizzeria",
  rating: 4.5,
  // cuisine and website are optional
};

const sushi: Restaurant = {
  id: 2,
  name: "Tokyo Sushi",
  rating: 4.8,
  cuisine: "Japanese",
  website: "https://tokyosushi.com",
};
```

Readonly Properties

Prevent modifications to specific properties:

```
type DatabaseConfig = {
  readonly host: string;
  readonly database: string;
  connectionTimeout: number;  // not readonly
};

const dbConfig: DatabaseConfig = {
  host: "localhost",
  database: "production",
  connectionTimeout: 5000,
};

dbConfig.connectionTimeout = 10000;  // allowed
```

```
dbConfig.host = "remote-server";      // Error: Cannot assign to readonly property
```

Nested Objects

Objects can contain other objects:

```
type Location = {
  latitude: number;
  longitude: number;
  timezone: string;
};

type Store = {
  name: string;
  founded: number;
  location: Location;
};

const store: Store = {
  name: "Tech Hub",
  founded: 2020,
  location: {
    latitude: 40.7128,
    longitude: -74.0060,
    timezone: "EST",
  },
};
```

Using Unions in Objects

Combine union types with object properties for flexible shapes:

```
type Response = {
  status: "ok" | "failed";
  payload?: unknown;
  message?: string;
};

const success: Response = {
  status: "ok",
  payload: { count: 42, items: [] },
};

const error: Response = {
  status: "failed",
  message: "Permission denied",
};
```

Adding Methods to Objects

Object types can include methods:

```
type Logger = {
  info(msg: string): void;
  error(msg: string): void;
};
```

```
const logger: Logger = {
  info: (msg) => console.log(`[INFO] ${msg}`),
  error: (msg) => console.error(`[ERROR] ${msg}`),
};

logger.info("Server started");  // [INFO] Server started
```

Index Signatures

Allow dynamic property keys with a consistent value type:

```
type ScoreBoard = {
  [key: string]: number;
};

const scores: ScoreBoard = {
  alice: 950,
  bob: 875,
  charlie: 1050,
};

scores["diana"] = 920;  // allowed
scores["eve"] = "pending";  // Error: must be number
```

Comparing Object Literal Types

Two object types with the same shape are compatible:

```
type Coordinate = { x: number; y: number };
type Position = { x: number; y: number };

const coord: Coordinate = { x: 10, y: 20 };
const pos: Position = coord;  // allowed - same shape
```

Common Errors

Missing properties:

```
type Book = { title: string; isbn: string };
const book: Book = { title: "1984" };
// Error: Property 'isbn' is missing
```

Extra properties (with strict checking):

```
const item: Book = {
  title: "Brave New World",
  isbn: "978-0060850524",
  author: "Aldous Huxley",
};
// Error: Object literal may only specify known properties
```

Typos in property names:

```
const item: Book = {
  titel: "Dune",       // typo: 'titel' instead of 'title'
  isbn: "978-0441172719",
```

```
};
// Error: Object literal may only specify known properties
```

Best Practices

- Define object types as named type aliases for clarity and reuse
- Use optional properties (`?`) for truly optional data, not as a substitute for union types
- Apply `readonly` to properties that should not change after initialization
- Organize nested objects into separate types for readability
- Use index signatures sparingly—prefer exact property definitions when possible
- Avoid overly broad types like `{ [key: string]: any }`

Summary

- Object literal types describe the exact shape of objects
- Use type aliases for reusable object types
- Optional properties help model incomplete data
- Readonly properties enforce immutability
- TypeScript catches property typos and shape mismatches at compile time

Object types are the foundation of type-safe object-oriented code in TypeScript. They combine the flexibility of JavaScript objects with compile-time safety and excellent editor support.

Extra Properties

TypeScript distinguishes between passing objects through variables versus passing object literals directly. This distinction affects how "excess property checking" is applied.

The Rule

When passing objects to functions:

- **With a variable**: Extra properties are allowed (structural typing)
- **With an object literal**: Extra properties cause errors (excess property checking)

Variables with Extra Properties

When you assign an object to a variable and then pass it to a function, TypeScript allows extra properties:

```
type Dog = {
  name: string;
  breed: string;
};

function describeDog(dog: Dog) {
  console.log(`${dog.name} is a ${dog.breed}`);
}

// Create object with extra property
const myDog = {
  name: "Max",
  breed: "Golden Retriever",
  age: 3,  // extra property
};
```

```
// This is allowed
describeDog(myDog);
// Max is a Golden Retriever
```

The function `describeDog` only needs `name` and `breed`. Since we're passing through a variable, TypeScript allows the extra `age` property.

Object Literals with Extra Properties

When you pass an object literal directly (without a variable), TypeScript enforces strict excess property checking:

```
const myDog = {
  name: "Max",
  breed: "Golden Retriever",
  age: 3,
};

// Error: Object literal may only specify known properties,
// and 'age' does not exist in type 'Dog'
describeDog({
  name: "Max",
  breed: "Golden Retriever",
  age: 3,
});
```

TypeScript rejects the literal because it contains `age`, which is not in the `Dog` type.

Why the Difference?

With variables: TypeScript applies structural typing. If an object has at least the required properties, it's compatible, even if it has extras.

With literals: TypeScript applies excess property checking to catch typos and potential mistakes early.

```
// Scenario 1: Maybe you meant 'age' as a different property?
describeDog({ name: "Max", breed: "Golden Retriever", agee: 3 });
// Error helps catch this typo

// Scenario 2: Maybe you're using the wrong type?
describeDog({ name: "Max", breed: "Golden Retriever", owner: "Alice" });
// Error flags that 'owner' is unexpected
```

Workarounds

1. Use a Variable (Bypass Checking)

```
const input = {
  name: "Max",
  breed: "Golden Retriever",
  age: 3,
};

describeDog(input);  // allowed
```

2. Use Type Assertion (Explicitly Tell TypeScript)

```
describeDog({
  name: "Max",
  breed: "Golden Retriever",
  age: 3,
} as Dog);  // explicitly assert it's a Dog
```

3. Add the Property to the Type

```
type Dog = {
  name: string;
  breed: string;
  age?: number;  // make it optional
};

describeDog({
  name: "Max",
  breed: "Golden Retriever",
  age: 3,
});  // now allowed
```

Real-World Examples

API Request Bodies

```
type CreateUserRequest = {
  email: string;
  password: string;
};

function registerUser(req: CreateUserRequest) {
  // Process registration
}

// This works (variable)
const newUser = {
  email: "alice@example.com",
  password: "secure123",
  referralCode: "FRIEND2025",
};
registerUser(newUser);

// This fails (literal)
registerUser({
  email: "bob@example.com",
  password: "secure456",
  referralCode: "FRIEND2025",  // Error: not in type
});
```

Configuration Objects

```
type ServerOptions = {
  port: number;
  hostname: string;
};
```

```
function startServer(opts: ServerOptions) {
  // Start server
}

// Variable: allowed
const config = {
  port: 3000,
  hostname: "localhost",
  ssl: true,  // extra property
};
startServer(config);

// Literal: not allowed
startServer({
  port: 3000,
  hostname: "localhost",
  ssl: true,  // Error
});
```

Best Practices

Be aware of the behavior: Understand when excess property checking applies.

Prefer variables for flexibility: If you genuinely have extra properties, use a variable.

Update types when needed: If extra properties are intentional, add them to the type as optional.

Use `as const` for specific types:

```
const config = {
  port: 3000,
  hostname: "localhost",
} as const;

startServer(config);  // TypeScript infers exact literal types
```

Configuration Note

This behavior can be configured in `tsconfig.json`. We'll cover TypeScript configuration later in the course. For now, remember the default behavior:

- Variables: extra properties allowed
- Object literals: extra properties rejected

Summary

- **Variables bypass excess property checking**: Extra properties are allowed if the required properties exist.
- **Object literals trigger strict checking**: Extra properties cause compile errors.
- **The difference helps catch typos**: Excess property checking is a feature, not a bug.
- **You have workarounds**: Use variables, type assertions, or update the type.

This behavior is a balance between flexibility (structural typing) and safety (catching mistakes early). Understanding when each applies helps you write TypeScript that catches bugs before they reach production.

Optional Object Properties

Optional properties allow you to define object types where some properties may or may not be present. Mark properties as optional using the `?` operator.

Basic Syntax

```
type Superhero = {
  name: string;
  strength: number;
  cape?: boolean;  // optional property
};

const batman: Superhero = {
  name: "Batman",
  strength: 85,
  cape: true,
};

const clark: Superhero = {
  name: "Superman",
  strength: 100,
  // cape is optional, so it can be omitted
};
```

The `cape?` property means the field is optional—it can be present or absent.

Optional Properties Are Union Types

When you mark a property as optional with `?`, TypeScript treats it as a union with `undefined`:

```
type Superhero = {
  name: string;
  cape?: boolean;
};

// cape is actually: boolean | undefined
```

This means accessing an optional property requires checking for `undefined`:

```
function useSuperpowers(hero: Superhero) {
  if (hero.cape) {
    console.log(`${hero.name} has a cape!`);
  } else {
    console.log(`${hero.name} has no cape.`);
  }
}
```

Difference from Explicit Union

Two approaches can look similar but have different implications:

```
// Approach 1: Optional property
type Hero1 = {
  name: string;
  power?: string;
};
```

```
// Approach 2: Union with undefined
type Hero2 = {
  name: string;
  power: string | undefined;
};

// Both are equivalent
const h1: Hero1 = { name: "Alice" };              // allowed
const h2: Hero2 = { name: "Bob" };                // Error: must include power

const h3: Hero2 = { name: "Bob", power: undefined };  // allowed
```

Key difference: Optional (`?`) allows omitting the property entirely. Union with `undefined` requires the property to exist (even if set to `undefined`).

Required vs Optional Properties

```
type Blog = {
  title: string;          // required
  author: string;         // required
  description?: string;   // optional
  publishedDate?: Date;   // optional
  viewCount?: number;     // optional
};

const post: Blog = {
  title: "Learning TypeScript",
  author: "Jane Doe",
  // description, publishedDate, and viewCount are optional
};

const fullPost: Blog = {
  title: "Advanced TypeScript",
  author: "John Smith",
  description: "Deep dive into advanced types",
  publishedDate: new Date("2025-01-15"),
  viewCount: 5420,
};
```

Accessing Optional Properties Safely

Always check for `undefined` before using optional properties:

```
type User = {
  id: number;
  name: string;
  email?: string;
  phone?: string;
};

function contactUser(user: User) {
  console.log(`User: ${user.name}`);

  // Check before using optional properties
```

```
  if (user.email) {
    console.log(`Email: ${user.email}`);
  }

  if (user.phone) {
    console.log(`Phone: ${user.phone}`);
  }
}

const guest: User = { id: 1, name: "Guest" };
contactUser(guest);  // no error, even without email or phone
```

Optional Chaining

Use optional chaining (`?.`) to safely access nested optional properties:

```
type Profile = {
  name: string;
  social?: {
    twitter?: string;
    github?: string;
  };
};

const profile: Profile = { name: "Alice" };

// Optional chaining safely accesses nested properties
const twitter = profile.social?.twitter;  // undefined, not an error
const github = profile.social?.github;    // undefined, not an error
```

Nullish Coalescing

Provide default values for optional properties using `??`:

```
type Settings = {
  theme?: string;
  fontSize?: number;
};

const userSettings: Settings = {};

const theme = userSettings.theme ?? "light";      // "light"
const fontSize = userSettings.fontSize ?? 16;     // 16

console.log(`Theme: ${theme}, Font: ${fontSize}`);
```

Best Practices

1. Minimize Optional Properties

Only mark properties optional if they're genuinely optional. Too many optional properties make code harder to reason about:

```
// Too permissive
type Profile = {
  name?: string;
```

```
  email?: string;
  bio?: string;
  website?: string;
};

// Better: require core fields
type Profile = {
  id: number;
  name: string;
  email: string;
  bio?: string;          // truly optional
  website?: string;      // truly optional
};
```

2. Avoid Deep Optional Nesting

Deeply nested optional properties create complexity:

```
// Hard to reason about
type Config = {
  database?: {
    connection?: {
      timeout?: number;
    };
  };
};

// Better: separate concerns
type DatabaseConnection = {
  timeout: number;
};

type DatabaseConfig = {
  connection?: DatabaseConnection;
};

type AppConfig = {
  database?: DatabaseConfig;
};
```

3. Use Discriminated Unions Instead

For complex optional cases, consider discriminated unions:

```
// Too many optionals
type Response = {
  success?: boolean;
  data?: unknown;
  error?: string;
  statusCode?: number;
};

// Better: discriminated union
type Response =
  | { status: "success"; data: unknown; statusCode: number }
  | { status: "error"; error: string; statusCode: number };
```

Common Mistakes

Don't Create Too Many Checks

```
// Problematic: many optional fields create many checks
type Order = {
  id: number;
  items?: string[];
  subtotal?: number;
  tax?: number;
  shipping?: number;
  discount?: number;
};

function calculateTotal(order: Order) {
  if (order.subtotal && order.tax && order.shipping) {
    return order.subtotal + order.tax + order.shipping - (order.discount ?? 0);
  }
  // What if only some are present?
}
```

Better approach:

```
type OrderPricing = {
  subtotal: number;
  tax: number;
  shipping: number;
  discount: number;
};

type Order = {
  id: number;
  items: string[];
  pricing?: OrderPricing;  // either all or none
};
```

Don't Confuse Optional with Nullable

```
// Optional: property may not exist
type Optional = { name?: string };
const obj1: Optional = {};  // allowed

// Nullable: property exists but can be null
type Nullable = { name: string | null };
const obj2: Nullable = { name: null };  // allowed
// const obj3: Nullable = {};  // Error: name is required
```

Real-World Example

```
type Article = {
  // Required fields
  id: number;
  title: string;
  content: string;
  author: string;
  createdAt: Date;
```

```
  // Optional fields
  coverImage?: string;
  summary?: string;
  tags?: string[];
  updatedAt?: Date;
  publishedAt?: Date;
};

function displayArticle(article: Article) {
  console.log(`${article.title} by ${article.author}`);

  if (article.coverImage) {
    console.log(`Image: ${article.coverImage}`);
  }

  if (article.summary) {
    console.log(`Summary: ${article.summary}`);
  }

  if (article.tags && article.tags.length > 0) {
    console.log(`Tags: ${article.tags.join(", ")}`);
  }
}
```

Summary

- Mark properties optional with `?`: `property?: Type`
- Optional properties are unions with `undefined`
- Always check for `undefined` before using optional properties
- Use optional chaining (`?.`) and nullish coalescing (`??`) for safe access
- Minimize optional properties to reduce complexity and runtime checks
- Prefer required properties by default; make exceptions intentional

Optional properties are powerful for modeling real-world data structures, but use them judiciously. A well-designed type with mostly required properties is easier to work with than one with many optional fields requiring constant checks.

Empty Object Type

Say I innocently create a new empty object:

```
let newUser = {};
```

Then go to add properties to it later:

```
// Property 'name' does not exist on type '{}'
newUser.name = "Predrag";
```

TypeScript doesn't like that!

It makes sense—we never told TypeScript which properties to allow… but here's what's really crazy: this is actually allowed:

```
let newUser = {};
newUser = "Predrag";
```

Yup. You can reassign the variable, which initially held an empty object to a string. In fact, you can reassign it to anything except `null` or `undefined`, because everything else is technically an object!

So, to get back to our first example, what you probably want to do is just predefine the allowed field(s):

```
type User = {
    name: string;
};

let newUser: User = {
    name: "Predrag",
};
```

Defining the shape up front ensures TypeScript enforces the intended properties and prevents unsafe reassignment.

Discriminated Unions

A union of two primitive types, like `string | number`, is straightforward: it's a string or a number. The same is true of object types, but it can be tricky to know which shape you're dealing with at runtime. That's where discriminant properties (or "tags") help. A discriminant is a property whose value uniquely identifies the variant, making it easy to write conditional logic. Importantly, the discriminant can only be one specific value in each variant.

Unions of objects with a discriminant property are called discriminated unions (or tagged unions).

```
type MultipleChoiceLesson = {
    kind: "multiple-choice"; // Discriminant property
    question: string;
    studentAnswer: string;
    correctAnswer: string;
};

type CodingLesson = {
    kind: "coding"; // Discriminant property
    studentCode: string;
    solutionCode: string;
};

type Lesson = MultipleChoiceLesson | CodingLesson;

function isCorrect(lesson: Lesson): boolean {
    switch (lesson.kind) {
        case "multiple-choice":
            return lesson.studentAnswer === lesson.correctAnswer;
        case "coding":
            return lesson.studentCode === lesson.solutionCode;
    }
}
```

Discriminated unions shine when you add new variants, because TypeScript helps ensure you handle all cases. For example, if we introduce another lesson type:

```
type TrueFalseLesson = {
    kind: "true-false"; // Discriminant property
    question: string;
    studentAnswer: boolean;
    correctAnswer: boolean;
```

```
};

type Lesson = MultipleChoiceLesson | CodingLesson | TrueFalseLesson;
```

TypeScript will now complain: “Function lacks ending return statement and return type does not include ‘undefined’ ”, nudging you to handle the new case.

```
function isCorrect(lesson: Lesson): boolean {
    switch (lesson.kind) {
        case "multiple-choice":
            return lesson.studentAnswer === lesson.correctAnswer;
        case "coding":
            return lesson.studentCode === lesson.solutionCode;
        case "true-false":
            return lesson.studentAnswer === lesson.correctAnswer;
    }
}
```

Do you have to use `kind` as the tag? Not technically. Should you use `kind`? Yes—it’s the common convention and keeps code consistent.

Sets

TypeScript has a built-in type for sets, which are collections of unique values. You can ensure that all the values in the set are of the same type by specifying a type parameter `<T>`.

```
// A Set that contains only strings
const justiceLeague = new Set<string>();

justiceLeague.add("Green Arrow");
justiceLeague.add("Flash");

// Error: Argument of type '2' is not assignable to parameter of type 'string'
justiceLeague.add(2);
```

An array can be converted into a set, which automatically removes duplicate values:

```
// A Set automatically removes duplicate values from an array
const names = ["plasticman", "firestorm", "plasticman"];
const justiceLeague = new Set<string>(names);

console.log(justiceLeague);
// Set { 'plasticman', 'firestorm' }
```

Sets also have a few other useful methods and properties:

- `delete()`
- `has()`
- `forEach()`
- `size`

```
const justiceLeague = new Set<string>(["Atom", "Black Canary", "Blue Beetle"]);

console.log(justiceLeague.size); // 3

justiceLeague.delete("Blue Beetle");
console.log(justiceLeague.has("Blue Beetle")); // false

justiceLeague.forEach((member) => console.log(member));
```

```
// Atom
// Black Canary
```

Maps

TypeScript has a built-in `Map` type for collections of key-value pairs. You can specify the types of keys and values using type parameters `<K, V>`.

Creating Maps

```
// A Map with string keys and number values
const podracerSpeeds = new Map<string, number>();

podracerSpeeds.set("Anakin Skywalker", 947);
podracerSpeeds.set("Sebulba", 941);
```

TypeScript ensures type safety for both keys and values:

```
podracerSpeeds.set("R2-D2", true);
// Error: Argument of type 'true' is not assignable to parameter of type 'number'

podracerSpeeds.set(420, 69);
// Error: Argument of type 'number' is not assignable to parameter of type 'string'
```

Size Property

Maps use the `size` property instead of `length`:

```
console.log(podracerSpeeds.size);
// 2
```

Iterating Over Maps

You can easily iterate over a map using a `for...of` loop with destructuring:

```
for (const [racer, speed] of podracerSpeeds) {
  console.log(`${racer} raced at ${speed} speed`);
}
// Anakin raced at 947 speed
// Sebulba raced at 941 speed
```

Common Methods

`get()`

Retrieve a value by key:

```
console.log(podracerSpeeds.get("Sebulba"));
// 941
```

`has()`

Check if a key exists in the map:

```
console.log(podracerSpeeds.has("Sebulba"));
// true
```

`delete()`

Remove an entry by key:

```
podracerSpeeds.delete("Sebulba");
console.log(podracerSpeeds.get("Sebulba"));
// undefined
```

Dynamic Keys

Sometimes you don't know all of an object's property names in advance. For example, imagine building a customer management system where employees can add custom key/value pairs to customer records:

- `favoriteColor: "green"`
- `favoriteFood: "burger"`
- `favoriteAnimal: "dog"`
- etc.

You can't predict what users will add ahead of time, but you still want type safety for the data structure.

Index Signatures

You can define dynamic keys using an **index signature**:

```
type UserMetrics = {
  [key: string]: number;
};
```

This type says: "this object can have any number of properties where keys are strings and values are numbers."

Using Dynamic Keys

Once defined, you can create objects that conform to the type:

```
const metrics: UserMetrics = {
  wordsPerMinute: 50,
  errors: 2,
  timeOnPage: 120,
};
```

You can add new properties dynamically:

```
metrics["refreshRate"] = 60; // OK
metrics["theme"] = "dark"; // Error: Type 'string' is not assignable to type 'number'
```

TypeScript enforces the value type even for dynamically added keys, ensuring type safety throughout your code.

Dynamic Default Properties

There's a useful pattern you'll see in the wild that combines index signatures with specific properties:

```
type FormData = {
  [field: string]: string;
  email: string;
  password: string;
};
```

Why Require Specific Properties?

You might wonder why `email` and `password` are explicitly defined when the index signature already allows any string properties. The reason is to **require certain properties**.

This type says: - The object **must** have `email` and `password` properties - The object **can** have any number of additional string properties

Combining Different Value Types

You can also combine index signatures with different types:

```
type FormData = {
  [field: string]: string | number | boolean;
  email: string;
  password: string;
  age: number;
};
```

This type says: - The object **must** have `email` (string), `password` (string), and `age` (number) - The object **can** have any number of additional string, number, or boolean properties

Best Practices

Avoid overusing this pattern. Only use dynamic keys when you truly need unknown keys. If you have optional properties, use the `?` operator instead:

```
//  Prefer this for known optional properties
type FormData = {
  email: string;
  password: string;
  nickname?: string;
};

//  Avoid this unless you need truly dynamic keys
type FormData = {
  [field: string]: string;
  email: string;
  password: string;
};
```

PropertyKey

Dynamic keys are usually typed as `string`, but JavaScript objects can also use `number` and `symbol` keys. TypeScript provides the built-in `PropertyKey` union to cover every allowed key type:

```
// built-in type
type PropertyKey = string | number | symbol;
```

Using `PropertyKey` lets index signatures accept numbers and symbols in addition to strings:

```
type InfrastructureTags = {
    [key: PropertyKey]: any;
};

const janesServer: InfrastructureTags = {
    name: "Jane's Server",
    1: 420,
```

```
    [Symbol("role")]: "Admin",
};
```

A `symbol` is a unique, immutable value that can be used as a property key. Access symbol-keyed properties with the symbol itself and bracket notation; dot notation will not work.

```
const ROLE = Symbol("role");
const user = { [ROLE]: "Admin" };

user[ROLE]; // "Admin"
// user.ROLE; // undefined
```

Readonly Modifier

The `readonly` modifier works like `const` for object properties: once set, they cannot be reassigned.

```
type Point = {
    readonly x: number;
    y: number;
};

const point: Point = {
    x: 10,
    y: 20,
};

point.y = 30; // OK
// point.x = 15; // Error: Cannot assign to 'x' because it is a read-only property
```

Use `readonly` when a property should stay stable after initialization. It is most useful on data that rarely changes; overusing it can make updates noisy because you must create new objects instead of mutating existing ones.

"As Const" and Object.freeze

The `as const` assertion creates a deeply readonly type using literal values.

```
const priorities = ["low", "medium", "high"] as const;

// Error: Property 'push' does not exist on type 'readonly ["low", "medium", "high"]'
priorities.push("urgent");
```

It also works with objects and makes every nested property readonly automatically.

```
const configConst = {
    baseUrl: "https://api.library.io",
    featureFlags: {
        search: true,
        sync: false,
    },
    timeouts: [200, 500, 1000],
} as const;

// Error: Cannot assign to 'baseUrl' because it is a read-only property
configConst.baseUrl = "https://api.alt.io";
```

```
// Error: Cannot assign to 'search' because it is a read-only property
configConst.featureFlags.search = false;

// Error: Property 'push' does not exist on type 'readonly [200, 500, 1000]'
configConst.timeouts.push(1500);
```

Object.freeze()

`Object.freeze()` is a runtime helper that prevents top-level mutations of an object. TypeScript infers those top-level properties as readonly, but nested objects and arrays stay mutable unless you freeze them too.

```
const frozenProfile = Object.freeze({
    username: "neo",
    stats: {
        level: 5,
        xp: 8000,
    },
    badges: ["operator", "architect"],
});

// Error (type-level): Cannot assign to 'username' because it is a read-only property
frozenProfile.username = "trinity";

// OK: nested properties are not frozen automatically
frozenProfile.stats.level = 6;

// OK: the array itself is still mutable
frozenProfile.badges.push("sentinel");
```

Because `Object.freeze()` runs at runtime, attempts to change top-level properties will also fail at runtime (silent in non-strict mode, `TypeError` in strict mode). For deep immutability, combine `as const` or additional freezes on nested values.

Satisfies

The `satisfies` operator validates that a value matches a type without widening the inferred type. It solves a common dilemma:

- Type inference keeps narrow types but may miss structural errors
- Explicit type annotations catch errors but widen types and lose precision

Here's plain inference, which misses a typo:

```
const colors = {
  red: "#ff0000",
  green: "#00ff00",
  blue: 255,
  yelow: "#ffff00", // typo not caught
};
```

Adding an explicit type catches the typo but widens the value types:

```
type ColorMap = {
  red: string | number;
  green: string | number;
  blue: string | number;
  yellow: string | number;
```

```
};

const colorsTyped: ColorMap = {
  red: "#ff0000",
  green: "#00ff00",
  blue: 255,
  // Error: "yelow" is not in type ColorMap
  yelow: "#ffff00",
};

// redHex is widened to 'string | number'
type redHex = typeof colorsTyped.red;

// Error: Property 'toUpperCase' does not exist on type 'string | number'
const redUpper = colorsTyped.red.toUpperCase();
```

Using `satisfies` validates the structure while preserving narrowed types:

```
const colorsSatisfies = {
  red: "#ff0000",
  green: "#00ff00",
  blue: 255,
  yellow: "#ffff00",
  // Error: "yelow" is not in type ColorMap
  // yelow: "#ffff00"
} satisfies ColorMap;

// red stays narrowed to 'string'
type RedHexSatisfies = typeof colorsSatisfies.red;
const redUpper = colorsSatisfies.red.toUpperCase(); // "#FF0000"
```

Use `satisfies` when you need both structural validation and precise type inference.

Function Overloads

Function overloads let you constrain which parameter combinations are valid while maintaining flexibility. They provide type safety for functions that accept different argument patterns.

Define overload signatures above the implementation to specify allowed call patterns:

```
type Product = {
  id: string;
  name: string;
  price: number;
};

// Overload signatures
function createOrder(product: Product): string;
function createOrder(
  product: Product,
  expedited: true,
  deliveryDate: Date,
): string;

// Implementation signature
function createOrder(
```

```
  product: Product,
  expedited?: boolean,
  deliveryDate?: Date,
): string {
  if (!expedited) {
    return `Order for ${product.name} ($${product.price}) - Standard Shipping`;
  }
  return `Order for ${product.name} ($${product.price}) - Expedited: ${deliveryDate?.toLocaleDateString
}
```

Without overloads, the implementation would accept any combination of arguments. The overloads restrict calls to exactly two patterns: either one argument or three arguments where `expedited` must be `true`.

```
const laptop: Product = { id: "L123", name: "ThinkPad", price: 1200 };

// OK: one argument
const order1 = createOrder(laptop);
// Order for ThinkPad ($1200) - Standard Shipping

// OK: three arguments
const order2 = createOrder(laptop, true, new Date("2024-12-20"));
// Order for ThinkPad ($1200) - Expedited: 12/20/2024

// Error: No overload expects 2 arguments, but overloads do exist that expect either 1 or 3 arguments.
const order3 = createOrder(laptop, true);
```

Use overloads to enforce relationships between parameters, like requiring a delivery date when expedited shipping is requested.

Objects

Objects are fundamental to TypeScript. This section covers object literal types, structural typing rules, advanced patterns like discriminated unions, and practical tools for working with object data structures.

Contents

1. **Object Literal Types** — Define the shape and structure of objects with type aliases and inline annotations.
2. **Extra Properties** — Understand how TypeScript's excess property checking differs between variables and object literals.
3. **Optional Object Properties** — Use the `?` operator to make properties optional and handle `undefined` safely.
4. **Empty Object Type** — Learn why `{}` behaves unexpectedly and how to define proper empty objects.
5. **Discriminated Unions** — Use tagged unions with discriminant properties to handle multiple object shapes safely.
6. **Sets** — Work with typed `Set<T>` collections that enforce unique values.
7. **Maps** — Use `Map<K, V>` for type-safe key-value pairs with flexible iteration.
8. **Dynamic Keys** — Define objects with unknown property names using index signatures.
9. **Dynamic Default Properties** — Combine index signatures with required properties for flexible yet constrained types.

10. **PropertyKey** — Use the built-in `PropertyKey` union to accept string, number, and symbol keys.
11. **Readonly Modifier** — Prevent property reassignment with the `readonly` modifier.
12. **"As Const" and Object.freeze** — Create deeply readonly types with `as const` and compare with runtime `Object.freeze()`.
13. **Satisfies** — Validate object structure without widening types using the `satisfies` operator.
14. **Function Overloads** — Constrain parameter combinations with function overload signatures.

Tuples

A tuple is a fixed-length array where each position has a specific type.

```
const cityLocation: [string, number] = ["Tokyo", 13960000];
```

Tuples are safer than arrays for small, structured collections because the length is fixed and each index has a known type.

Be Explicit With Tuples

Always provide explicit tuple types. Without an annotation, TypeScript infers a general array type instead of a tuple.

```
// Tuple: [string, number]
const coordinates: [string, number] = ["Latitude", 40.7128];

// Array: (string | number)[]
const coordinates2 = ["Longitude", -74.0060];
```

An inferred array allows mutations that break the intended structure:

```
const productInfo = ["Laptop", 1299];
productInfo[1] = "Desktop"; // OK, but probably not what you want
```

With an explicit tuple type, TypeScript catches type mismatches:

```
const productInfo: [string, number] = ["Laptop", 1299];
// Error: Type 'string' is not assignable to type 'number'.
productInfo[1] = "Desktop";
```

Use tuples when you need a fixed structure with known types at each position.

Readonly

Tuples in TypeScript are still arrays under the hood, so counterintuitively you can still push to them and pop from them. This is a bit of a gotcha, tuples in most languages are fixed length.

```
const nameAndAge: [string, number] = ["Martha Jones", 24];
nameAndAge.push("Donna Noble");
```

So you still need to be careful about underlying array length… that is, unless you use readonly tuples, which is really the only way I use tuples.

```
const nameAndAge: readonly [string, number] = ["Martha Jones", 24];
// Error: Property 'push' does not exist on type 'readonly [string, number]'.
nameAndAge.push("Donna Noble");
```

Much better! I use `readonly` any time I possibly can, its kinda like using `const` over `let` whenever possible. However, keep in mind that `readonly` is TypeScript specific, which means it's enforced at compile time, but not at runtime (like `const` is).

Tuples vs. Objects

So why would you use a tuple instead of an object? For example, this:

```
function getCoordinates(): [number, number] {
    return [40.7128, -74.006]; // latitude, longitude
}
```

Instead of this:

```
function getCoordinatesAsObject(): { lat: number; lng: number } {
    return { lat: 40.7128, lng: -74.006 };
}
```

Coordinates have a conventional order (latitude, then longitude), so a tuple's positional semantics fit well. Objects are clearer for named access (e.g., `coords.lat`).

Counterexample

I would probably model a user as an object:

```
type User = { name: string; age: number; email: string };
const user: User = { age: 60, name: "Lane", email: "super@secret.com" };
```

A tuple like `[string, number, string]` would be confusing — `user[0]` for name? `user[2]` for email? Objects' descriptive keys are more intuitive.

Destructuring Tuples

Tuples pair nicely with destructuring when you want to return multiple, well-typed values from a function without introducing a new object type. The values are positional, and TypeScript keeps their types aligned with each position.

```
function getName(fullName: string): [string, string] {
    const parts = fullName.split(" ");
    return [parts[0], parts[1]];
}

const [firstName, lastName] = getName("Frodo Baggins");
```

The tuple return type `[string, string]` ensures `firstName` and `lastName` are both `string`. Destructuring makes the intent clear and avoids manual indexing like `value[0]`.

Nested Destructuring

You can destructure nested tuples and objects in a single statement. This is helpful when a tuple contains structured data.

```
type UserWithAddress = [string, { city: string; country: string }];

const userData: UserWithAddress = [
    "Aragorn",
    { city: "Minas Tirith", country: "Gondor" },
];
```

```
const [userName, { city, country }] = userData;
console.log(city); // "Minas Tirith"
```

In this example, `userName` is a `string`, and the destructured `city` and `country` variables are strongly typed as `string` properties from the nested object. The `console.log(city)` statement prints "Minas Tirith".

Tips

- Prefer explicit tuple types for clarity and safety.
- Use destructuring to name each position, which improves readability compared to numeric indices.
- Keep tuple sizes small; for larger records, a named object type is usually clearer.

Named Tuples

Position-based access in tuples can be hard to read. You can label tuple elements (often called "named tuples") to make types more descriptive in your editor and tooling.

```
// Unlabeled
type UserData = [string, number, boolean];

// Labeled (named tuple)
type UserDataLabeled = [name: string, age: number, isAdmin: boolean];
```

Labels make the type self-descriptive and improve editor hints:

```
// Your editor shows the full type:
// [name: string, age: number, isAdmin: boolean]
function getUser(): UserDataLabeled {
    return ["Frodo", 33, false];
}
```

Labels Are Documentation Only

Labels are for TypeScript tooling; they don't change how values are accessed. Tuple access and destructuring are still positional.

```
const user: [name: string, age: number] = ["Bilbo", 111];

// Destructure in reverse order
const [age, name] = user;
console.log(age); // "Bilbo"
console.log(name); // 111
```

Even though the labels say `name` then `age`, the variables you choose in destructuring don't matter—only the positions do. In this example, `age` is inferred as `string` and `name` as `number` because they map to the 0th and 1st elements respectively.

Tips

- Use labels to improve readability and editor tooltips.
- Remember: labels do not affect runtime or access order—tuples remain positional.
- Prefer objects for larger or evolving data structures; use tuples for small, fixed shapes.

Optional Elements in Tuples

Like object properties, tuple elements can be optional using the `?` modifier. This helps model values that may or may not be present while keeping the tuple's fixed shape.

```
type HttpResponse = [statusCode: number, data: string, error?: string];

// Both of these work!
const successResponse: HttpResponse = [200, "Success!"];
const errorResponse: HttpResponse = [404, "", "Resource not found"];
```

Optional Values Are Last

Just like optional function parameters, all required elements must come before optional elements.

```
// Invalid: required after optional
type BadResponse = [statusCode: number, data?: string, error: string];

// Valid: optional elements trail
type GoodResponse = [statusCode: number, data?: string, error?: string];
```

Optional Types Include `undefined`

Optional elements are automatically unioned with `undefined`.

```
type UserInfo = [name: string, age: number, address?: string];

function handleUserInfo(user: UserInfo) {
  const [name, age, address] = user;
  // name: string
  // age: number
  // address: string | undefined
}
```

Tips

- Keep optional elements at the end to avoid confusing access patterns.
- Destructure carefully; optional positions may be `undefined`.
- Prefer object types when you have many optional properties; objects are easier to extend and validate.

Tuple Rest Elements

TypeScript allows tuples to have a variable number of elements of a specific type using rest elements. This is handy when you want a tuple to have a fixed-length beginning but a flexible-length ending:

```
// A tuple with a rest element
type NameAndScores = [string, ...number[]];

// All of these are valid
const nameAndScores: NameAndScores = ["Alphonse", 69, 420, 300];
const nameAndScores: NameAndScores = ["Winry", 42];
const nameAndScores: NameAndScores = ["Edward"];
```

This idea of flexibly sized tuples honestly barely feel like tuples to me; it feels like arrays with some type constraints—but I digress.

One great use case for rest elements is modeling a command-line argument pattern:

```
type Command = [name: string, ...args: string[]];

const gitCommit: Command = ["git", "commit", "-m", "Add new feature"];
const npmInstall: Command = ["npm", "install", "typescript"];

// Function that handles commands
function executeCommand([cmd, ...args]: Command) {
    console.log(`Executing ${cmd} with arguments: ${args.join(", ")}`);
}
```

It says, "I need a command string, but everything after that is optional." Pretty neat. The whole point of a great type system is to more accurately (and narrowly) model the valid states of your program.

Tuples

Tuples are fixed-length arrays where each position has a specific type. They provide a way to express structured data with known types at each index position.

```
const cityLocation: [string, number] = ["Tokyo", 13960000];
```

Overview

This section covers all aspects of working with tuples in TypeScript:

1. **Tuples** – Basics of defining and using tuple types with explicit annotations
2. **Readonly** – Making tuples immutable with the `readonly` modifier
3. **Tuples vs. Objects** – When to use tuples versus object types
4. **Destructuring Tuples** – Extracting values from tuples with destructuring syntax
5. **Named Tuples** – Labeling tuple elements for better readability
6. **Optional Elements in Tuples** – Making tuple positions optional with `?`
7. **Tuple Rest Elements** – Allowing variable-length tuple endings with `...` syntax

Key Concepts

Why Tuples Matter

Tuples are safer than plain arrays for small, structured collections because: - Length is fixed and known - Each index has a specific, expected type - TypeScript can catch type mismatches at each position

Always Be Explicit

Without explicit tuple annotations, TypeScript infers a general array type instead of a tuple:

```
// Tuple (with annotation): [string, number]
const coordinates: [string, number] = ["Latitude", 40.7128];

// Array (inferred): (string | number)[]
const coordinates2 = ["Longitude", -74.0060];
```

Tuples vs. Objects

Choose tuples when order is conventional (e.g., `[latitude, longitude]`). Choose objects for named, self-documenting access (e.g., `{ lat, lng }`).

Readonly by Default

Consider using `readonly` tuples to prevent accidental mutations:

```ts
const nameAndAge: readonly [string, number] = ["Martha", 24];
// nameAndAge.push("Extra"); // Error!
```

Destructuring

Tuples work great with destructuring to name positional values:

```ts
const [firstName, lastName] = getName("Frodo Baggins");
```

Advanced Features

- **Named tuples** – Add labels to positions for clarity
- **Optional elements** – Mark positions as `?` (must be at the end)
- **Rest elements** – Use `...type[]` for variable-length endings

Intersections of Types

An intersection type combines multiple types into one with the `&` operator. The resulting intersection type satisfies all the component types simultaneously.

```ts
type IndividualContributor = {
  id: number;
  name: string;
  tasks: string[];
};

type Manager = {
  directReports: number[];
};

type GoodManager = IndividualContributor & Manager;

const predrag: GoodManager = {
  id: 1,
  name: "Predrag Milanovic",
  tasks: ["Fixing B*llsh*t code", "Vibe Coding"],
  directReports: [2, 3, 4],
};
```

A `GoodManager` must have all the properties of both an `IndividualContributor` and a `Manager`. When you intersect object types, TypeScript merges their properties:

```ts
type Point2D = {
  x: number;
  y: number;
};

type Point3D = Point2D & {
  z: number;
};

// Equivalent to:
// type Point3D = {
//   x: number;
//   y: number;
```

```
//   z: number;
// };
```

The Never Type

In TypeScript, the `never` type represents values that can't occur. It sounds useless, but it's actually a powerful tool for ensuring exhaustive handling of all cases.

Detecting Unhandled Cases

Consider a function that should handle 3 cases:

```
function handleStatusCode(code: 200 | 404 | 500) {
  if (code === 200) {
    console.log("OK");
    return;
  }
  if (code === 404) {
    console.log("Not Found");
    return;
  }
  throw new Error(`Unknown status code: ${code}`);
}
```

This function only handles `200` and `404`, but TypeScript doesn't complain. However, you can catch this by tracking type narrowing. After each conditional, the type of `code` is narrowed down:

```
function handleStatusCode(code: 200 | 404 | 500) {
  if (code === 200) {
    console.log("OK");
    return;
  }
  // code is now 404 | 500
  if (code === 404) {
    console.log("Not Found");
    return;
  }
  // code is now 500
  throw new Error(`Unknown status code: ${code}`);
}
```

Using Never for Exhaustive Checks

If you assign `code` to a variable of type `never`, TypeScript will complain unless `code` has actually been narrowed to no possible values:

```
function handleStatusCode(code: 200 | 404 | 500) {
  if (code === 200) {
    console.log("OK");
    return;
  }
  if (code === 404) {
    console.log("Not Found");
    return;
  }
  // Error: Type '500' is not assignable to type 'never'.
```

```
  const err: never = code;
  return err;
}
```

This error tells you that not all cases were handled. Fix it by handling every case properly:

```
function handleStatusCode(code: 200 | 404 | 500) {
  if (code === 200) {
    console.log("OK");
    return;
  }
  if (code === 404) {
    console.log("Not Found");
    return;
  }
  if (code === 500) {
    console.log("Internal Server Error");
    return;
  }
  // No errors! code is never
  const err: never = code;
  return err;
}
```

Now the type system confirms that all cases are covered. This pattern is invaluable for maintaining code safety as you add new union members.

Intersecting Incompatible Types

What happens when we intersect types with overlapping properties?

```
type Saiyan = {
  name: string;
  powerLevel: number;
};

type Human = {
  name: string;
  age: number;
};

type SaiyanHuman = Saiyan & Human;
```

We get this `SaiyanHuman` type that's the equivalent of:

```
type SaiyanHuman = {
  name: string;
  powerLevel: number;
  age: number;
};
```

It merges the properties of both `Saiyan` and `Human`, and because `name` overlaps, it safely combines the two types and appears once in the resulting type.

When Things Go Wrong

What happens if the `name` field were incompatible types? For example:

```ts
type Saiyan = {
  name: "goku" | "vegeta";
  powerLevel: number;
};

type Human = {
  name: "krillin" | "yamcha";
  age: number;
};

type SaiyanHuman = Saiyan & Human;
```

Now the `name` property can't possibly satisfy both! Humans must be `krillin` or `yamcha`, and Saiyans must be `goku` or `vegeta`. So, the `name` property in `SaiyanHuman` becomes `never`, which in turn makes the entire `SaiyanHuman` type `never`.

```ts
// Type '{}' is not assignable to type 'never'
const theLaneagen: SaiyanHuman = {};
```

It's TypeScript saying, "Hey, the `SaiyanHuman` type is impossible, do something else." Most of the time, the solution here is to redesign your types to avoid incompatible intersections and make sense.

Intersections vs. Unions

So we've covered how unions (`|`) and intersections (`&`) are both used to smoosh types together… but which should you use?

Unions

- Use the `|` operator (suspiciously similar to the logical OR operator)
- Widen the resulting type (more possible values)
- Useful for modeling mutually exclusive options or states

Intersections

- Use the `&` operator (like logical AND)
- Narrow the resulting type (fewer possible values)
- Useful for combining multiple constraints or adding more required properties to existing types

Example

```ts
type Human = {
  name: string;
  age: number;
};

type Elf = {
  name: string;
  ears: "pointy";
};

// Must have name, age, and pointy ears
type ElfHuman = Human & Elf;

// Must have name (shared between Human and Elf)
```

```
// Can have either age or ears (or both)
type ElfOrHuman = Human | Elf;
```

Key Takeaways

- Use unions to say your type is "this OR that"
- Use intersections to say your type is "this AND that", or sometimes more simply, "this with the additional properties of that"

Super Set Unions

So we know we can drastically narrow a primitive type like `number` by using a union of literal types. For example, maybe only 3 error codes are valid:

```
type ErrorSlugs = "OK" | "NOT_FOUND" | "INTERNAL_ERROR";
```

This works great if these are the only valid error codes, but what if:

- Any string can be used as an error slug
- `"OK"`, `"NOT_FOUND"`, and `"INTERNAL_ERROR"` are the most common values and we like to have them show up in autocomplete

TypeScript has a hacky way for us to express this: super set unions.

```
type ErrorCodes = "OK" | "NOT_FOUND" | "INTERNAL_ERROR" | (string & {});
```

Why Not Just Use `string`?

You might be wondering, "Why wouldn't I just use `string` - the set of allowed values is the same?"

And you're right, but there's one subtle difference. By adding `(string & {})`, TypeScript won't change which values are allowed. Any string is allowed. But it will still give us autocomplete in our editor for the values `"OK"`, `"NOT_FOUND"`, and `"INTERNAL_ERROR"`.

Intersections

Intersection types combine multiple types into one using the `&` operator, creating a type that must satisfy all component types simultaneously. This chapter explores how intersections work, their relationship to unions, and important edge cases to watch for.

Contents

1. **Intersections of Types** - Learn how to combine multiple types using the `&` operator and merge object properties
2. **The Never Type** - Understand how `never` helps ensure exhaustive case handling
3. **Intersecting Incompatible Types** - Discover what happens when intersecting types with conflicting properties
4. **Intersections vs. Unions** - Learn when to use `&` versus `|` for combining types
5. **Super Set Unions** - Master the technique for providing autocomplete hints while accepting any string value

Quick Reference

Basic Intersection Syntax

```
type A = { x: number };
type B = { y: string };
```

```
type C = A & B; // { x: number; y: string; }
```

Intersections vs. Unions

Feature	Intersection (`&`)	Union (`|`)
Operator	`&` (AND)	`|` (OR)
Effect	Narrows type	Widens type
Result	Must satisfy ALL types	Must satisfy ANY type
Use case	Combine requirements	Alternative options

Key Concepts

- **Intersection types** require values to satisfy all component types simultaneously
- **The `never` type** represents impossible values and helps catch unhandled cases
- **Incompatible intersections** result in `never` when property types conflict
- **Super set unions** like `"a" | "b" | (string & {})` provide autocomplete while accepting any string

Common Patterns

Combining Object Types

```
type Person = {
  name: string;
  age: number;
};

type Employee = {
  id: number;
  department: string;
};

type EmployeePerson = Person & Employee;
// Has all properties: name, age, id, department
```

Exhaustive Type Checking with `never`

```
function handleStatus(status: "success" | "error" | "pending") {
  if (status === "success") return " ";
  if (status === "error") return " ";
  if (status === "pending") return "...";

  // Ensures all cases handled
  const exhaustive: never = status;
  return exhaustive;
}
```

Super Set Union for Autocomplete

```
type StatusCode = 200 | 404 | 500 | (number & {});
// Accepts any number but suggests 200, 404, 500 in autocomplete
```

When to Use What

Use Intersections (`&`) when you want to: - Combine multiple object types - Add properties to an existing type - Create a type that must satisfy multiple constraints - Model “this AND that”

Use Unions (`|`) when you want to: - Represent alternative options - Model "this OR that" - Create types with multiple possible shapes - Narrow primitive types to specific literals

Important Notes

Incompatible Property Types: When intersecting types with the same property name but different types, the result is `never` if the types can't be satisfied simultaneously.

Never Type Safety: Use `never` assignments to ensure all union cases are handled exhaustively. The compiler will error if cases are missing.

Union Widening: Unlike intersections, unions widen the set of possible values. Choose based on whether you need more or fewer possible values.

Interfaces

There are two ways to define object types: the `type` keyword (as we've seen with type aliases) and `interface`:

```
type Superhero = {
  name: string;
  powers: string[];
  isAvenger: boolean;
};

interface Superhero {
  name: string;
  powers: string[];
  isAvenger: boolean;
}
```

I'm a huge fan of having multiple ways to do the same things in a language.

In 9/10 scenarios, they work the same way, but there are a few key differences that we'll cover in this chapter. For now, just know that I recommend using `type` in most cases, but there are a few scenarios where `interface` is the better choice, which we'll talk about later.

Extending Interfaces

This is the exception to the previous rule of thumb that "you should prefer `type` over `interface`". Interfaces are a bit better when it comes to extending other interfaces (inheriting properties).

With types, you use the `&` (intersection) operator to extend types:

```
type Character = {
  name: string;
  level: number;
};

type Wizard = Character & {
  spellbook: string[];
  mana: number;
};
```

With interfaces, you use the `extends` keyword:

```
interface Character {
  name: string;
```

```
  level: number;
}

interface Wizard extends Character {
  spellbook: string[];
  mana: number;
}
```

In both cases, a `Wizard` now has all four properties: `name`, `level`, `spellbook`, and `mana`.

Why Is "Interface Extends" Usually Better?

To quote Microsoft's wiki:

> Interfaces create a single flat object type that detects property conflicts, which are usually important to resolve! Intersections on the other hand just recursively merge properties, and in some cases produce `never`. Interfaces also display consistently better, whereas type aliases to intersections can't be displayed in part of other intersections. Type relationships between interfaces are also cached, as opposed to intersection types as a whole. A final noteworthy difference is that when checking against a target intersection type, every constituent is checked before checking against the "effective"/"flattened" type.

For this reason, extending types with `interfaces`/`extends` is suggested over creating intersection types.

Put simply, with interfaces the developer ergonomics are a bit better and compilation is a bit faster.

Extending Multiple Interfaces

TypeScript allows an interface to extend multiple interfaces simultaneously, enabling you to compose complex types from smaller, focused interfaces. This is particularly useful for creating flexible type hierarchies and promoting code reuse.

Basic Syntax

You can extend multiple interfaces by listing them after the `extends` keyword, separated by commas:

```
type Character = {
  name: string;
  level: number;
};

interface Magical {
  mana: number;
  castSpell(spell: string): void;
}

interface Physical {
  strength: number;
  attack(): void;
}

interface BattleMage extends Character, Magical, Physical {
  combineAttacks(): void;
}
```

The `BattleMage` interface now has all 7 properties and methods:

- `name` (from `Character`)
- `level` (from `Character`)
- `mana` (from `Magical`)
- `castSpell` (from `Magical`)
- `strength` (from `Physical`)
- `attack` (from `Physical`)
- `combineAttacks` (defined in `BattleMage`)

Using the Extended Interface

```
const player: BattleMage = {
  name: "Gandalf",
  level: 50,
  mana: 1000,
  strength: 45,
  castSpell(spell: string) {
    console.log(`Casting ${spell}!`);
  },
  attack() {
    console.log("Physical strike!");
  },
  combineAttacks() {
    console.log("Unleashing magical and physical fury!");
  }
};

player.castSpell("Fireball");     // Casting Fireball!
player.attack();                   // Physical strike!
player.combineAttacks();           // Unleashing magical and physical fury!
```

Real-World Example: User Permissions

Multiple interface extension is excellent for modeling systems with layered capabilities:

```
interface Identifiable {
  id: string;
  createdAt: Date;
}

interface Authenticatable {
  email: string;
  passwordHash: string;
  verifyPassword(password: string): boolean;
}

interface Permissioned {
  role: "admin" | "user" | "guest";
  permissions: string[];
  hasPermission(permission: string): boolean;
}

interface AdminUser extends Identifiable, Authenticatable, Permissioned {
  adminLevel: number;
  auditLog: string[];
}
```

An `AdminUser` must implement all properties and methods from all three parent interfaces, plus its own specific properties.

Mixing Types and Interfaces

You can extend both type aliases and interfaces, as shown in the first example where `BattleMage` extends `Character` (a type) along with two interfaces.

Conflict Resolution

If multiple parent interfaces define the same property with different types, TypeScript will raise an error:

```
interface A {
  value: string;
}

interface B {
  value: number;
}

//  Error: Interface 'C' cannot simultaneously extend types 'A' and 'B'
// Named property 'value' of types 'A' and 'B' are not identical
interface C extends A, B {
  extra: boolean;
}
```

However, if the types are compatible, TypeScript will create an intersection:

```
interface Vehicle {
  speed: number;
  move(): void;
}

interface Electric {
  speed: number;
  batteryLevel: number;
}

//  Works: both define speed as number
interface ElectricVehicle extends Vehicle, Electric {
  charge(): void;
}
```

Benefits of Multiple Extension

1. **Composition over inheritance**: Build complex types from simple, reusable interfaces
2. **Separation of concerns**: Each interface can represent a distinct capability
3. **Type safety**: TypeScript ensures all contracts are fulfilled
4. **Flexibility**: Objects can satisfy multiple interface contracts simultaneously

Overriding Interface Properties

You can replace a property from a base interface as long as the new type is **compatible** with the original.

```
interface Character {
    rank: string | number;
```

```
    name: string;
    level: number;
}

interface Wizard extends Character {
    // Narrowing is allowed because number is assignable to string | number
    rank: number;
    mana: number;
}
```

Trying to override with an incompatible type fails:

```
interface Character {
    rank: string;
    name: string;
    level: number;
}

interface Wizard extends Character {
    // Error: number is not assignable to string
    rank: number;
    mana: number;
}
```

Declaration Merging

Declaration merging is a feature where multiple declarations with the same name are automatically combined into one. While useful in niche cases, it's often a source of confusion and bugs. This is one reason why `type` is often preferred over `interface`.

How It Works

When you declare the same interface multiple times, all declarations merge:

```
interface Spaceship {
  name: string;
}

interface Spaceship {
  engines: number;
}

interface Spaceship {
  lightSpeed: boolean;
}
```

The above is equivalent to:

```
interface Spaceship {
  name: string;
  engines: number;
  lightSpeed: boolean;
}
```

Types Don't Merge

If you use `type` instead, redeclaring it is an error:

```
type Spaceship = {
  name: string;
};

// Error: Duplicate identifier 'Spaceship'
type Spaceship = {
  engines: number;
};
```

This prevents accidental redeclaration and makes the code more predictable.

Interfaces

Interfaces are one of two ways to define object types in TypeScript. The other way is using the `type` keyword with type aliases. In most scenarios they work the same way, but there are key differences that make each better in specific situations.

Basic Syntax

Both approaches define the same structure:

Type Alias:

```
type Superhero = {
  name: string;
  powers: string[];
  isAvenger: boolean;
};
```

Interface:

```
interface Superhero {
  name: string;
  powers: string[];
  isAvenger: boolean;
}
```

In general, `type` is recommended in most cases, but interfaces shine when extending other interfaces.

Extending Interfaces

Interfaces are superior when it comes to inheritance. Use the `extends` keyword to add properties from a base interface:

With Interfaces (Recommended):

```
interface Character {
  name: string;
  level: number;
}

interface Wizard extends Character {
  spellbook: string[];
  mana: number;
}
```

With Types (Using Intersection):

```
type Character = {
  name: string;
  level: number;
};

type Wizard = Character & {
  spellbook: string[];
  mana: number;
};
```

Interface extension is better because it creates a single flat object type, detects property conflicts, and has better performance during type checking.

Extending Multiple Interfaces

An interface can extend multiple interfaces simultaneously, enabling composition of complex types:

```
interface Character {
  name: string;
  level: number;
}

interface Magical {
  mana: number;
  castSpell(spell: string): void;
}

interface Physical {
  strength: number;
  attack(): void;
}

interface BattleMage extends Character, Magical, Physical {
  combineAttacks(): void;
}
```

The `BattleMage` now has all properties and methods from its parent interfaces.

You can also mix types and interfaces in inheritance:

```
interface BattleMage extends Character, Magical, Physical {
  // ...
}
```

Overriding Interface Properties

When extending an interface, you can override properties as long as the new type is **compatible** with the original:

```
interface Character {
  rank: string | number;
  name: string;
  level: number;
}

interface Wizard extends Character {
```

```
  // Narrowing is allowed-number is assignable to string | number
  rank: number;
  mana: number;
}
```

Overriding with an incompatible type causes an error:

```
interface Character {
  rank: string;
  name: string;
  level: number;
}

interface Wizard extends Character {
  // Error: number is not assignable to string
  rank: number;
  mana: number;
}
```

Declaration Merging

A unique feature of interfaces is **declaration merging**: multiple declarations with the same name are automatically combined.

```
interface Spaceship {
  name: string;
}

interface Spaceship {
  engines: number;
}

interface Spaceship {
  lightSpeed: boolean;
}

// Results in:
// interface Spaceship {
//   name: string;
//   engines: number;
//   lightSpeed: boolean;
// }
```

This can be useful in niche cases (like extending the global `Window` type in browsers), but it's often a source of confusion and bugs. This is another reason `type` is usually preferred—types cannot be redeclared:

```
type Spaceship = {
  name: string;
};

// Error: Duplicate identifier 'Spaceship'
type Spaceship = {
  engines: number;
};
```

When to Use Interfaces vs Types

- **Use `interface`**: When building inheritance hierarchies or extending other interfaces. Interfaces have better performance and error messages.
- **Use `type`**: For most other cases. Types are more flexible and declaration merging is prevented by default.

Enums

Enums are a set of defined constants. The simplest form of enum is a numeric enum:

```
enum Direction {
  North, // 0
  East, // 1
  South, // 2
  West, // 3
}

let myDirection: Direction = Direction.North;
console.log(myDirection); // Outputs: 0
```

The killer feature of enums is that in your code you can have nicely named identifiers like `Direction.North`, and under the hood you can have simple unique values, like `0`. TypeScript automatically increments the underlying values for us as we define new enums.

You can also explicitly set the values, and TypeScript will ensure they're unique:

```
enum StatusCode {
  OK = 200,
  Created = 201,
  BadRequest = 400,
  Unauthorized = 401,
  NotFound = 404,
}
```

Bidirectional Mapping

Numeric enums are bidirectional, which just means you can easily convert from the underlying value to the name and vice versa:

```
const directionValue: number = Direction.South;
// 2
const directionName: string = Direction[directionValue];
// "South"
```

String Enums

Numeric enums can be nice when:

- You actually want numbers
- You really want to eke out every last bit of performance (numbers use less memory than strings)

But often, string enums are easier to work with if you just want labels.

```
enum LogLevel {
  ERROR = "ERROR",
  WARN = "WARN",
```

```
  INFO = "INFO",
  DEBUG = "DEBUG",
}

function structuredLog(message: string, level: LogLevel) {
  console.log(`[${level}] ${message}`);
}

structuredLog("User not found", LogLevel.ERROR);
// Outputs: [ERROR] User not found
```

When to use string enums

When enums only exist within your code, numeric enums are totally fine. They start to get really hairy when you need to serialize them to JSON or store them in a database. There's nothing worse than debugging API responses and seeing this:

```
{
  "id": "94e83b65-ae9c-47f4-b788-d3f4fd085067",
  "name": "Lane",
  "user_type": 7
}
```

What the heck is 7?! String enums make your data much more readable and maintainable.

Enum Compilation

Unlike most TypeScript features, enums generate additional JavaScript code at runtime. Let's see what happens when we compile this enum:

```
enum Class {
  Rogue,
  Mage,
  Warrior,
  Priest,
}
```

We get a JavaScript object that looks like this:

```
var Class;
(function (Class) {
  Class[(Class["Rogue"] = 0)] = "Rogue";
  Class[(Class["Mage"] = 1)] = "Mage";
  Class[(Class["Warrior"] = 2)] = "Warrior";
  Class[(Class["Priest"] = 3)] = "Priest";
})(Class || (Class = {}));
```

This generated code creates the **bidirectional mapping** that we talked about before:

- From name to value: `Class["Rogue"] = 0`
- From value to name: `Class[0] = "Rogue"`

String Enum Compilation

Strings compile in a similar way:

```
enum Class {
  Rogue = "Rogue",
```

```
  Mage = "Mage",
  Warrior = "Warrior",
  Priest = "Priest",
}
```

Compiles to:

```
var Class;
(function (Class) {
  Class["Rogue"] = "Rogue";
  Class["Mage"] = "Mage";
  Class["Warrior"] = "Warrior";
  Class["Priest"] = "Priest";
})(Class || (Class = {}));
```

String enums do not support reverse mapping — the compiled JavaScript only maps from name to string value, not the other way around.

Const Enums

There's a special variant of enums, **const enums**, which are completely removed during compilation and replaced with their literal values. Unlike regular enums, they don't ship extra mapping code.

```
const enum Direction {
  North = "NORTH",
  East = "EAST",
  South = "SOUTH",
  West = "WEST",
}

const whereWinterComesFrom = Direction.North;
```

Const enums are more performant, but do come with some limitations:

No computed values

They can reference other enum members, but can't use arbitrary expressions.

This is okay, as it references enum members:

```
const enum FavoriteActor {
  BradPitt = "Brad Pitt",
  AngelinaJolie = "Angelina Jolie",
  BestCouple = FavoriteActor.BradPitt + " and " + FavoriteActor.AngelinaJolie,
}
```

This is **not** okay, as it uses an arbitrary expression:

```
const enum FavoriteActor {
  BradPitt = "Brad Pitt",
  AngelinaJolie = "Angelina Jolie",
  // const enum member initializers must be constant expressions
  BestCouple = getBestCouple(),
}
```

Mapping issues

Const enums don't have runtime representation, so getting the name from the number isn't possible.

```
const enum Direction {
  North,    // 0
  East,     // 1
  South,    // 2
  West,     // 3
}

const directionValue = Direction.West;
```

This will error:

```
// A const enum member can only be accessed using a string literal.(2476)
const directionName = Direction[directionValue];
```

If you use a string literal, it just returns the value again:

```
const directionValueAgain = Direction["West"];
// 3
```

When to use const enums

I'd only use const enums when I'm really concerned about performance and bundle size.

Enums vs. Union Types

Sometimes you'll see code model a fixed set of options with an `enum`:

```
enum CardSuit {
    Hearts = "Hearts",
    Diamonds = "Diamonds",
    Clubs = "Clubs",
    Spades = "Spades",
}
```

But you can represent the same idea with a union of string literals:

```
type CardSuit = "Hearts" | "Diamonds" | "Clubs" | "Spades";
```

Both approaches let you constrain values and get good editor tooling. The choice mostly comes down to trade-offs in refactoring ergonomics and runtime behavior.

Pros of Unions

- Consistency: You already use unions for complex types; using them for primitives keeps your type style consistent.
- Zero runtime: Unions are erased at compile time, so they add no JavaScript code or payload.
- Brevity: They're concise to declare and easy to extend or narrow.

Pros of Enums

- Refactor-friendly values: If you change an enum member's value (e.g., `"Hearts"` → `"hearts"`), you don't need to update every usage site—callers still reference `CardSuit.Hearts`.
- Numeric reverse mapping: Numeric enums provide reverse lookups, which can be handy in some scenarios.
- Namespaced context: `CardSuit.Hearts` can read clearer than a bare string like `"Hearts"`. That said, modern editors usually show the type on hover, so the benefit is modest.

Practical Guidance

- Prefer unions for most "label-like" sets where you don't need reverse mapping or extra runtime constructs.
- Reach for enums when you specifically want namespacing or numeric reverse mapping, or when changing the underlying serialized value without touching call sites is a priority.

Personally, I use unions over enums most of the time. Even Anders Hejlsberg (the creator of TypeScript) has noted that if they were starting over, enums might not be added. Video here.

Enums

Enums provide named constants for a fixed set of options. They come in two main flavors—numeric and string—and have unique runtime behavior compared to most TypeScript types.

Quick Primer

```
// Numeric enum (auto-incremented values, reverse mapping available)
enum Direction {
    North, // 0
    East,  // 1
    South, // 2
    West,  // 3
}

const d: Direction = Direction.North; // 0 under the hood
const name = Direction[d]; // "North"

// String enum (readable values, no reverse mapping)
enum LogLevel {
    ERROR = "ERROR",
    WARN = "WARN",
    INFO = "INFO",
    DEBUG = "DEBUG",
}

function log(msg: string, level: LogLevel) {
    console.log(`[${level}] ${msg}`);
}
```

Runtime and Compilation

Unlike most TypeScript features, enums emit JavaScript. Numeric enums generate code that supports reverse mapping (value → name). String enums emit name → value mappings only. This makes enums convenient, but they add runtime weight; unions do not. See 03-enum-compilation.md.

Const Enums

`const enum` inlines values and erases the enum at compile time—no runtime object is emitted. This is great for size/perf, but you lose reverse mapping and face restrictions (no arbitrary computed members). See 04-const-enums.md.

Enums vs. Union Types

Many enum use cases can be modeled as unions of string literals:

```ts
// Equivalent to a string enum for many cases
type CardSuit = "Hearts" | "Diamonds" | "Clubs" | "Spades";
```

- Prefer unions for label-like sets where you want zero runtime and concise syntax.
- Prefer enums when you need namespacing (`CardSuit.Hearts`), numeric reverse mapping, or you want to change underlying values without touching call sites.

More details in 05-enums-vs.-union-types.md.

In This Section

- 01-enums.md: Basics of numeric enums and reverse mapping.
- 02-string-enums.md: When and why to use string enums.
- 03-enum-compilation.md: What JavaScript enums compile to.
- 04-const-enums.md: Compile-time-only enums and their trade-offs.
- 05-enums-vs.-union-types.md: Choosing between enums and unions.

Narrowing Types

Type narrowing is the simple process of making a type more and more specific as you write your code. As a general rule (don't abuse it, for the love…) the more specific your types, the better. With narrower types:

- Your editor tooling will be more helpful
- Your code will self-document much better
- You'll catch more errors at compile time.

Conditional Narrowing

One of the coolest features of TypeScript is how smart it is about recognizing how types are being narrowed in "regular" code. For example:

```ts
type WitcherCharacter = {
  type: "witcher";
  name: string;
  magicPower: boolean;
};

type StarWarsCharacter = {
  type: "star-wars";
  name: string;
  forceSensitive: boolean;
};

type Character = WitcherCharacter | StarWarsCharacter;

function fight(player1: Character, player2: Character) {
  if (player1.type === "witcher" && player2.type === "witcher") {
    // I don't need to type cast (convert)
    // player1 and player2 to WitcherCharacter - TypeScript
    // does that automatically because this branch of the
    // conditional narrows the type
    fightWitcher(player1, player2);
  } else if (player1.type === "star-wars" && player2.type === "star-wars") {
    // same thing here
    fightStarWars(player1, player2);
  } else {
```

```
    throw new Error("Can't fight characters from different universes");
  }
}

function fightWitcher(player1: WitcherCharacter, player2: WitcherCharacter) {
  // witcher specific logic
}

function fightStarWars(player1: StarWarsCharacter, player2: StarWarsCharacter) {
  // star wars specific logic
}
```

Unknown Type

We've talked about the `any` type in TypeScript, and how it can represent anything - it's the "widest" type.

The `unknown` type can be used for similar purposes, but it's a much safer alternative because it forces you to explicitly assert the type before using it in a specific way.

The "any" Problem

The `any` type basically turns off TypeScript's type checking:

```
function processData(data: any) {
  // TypeScript allows this even though it might crash
  console.log(data.toLowerCase());

  // TypeScript allows this too - it's like we're using plain JavaScript
  return data.nonExistentMethod();
}

// No errors when calling the function
processData(42); // Will crash at runtime
```

When you take plain JavaScript code and run it through TypeScript tooling, almost everything is `any` by default.

The "unknown" Solution

The `unknown` type doesn't allow that kind of tomfoolery:

```
function processData(data: unknown) {
  // Error: Object is of type 'unknown'
  console.log(data.toLowerCase());

  // Error: Object is of type 'unknown'
  return data.nonExistentMethod();
}
```

With `unknown`, you can still assign any value to it (e.g. call this function with any value), but you can't use that value in a meaningful way without first checking its type:

```
function processData(data: unknown) {
  // We do a type assertion
  if (typeof data === "string") {
    // Now TypeScript knows data is a string
    console.log(data.toLowerCase());
```

```
    return data;
  }
  if (typeof data === "number") {
    // Now TypeScript knows data is a number
    return data * 2;
  }

  // Throw an error for other types
  // that we can't handle
  throw new Error("Expected string data");
}
```

When to Use unknown

`unknown` is a fantastic alternative to `any` when it comes to dealing with values that are coming into your program from the outside world (e.g. user input, API responses, etc.). It forces you to add type checks at that I/O boundary so that you can then be confident working with the data inside your program.

Type Hierarchy

All types in TypeScript can be arranged into a hierarchy. Understanding this hierarchy is fundamental to mastering type narrowing and type safety.

Understanding the Hierarchy

The diagram above illustrates how TypeScript types are organized:

- **Top of the hierarchy (widest)** - Types like `any` and `unknown` encompass the most possible values. Very little is known about them.
- **Middle levels** - Primitive types (`string`, `number`, `boolean`) and special types like `void`, followed by unions and literal values.
- **Bottom of the hierarchy (narrowest)** - Specific literal types (`"Armin"`, `69`, `true`) and finally `never`, which represents values that can't occur.

Assignability Direction

The arrows in the diagram show that **assignability flows upward**: - Types lower in the hierarchy are assignable to types above them - Types higher in the hierarchy are **not** assignable to types below them

Key Types in the Hierarchy

Top Types

- `any` (shown with dashed border) - The exceptional type that breaks all type safety rules. It allows you to do whatever you want, effectively opting out of type checking.
- `unknown` - The safe alternative to `any`. It's at the top but requires type narrowing before use.

Primitive Types

- `string` - All string values, including literal strings like `"Armin"` or `"Eren"`
- `number` - All numeric values, including specific numbers like `69`
- `boolean` - Includes both `true` and `false` literal values
- `void` - Represents the absence of a value (typically for functions that don't return)

Literal Types

Literal types represent specific, exact values: - String literals: `"Armin"`, `"Eren"` - Number literals: `69` - Boolean literals: `true`, `false`

Union Types

Types like `"Armin" | "Eren"` sit between their constituent literals and their primitive type.

Special Types

- **`undefined`** and **`null`** - Specific types that are assignable to `void` (when `strictNullChecks` is disabled)
- **`never`** - At the very bottom, representing values that can't occur

Assignability Rules

Following the diagram, types can be assigned **upward** (from narrower to wider) but **not downward**:

Example: String Hierarchy

```
// Following the path from bottom to top
const literal: "Armin" = "Armin";
const union: "Armin" | "Eren" = literal; //   "Armin" → "Armin" | "Eren"
const str: string = union;               //   "Armin" | "Eren" → string
const anything: unknown = str;           //   string → unknown

// Cannot go downward
const notAllowed: "Armin" = union;
//   Error: Type '"Armin" | "Eren"' is not assignable to type '"Armin"'
// What if the value happened to be "Eren"?
```

Example: Number and Boolean Hierarchies

```
// Number literals flow up to number
const specificNumber: 69 = 69;
const anyNumber: number = specificNumber; //   69 → number

// Boolean literals flow up to boolean
const t: true = true;
const b: boolean = t;                     //   true → boolean
```

Special Cases: `void`, `undefined`, and `null`

When `strictNullChecks` is disabled: - `undefined` and `null` are assignable to `void` - `void` is **not** assignable to `undefined` or `null`

```
// With strictNullChecks: false
function returnsVoid(): void {
  return undefined; //   Valid
}

const undef: undefined = returnsVoid(); //   Invalid
```

Why This Matters

Understanding the type hierarchy helps you:

1. **Predict type errors** - You'll know when assignments will fail
2. **Write safer code** - Choose the right type specificity for your needs
3. **Master type narrowing** - Move from wider types to narrower, more specific types
4. **Avoid any** - Use `unknown` when you need a top type but want type safety

The never Type: Everything Flows to Nothing

As shown at the bottom of the diagram, all types are assignable to `never`, but `never` isn't assignable to anything (except itself). This makes sense because `never` represents values that can't exist:

```
function throwError(): never {
  throw new Error("This never returns");
}

// All types can be assigned to never (theoretically)
// but you'll never actually have a never value to assign
const n: never = throwError(); // Function never returns
```

Narrowing Using In

The `in` operator checks if a property exists in an object, which is fantastic for type narrowing in object literals.

```
type TextMessage = {
  content: string;
  sentAt: Date;
};

type ImageMessage = {
  caption: string;
  sentAt: Date;
};

type VideoMessage = {
  duration: number;
  sentAt: Date;
};

type Message = TextMessage | ImageMessage | VideoMessage;

function displayMessage(message: Message) {
  if ("content" in message) {
    // TypeScript knows this is a TextMessage
    // because it's the only one with a 'content' property
    console.log(`Text content is: ${message.content}`);
  } else if ("caption" in message) {
    // TypeScript knows this is an ImageMessage
    // because it's the only one with an 'caption' property
    console.log(`Image caption is ${message.caption}`);
  } else {
    // TypeScript knows this is a VideoMessage because
    // it's the only other option
    console.log(`Video length is ${message.duration}`);
  }
}
```

Discriminated Unions vs. 'in' Checks

You might have noticed that this kind of logic feels very similar to using discriminated unions, and you're correct. Here's the same types with an explicit discriminant property:

```
type TextMessage = {
  kind: "text";
  content: string;
  sentAt: Date;
};

type ImageMessage = {
  kind: "image";
  caption: string;
  sentAt: Date;
};

type VideoMessage = {
  kind: "video";
  duration: number;
  sentAt: Date;
};
```

My recommendation is to prefer a discriminated union when you have full control of the types, but if you're using types from a library or package, or have another reason you don't want extra properties, the `in` operator is a great alternative.

Type Predicates

Sometimes the built-in type guards (`typeof`, `instanceof`, etc.) aren't enough.

TypeScript allows you to create your own type guards using type predicates. We do that by creating a function that:

- Accepts a wide type that we want to narrow
- Returns a boolean indicating if the value is of the desired type
- Uses the type predicate syntax `value is Type` in the return type

For example, here's a function that reports if a value is a string:

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

function processValue(value: unknown) {
  if (isString(value)) {
    // TypeScript knows value is a string here
    console.log(value.toUpperCase());
  }
}
```

For simple stuff like this, we could have just inlined the `typeof` check:

```
function processValue(value: unknown) {
  if (typeof value === "string") {
    // TypeScript knows value is a string here
    console.log(value.toUpperCase());
```

```
  }
}
```

But type predicates become really useful when the logic to check the type is a bit more complex. So we have a situation where one type, in this case, the `ManagerAdmin` type, shares properties with both other types:

```
interface ManagerAdmin {
  accessLevel: number;
  numEmployees: number;
}

interface Admin {
  accessLevel: number;
  payrollDate: Date;
}

interface Manager {
  numEmployees: number;
}
```

We can encapsulate the slightly more complex logic in a type predicate function:

```
function isManagerAdmin(
  boss: ManagerAdmin | Admin | Manager,
): boss is ManagerAdmin {
  return "numEmployees" in boss && "accessLevel" in boss;
}

// boss is a `ManagerAdmin | Admin | Manager`
if (isManagerAdmin(boss)) {
  // TypeScript knows boss is a ManagerAdmin here
  console.log(`Managing ${boss.numEmployees} employees`);
}
```

Exhaustive Checks

If you've ever heard a Rust enjoyer talk about how great the Rust programming language is, you've probably heard them mention "pattern matching" and "exhaustive checks".

To be fair, it's a pretty cool idea. Say we have this union type:

```
type Notif = "email" | "sms" | "push";
```

and we have this function that uses it:

```
function sendNotification(notif: Notif) {
  switch (notif) {
    case "email":
      return "Sending email";
    case "sms":
      return "Sending SMS";
    case "push":
      return "Sending push notification";
  }
  return "Unknown notification type";
}
```

This might be a very reasonable way to write JavaScript code, but that final `return "Unknown notification type";` is actually redundant in good TypeScript code. The switch statement is exhaustive, and TypeScript is smart enough to know that `return "Unknown notification type";` is actually unreachable code, and will give us a compiler error (assuming we have configured tsc to do so)!

Design your types so that you get these kinds of useful errors.

Guard Clauses

Guard clauses (a fancy way of saying "early returns") are my favorite way to quickly narrow types within a function. Peak production TypeScript code is often riddled with `undefined` and `null` types due to the nature of I/O and external APIs, so this is a classic pattern:

```
function processName(name: string | null | undefined) {
  if (name === null || name === undefined) {
    return "";
  }
  // TypeScript knows name is a string here
  return name.toUpperCase();
}
```

Now, an empty string keeps `processName`'s behavior straightforward (always returning a string), but depending on your use case, it might make more sense to throw an error instead:

```
function processName(name: string | null | undefined) {
  if (name === null || name === undefined) {
    throw new Error("Name is required");
  }
  // TypeScript knows name is a string here
  return name.toUpperCase();
}
```

Interestingly, throwing an error still narrows the type, but it doesn't change the function signature—this function still just returns a string. That's because errors in JavaScript and TypeScript are a control flow mechanism, not a type mechanism, so you do just kind of need to be aware, "hey this function can throw, I need to handle that".

In cases where my program won't break on an empty string, I might just coalesce to an empty string instead of throwing an error. This happens all the time with optional fields in web apps.

Type Assertion

Sometimes you know more about a value's type than TypeScript does… it's rare but it happens. The `as` keyword is the "trust me, bro" of TypeScript.

In the Boot.dev codebase, we have some places where we know a query parameter is a string, but Vue (our front-end framework) uses `string | string[]` for query params… which makes sense because query params can be arrays, but we know in many cases (because our back-end controls this) that it's always a string.

So, we have something like this:

```
// Property 'toLowerCase' does not exist on type 'string | string[]'
const userId = route.query?.userId.toLowerCase();
```

But we know it's never an array, so we just use `as string` to do this:

```
const userId = (route.query?.userId as string).toLowerCase();
```

We also capture values that come across the network as `unknown` and then use `as` to assert them into the shape we expect a given network response to be:

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

async function getUserRaw(userId: string): Promise<unknown> {
  const response = await fetch(`/api/users/${userId}`);
  return response.json();
}

export async function getUser(userId: string) {
  const data = await getUserRaw(userId);
  // here data is still just "unknown"
  // so we assert it to a User type
  return data as User;
}
```

Angle Bracket Syntax

There is an alternative syntax for type assertions using angle brackets and the type before the value:

```
const userIdRaw = <string>route.query?.userId;
const userId = userIdRaw.toLowerCase();
```

When to Do Type Assertions

- I try to avoid them. I'd rather use actual conditional narrowing over assertions unless I'm extremely confident. Conditional narrowing is safer because it doesn't involve assumptions.
- I prefer the `as` syntax over the angle bracket syntax. It's clearer, easier to read, and easier to write.

Double Assertion

TypeScript won't allow you to assert absolute nonsense:

```
const num = 42;

// Error: Conversion of type 'number' to type
// 'string' may be a mistake because neither
// type sufficiently overlaps with the other.
const str = num as string;
```

The `number` and `string` types have no overlap, making this assertion likely to be a mistake, so TypeScript complains. We can get around this with a double assertion:

```
const id = 42;

// This works - but is very unsafe!
const userId = id as unknown as string;

// Now TypeScript treats this as a string
console.log(userId.toUpperCase());
// Compiles, but still CRASHES at runtime!
```

I've never used this in production code. If you see this in the wild, pray that the author was a 100x engineer that knew what they were doing.

Non-Null Assertion

It's common for TypeScript libraries to assume that a value can be `null` or `undefined` even when you know it can't be. You can assert that it's not with the **non-null assertion** (`!`) operator. It tells the compiler that a value cannot be `null` or `undefined`, even when the type system thinks it might be.

```
// assume getCleanedText returns a string | null
import { getCleanedText, sendText } from "./text-utils";

const cleanedText = getCleanedText("some text");
// cleanedText is string | null
// but we know that it's not null because we passed in a valid string

// sendText expects a string, so we use a non-null assertion
sendText(cleanedText!);
```

You'll also see this fairly often when working with optional properties that you know exist:

```
interface User {
  id: string;
  name?: {
    first: string;
    last: string;
  };
}

// we don't control the User type (it's imported from a library)
// but we know that we always use the `name` property
sendText(user.name!.first);
```

The same rules-of-thumb as type assertions apply here. Only use non-null assertions when you're **absolutely confident** that the value can't be `null` or `undefined`. Just use a conditional guard clause if there is any uncertainty. This is always safer, albeit more verbose:

```
function sendTextSafely(text: string | null) {
  if (text === null) {
    throw new Error("Text is required");
  }
  sendText(text);
}
```

Type Narrowing

Type narrowing is the process of refining types from broader to more specific as you write code. It's one of TypeScript's most powerful features, enabling better tooling support, clearer self-documenting code, and catching more errors at compile time.

Overview

This section covers the essential techniques and concepts for working with type narrowing in TypeScript:

Core Concepts

- **Narrowing Types** - Introduction to conditional narrowing and how TypeScript automatically refines types in control flow
- **Unknown Type** - The safe alternative to `any` that forces explicit type checking
- **Type Hierarchy** - Understanding how types are organized from widest to narrowest and assignability rules

Narrowing Techniques

- **Narrowing Using In** - Property existence checks for discriminating object types
- **Type Predicates** - Creating custom type guards with `value is Type` syntax
- **Guard Clauses** - Using early returns to narrow types and handle edge cases
- **Exhaustive Checks** - Ensuring all union variants are handled in switch statements

Type Assertions (Use Sparingly)

- **Type Assertion** - Using `as` to override TypeScript's type inference
- **Double Assertion** - The escape hatch for asserting completely unrelated types
- **Non-Null Assertion** - Using `!` to assert values aren't `null` or `undefined`

Quick Reference

Conditional Narrowing

TypeScript automatically narrows types in conditionals:

```
type Response = { success: true; data: string } | { success: false; error: string };

function handleResponse(response: Response) {
  if (response.success) {
    // TypeScript knows: response is { success: true; data: string }
    console.log(response.data);
  } else {
    // TypeScript knows: response is { success: false; error: string }
    console.log(response.error);
  }
}
```

The `unknown` Type

Use `unknown` instead of `any` for safer external data handling:

```
function processData(data: unknown) {
  if (typeof data === "string") {
    return data.toUpperCase(); //  Safe
  }
  throw new Error("Expected string");
}
```

Type Hierarchy

Understanding assignability from narrower to wider types:

```
never → literals → unions → primitives → unknown
  ↑        ↑          ↑          ↑           ↑
narrowest                              widest
```

Property Checks with `in`

Discriminate object types by checking property existence:

```
type TextMsg = { content: string };
type ImageMsg = { imageUrl: string };
type Message = TextMsg | ImageMsg;

function display(msg: Message) {
  if ("content" in msg) {
    // TypeScript knows: msg is TextMsg
    console.log(msg.content);
  }
}
```

Custom Type Guards

Create reusable type predicates for complex checks:

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

function process(value: unknown) {
  if (isString(value)) {
    value.toUpperCase(); //  TypeScript knows it's a string
  }
}
```

Guard Clauses

Use early returns to narrow types and simplify control flow:

```
function getName(name: string | null | undefined): string {
  if (name === null || name === undefined) {
    return ""; // or throw an error
  }
  // TypeScript knows: name is string
  return name.toUpperCase();
}
```

Exhaustive Checks

Leverage TypeScript to ensure all union cases are handled:

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

function getStatusMessage(status: Status): string {
  switch (status) {
    case "idle":
      return "Ready";
    case "loading":
      return "Loading...";
    case "success":
      return "Done!";
    case "error":
      return "Failed";
```

```
  }
  // TypeScript knows all cases are covered
  // Adding a new status will cause a compile error here
}
```

Type Assertions (Use Cautiously)

Override TypeScript's inference when you have more information:

```
// Basic assertion
const userId = route.query?.userId as string;

// Non-null assertion
const user = findUser(id)!; // Assert it's not null/undefined

// Double assertion (rarely needed)
const value = input as unknown as TargetType;
```

Best Practices

1. **Prefer narrowing over assertions** - Use conditionals and type guards instead of `as` when possible
2. **Use `unknown` over `any`** - Force explicit type checking at I/O boundaries
3. **Design for exhaustiveness** - Structure unions so TypeScript catches missing cases
4. **Guard early, return early** - Handle edge cases at the start of functions
5. **Be specific with types** - Narrower types provide better tooling and catch more errors

Key Takeaway

Type narrowing is about progressively refining what you know about a value as your code executes. Master these techniques to write safer, more maintainable TypeScript code that leverages the full power of the type system.

Classes

Classes in TypeScript work mostly the same way that they do in JavaScript, but with the added benefit of static typing. One of the biggest differences is that you'll see type annotations on all the class properties at the top level of the class declaration.

```
class Hero {
  name: string;
  health: number;

  constructor(name: string, health: number) {
    this.name = name;
    this.health = health;
  }

  attack(damage: number): void {
    console.log(`${this.name} attacks for ${damage} damage!`);
  }

  getHealth() {
    return this.health;
  }
}
```

```
// Create an instance
const geralt = new Hero("Geralt", 100);
geralt.attack(25);
// "Geralt attacks for 25 damage!"
console.log(geralt.getHealth());
// 100
```

Key Points

- **Property Declarations**: Class properties must be declared at the top of the class with their types
- **Constructor**: The constructor method initializes the instance properties
- **Methods**: Class methods can have typed parameters and return types
- **Type Safety**: TypeScript ensures all property assignments and method calls are type-safe

Private Class Members

JavaScript added support for private class members in ES2022 with the # syntax. TypeScript respects that syntax and will give you compilation errors if you try to access private members outside of the class.

Example

```
class SecretAgent {
  // a private field
  #id: string;

  constructor(id: string) {
    this.#id = id;
  }

  // a public method
  getCodeName(): string {
    const idToCodeNameMap: Record<string, string> = {
      "007": "James Bond",
      "006": "Alec Trevelyan",
      // Add more mappings as needed
    };
    return idToCodeNameMap[this.#id] || "Unknown Agent";
  }
}

const bond = new SecretAgent("007");
console.log(bond.getCodeName()); // "James Bond"

// Property '#id' is not accessible outside class 'SecretAgent' because it has a private identifier.
console.log(bond.#id);
```

Key Benefit

In plain JavaScript, you'd only get the error at runtime, but with the same syntax in TypeScript, you get the error at compile time. Much better!

TypeScript Public and Private

JavaScript's `#` private fields didn't come until ES2022, but TypeScript developers had wanted public/private/protected access modifiers for a long time, so TypeScript added support for `private` and `protected` before then. As a result, a lot of older TypeScript code uses the keyword syntax.

Using the `private` Keyword

To create private members the TypeScript-only way, you use the `private` keyword:

```
class SecretAgent {
  // private field using the private keyword
  private id: string;

  constructor(id: string) {
    this.id = id;
  }

  // a public method
  getCodeName(): string {
    const idToCodeNameMap: Record<string, string> = {
      "007": "James Bond",
      "006": "Alec Trevelyan",
      // Add more mappings as needed
    };
    return idToCodeNameMap[this.id] || "Unknown Agent";
  }
}

const bond = new SecretAgent("007");
console.log(bond.getCodeName()); // "James Bond"

// Property 'id' is private and only accessible within class 'SecretAgent'
console.log(bond.id); // This will cause a compilation error
```

Which Syntax Should I Use?

The only reason TypeScript-specific syntax exists is because JavaScript didn't have the `#` syntax until ES2022. I recommend using the JavaScript `#` syntax because it's the JavaScript native way to do it. Remember, TypeScript is basically just tools for writing type-safe JavaScript, so conforming to JavaScript standards is the long-term play.

I only recommend using the TypeScript-specific syntax if you need to target an older version of JavaScript that doesn't support the `#` syntax yet.

Protected Data Members

The `protected` keyword is a TypeScript-only feature – it is not part of the ECMAScript standard. A `protected` member is accessible inside the class where it is declared, and inside any subclasses, but **not** from outside those classes.

You can think of `protected` as "`private`, but also accessible to subclasses".

```
class Character {
    protected health: number;

    constructor(health: number) {
```

```
        this.health = health;
    }

    protected takeDamage(amount: number): void {
        this.health -= amount;
        if (this.health < 0) {
            this.health = 0;
        }
    }
}

class Fighter extends Character {
    constructor(health: number) {
        super(health);
    }

    public fight(damage: number): void {
        // Can access protected members inherited from Character
        this.takeDamage(damage);
        console.log(`Fighter took ${damage} damage. Health: ${this.health}`);
    }
}

const fighter = new Fighter(100);
fighter.fight(30);

// Error: Property 'health' is protected and only accessible
// within class 'Character' and its subclasses
console.log(fighter.health);

// Error: Property 'takeDamage' is protected and only accessible
// within class 'Character' and its subclasses
fighter.takeDamage(10);
```

Here, `Fighter` can use the `health` property and `takeDamage` method because it extends `Character`. Code outside the class hierarchy cannot access those members.

Like `abstract`, the `protected` keyword has no direct equivalent in JavaScript at runtime. It exists only in TypeScript's type system to help you express and enforce class APIs. In JavaScript output, the `protected` keyword is erased.

In many codebases, you might see alternatives such as:

- Using `#` private fields when you want runtime-enforced privacy.
- Leaving members `public` when you are comfortable with subclasses and external code accessing them.

`protected` is most useful when you want to clearly signal an API that is meant only for subclasses, while still preventing access from the rest of your program at the type-checking level.

Abstract Classes and Methods

An abstract class is a class that cannot be instantiated directly. It acts as a template for subclasses, enforcing that they implement certain methods or properties.

Here is a simple `Shape` abstract class:

```
abstract class Shape {
    size: "small" | "medium" | "large";
```

```
    constructor(size: "small" | "medium" | "large") {
        this.size = size;
    }

    abstract calculateArea(): number;

    displayArea(): void {
        console.log(`The area of this shape is ${this.calculateArea()}`);
    }
}
```

Because `Shape` is abstract, you cannot create instances of it directly:

```
// Error: Cannot create an instance of an abstract class
const shape = new Shape("small");
```

Within an abstract class, abstract methods (like `calculateArea` above) declare a required method signature, but do not provide an implementation. Subclasses must implement these abstract methods. The class can still include regular methods (like `displayArea`) that are shared by all subclasses.

Here is a `Circle` class that extends `Shape` and provides the required `calculateArea` implementation:

```
class Circle extends Shape {
    radius: number;

    constructor(size: "small" | "medium" | "large") {
        super(size);

        if (this.size === "small") {
            this.radius = 5;
        } else if (this.size === "medium") {
            this.radius = 10;
        } else {
            this.radius = 15;
        }
    }

    calculateArea(): number {
        return Math.PI * this.radius * this.radius;
    }
}
```

Unlike `Shape`, the `Circle` class can be instantiated normally:

```
const circle = new Circle("medium");
circle.displayArea();
// The area of this shape is 314.1592653589793
```

Like `protected`, the `abstract` keyword does not exist in JavaScript at runtime. It is a TypeScript-only feature that helps enforce rules on subclasses at compile time. When your TypeScript is compiled to JavaScript, the `abstract` keyword and abstract-only information are removed from the output.

Classes Implement Interfaces

Classes can implement interfaces using the `implements` clause. This lets you say that a class must conform to the structure described by one or more interfaces.

Here are two simple interfaces:

```ts
interface Vehicle {
    make: string;
    model: string;
}

interface Drivable {
    drive(distance: number): void;
}
```

Now suppose we start with a class that has some of the required members:

```ts
class ElectricCar {
    make: string;
    model: string;
}
```

If we say that `ElectricCar` implements both `Vehicle` and `Drivable`, TypeScript will check that the class actually has all the properties and methods from those interfaces:

```ts
// Error: Class 'ElectricCar' incorrectly implements interface 'Drivable'.
class ElectricCar implements Vehicle, Drivable {
    make: string;
    model: string;
}
```

The error happens because `ElectricCar` does not yet have a `drive` method. Once we add it, the class correctly implements both interfaces:

```ts
class ElectricCar implements Vehicle, Drivable {
    make: string;
    model: string;

    // not required by the interfaces, but it's
    // fine to add extra properties
    private isRunning: boolean = false;

    constructor(make: string, model: string) {
        this.make = make;
        this.model = model;
        this.isRunning = false;
    }

    drive(distance: number): void {
        this.isRunning = true;
        console.log(`Driving ${distance} miles`);
    }
}
```

Implementing an interface does **not** prevent you from adding additional fields or methods; it only enforces that the required members exist and have compatible types.

Because `ElectricCar` implements both `Vehicle` and `Drivable`, you can use an instance anywhere either of those types is expected:

```ts
const myCar = new ElectricCar("Tesla", "Model S");

function testDrive(vehicle: Vehicle) {
    console.log(`Testing ${vehicle.make} ${vehicle.model}`);
```

```
}

testDrive(myCar); // "Testing Tesla Model S"

function takeForARide(drivable: Drivable) {
    drivable.drive(10);
}

takeForARide(myCar); // "Driving 10 miles"
```

This is a common pattern when you want classes to be interchangeable based on the behavior (interfaces) they support, instead of their specific implementation.

Classes vs. Interfaces and Types

When you design object shapes in TypeScript, you often have three different tools that can describe the same structure:

```
class Hero {
    name: string;
    health: number;
}

interface Hero {
    name: string;
    health: number;
}

type Hero = {
    name: string;
    health: number;
};
```

All three `Hero` definitions describe values with a `name` and `health`, but they come with different capabilities and tradeoffs.

When to Use Classes

If you come from an object-oriented background, you might prefer classes. Compared to interfaces and type aliases, classes can:

- Declare `private`, `protected`, `static`, and `abstract` members
- Define one or more constructors
- Provide concrete method implementations shared by all instances
- Participate in inheritance chains with `extends`

Because classes have runtime behavior (constructors, methods, static members), they are a good fit when you need both a *type* and a *runtime value* that work together.

When to Use Interfaces or Type Aliases

Interfaces and type aliases are purely compile-time constructs: they disappear after TypeScript compiles to JavaScript.

Some advantages of interfaces and type aliases are:

- No runtime overhead — they only exist for type checking
- Simpler to work with when you just need a shape for plain objects

- More flexible for modeling data that is not tied to a specific class
- Interfaces support extension and declaration merging in ways that classes and type aliases do not

In many codebases, it is common to start with interfaces or type aliases for data shapes, and introduce classes only when you need shared behavior, constructors, or other object-oriented features.

The `this` Type

The `this` keyword in JavaScript can be tricky to reason about, but TypeScript helps by giving `this` a precise type inside class methods.

Implicit `this` in Classes

Inside a class method, TypeScript automatically infers the type of `this` as the class itself:

```
class Counter {
    private count: number = 0;

    increment(): void {
        // `this` is implicitly typed as Counter
        this.count++;
    }

    getCount(): number {
        // `this` is implicitly typed as Counter
        return this.count;
    }
}
```

In most cases, this implicit typing is all you need — `this` is understood to be an instance of `Counter` whenever you call `counter.increment()` or `counter.getCount()`.

Explicit `this` Parameters

Sometimes you want more control over the type of `this`, for example when methods might be passed around as callbacks. TypeScript lets you declare an *explicit* `this` parameter to do exactly that.

The `this` parameter:

- Appears as the first parameter in a function or method signature
- Is only used for type checking; it does **not** exist at runtime

```
class Counter {
    private count: number = 0;

    increment(this: Counter, n: number): void {
        // `this` is explicitly typed as Counter
        this.count += n;
    }

    getCount(this: Counter): number {
        // `this` is explicitly typed as Counter
        return this.count;
    }
}

const counter = new Counter();
counter.increment(5);
```

```
console.log(counter.getCount());
// 5
```

Here, both `increment` and `getCount` declare `this: Counter`, so TypeScript can enforce that they are only called with a `Counter` instance as `this`, even if you pass them around as standalone functions.

Parameter Properties

TypeScript has a shorthand feature called *parameter properties* that lets you declare and initialize class properties directly in the constructor parameter list. This avoids having to both declare fields and then assign them inside the constructor body.

Without Parameter Properties

Normally, you might write a class like this:

```
class Hero {
    name: string;
    health: number;
    private level: number;

    constructor(name: string, health: number, level: number) {
        this.name = name;
        this.health = health;
        this.level = level;
    }
}
```

Here, each constructor parameter is manually copied into a class property.

With Parameter Properties

With parameter properties, you can get the same result with much less code:

```
class Hero {
    constructor(
        public name: string,
        public health: number,
        private level: number,
    ) {}
}
```

In this version, the constructor body `{}` is intentionally empty — the parameter properties are doing both the *declaration* and the *initialization* for you. Any other class members (methods, getters, setters, etc.) still belong in the class body, outside the constructor.

By adding an access modifier to a constructor parameter — `public`, `private`, `protected`, or `readonly` — TypeScript will automatically:

- Declare a property on the class with the same name and type
- Assign the argument passed to the constructor to that property

So the two `Hero` declarations above are equivalent from TypeScript's perspective.

Limitations

Parameter properties work with TypeScript's `private` keyword, but **not** with JavaScript's `#`-style private fields. If you need a `#` private field, you must declare and initialize it separately:

```
class Hero {
    #secretPower: string;

    constructor(
        public name: string,
        secretPower: string,
    ) {
        this.#secretPower = secretPower;
    }
}
```

In this example, `name` is a public parameter property, while `#secretPower` is a separate, truly private field initialized inside the constructor body.

Classes

Classes in TypeScript build on JavaScript's class syntax and add powerful type-system features: access modifiers, abstract classes, interface implementation, and more precise handling of `this` and constructor parameters.

Contents

1. **Classes** — Introduces TypeScript classes, including typed fields, constructors, and methods, and shows how static typing helps catch mistakes when creating and using class instances.
2. **Private Class Members** — Explains JavaScript's `#` private fields, how TypeScript understands them, and how compile-time errors prevent invalid access to truly private state.
3. **TypeScript Public and Private** — Covers TypeScript's `private` (and related) access modifiers, how they differ from `#` private fields, and guidance on when to prefer the JavaScript-native syntax.
4. **Protected Data Members** — Describes the TypeScript-only `protected` keyword, allowing subclasses (but not outside code) to access certain members, and contrasts it with `private` and public members.
5. **Abstract Classes and Methods** — Shows how to use `abstract` classes and members to define shared templates that cannot be instantiated directly but enforce implementations in subclasses.
6. **Classes Implement Interfaces** — Demonstrates using the `implements` clause so classes must satisfy one or more interfaces, enabling interchangeable behavior-based designs.
7. **Classes vs. Interfaces and Types** — Compares classes, interfaces, and type aliases for modeling object shapes, and explains when to reach for runtime-capable classes versus compile-time-only types.
8. **The `this` Type** — Explores how TypeScript types `this` inside class methods, including explicit `this` parameters that keep callbacks correctly typed when methods are passed around.
9. **Parameter Properties** — Introduces constructor parameter properties for concise field declaration and initialization, and notes their limitations with `#`-style private fields.

Single Source of Truth

As a TypeScript codebase grows, it's incredibly common to accumulate a *lot* of custom type definitions – hundreds of interfaces and type aliases that all describe roughly the same concepts, but with slightly different shapes.

For example, you might see something like this:

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

interface UserWithoutId {
    name: string;
    email: string;
    age: number;
}
```

This is not ideal. We're duplicating information: if we ever add, remove, or change a field, we now need to remember to do it in multiple places. Instead, we generally want to follow a **single source of truth** approach.

In this simple example, we can refactor by defining the common shape once and then building on top of it:

```ts
interface UserWithoutId {
    name: string;
    email: string;
    age: number;
}

interface User extends UserWithoutId {
    id: string;
}
```

Now `UserWithoutId` is our source of truth for the shared fields, and `User` simply extends it with an `id`. If we later decide to add a `username` field or remove `age`, we only need to update `UserWithoutId`. The `User` type automatically stays in sync.

The rest of this chapter will introduce utility types and type transformations that let you apply this idea more systematically. The goal is to define core types once, then derive the variants you need — so when requirements change, you can update a single definition and have the rest of your type system follow along.

Partial Utility Type

TypeScript includes several built-in *utility types* that transform existing types into new ones. One of the most common and useful of these is `Partial<T>`, which makes **all properties of a type optional**.

Consider this `User` type:

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

Imagine we want a function that can update any subset of a user's fields. Without `Partial<T>`, we might write something like this:

```ts
// Without Partial
function updateUser(
    userId: string,
    userInfo: {
        id?: string;
```

```
        name?: string;
        email?: string;
    },
) {
    // ...
}
```

This works, but it duplicates the `User` type structure and will quickly fall out of sync if `User` changes.

With `Partial<T>`, we can instead derive the type from `User` directly:

```
// With Partial
function updateUser(userId: string, userInfo: Partial<User>) {
    // ...
}
```

Now we have a *single source of truth*: `Partial<User>` is automatically updated whenever `User` changes. If we add a `username` field to `User`, `updateUser` will immediately accept `username` as an optional property too.

In other words, `Partial<T>` lets us generate a new type from an existing one, instead of copy/pasting and maintaining multiple versions by hand.

Nested Objects

It's important to understand that `Partial<T>` only makes the **top-level properties** of `T` optional; it does **not** recursively make nested properties optional.

For example:

```
type User = {
    id: string;
    name: string;
    preferences: {
        theme: string;
        notifications: boolean;
    };
};
```

If we use `Partial<User>`, the resulting type looks like this:

```
// same as 'type LooseyGooseyUser = Partial<User>'
type LooseyGooseyUser = {
    id?: string;
    name?: string;
    preferences?: {
        theme: string;
        notifications: boolean;
    };
};
```

Here, `id`, `name`, and `preferences` are optional. But **inside** `preferences`, the `theme` and `notifications` properties are still required whenever `preferences` itself is present.

Required Utility Type

Where `Partial<T>` makes all properties of a type optional, the `Required<T>` utility type does the **opposite**: it forces all properties of a type to be required, even ones that were originally optional.

Using `Required<T>`

Here's a practical example:

```
interface BlogPost {
    title: string;
    content: string;
    tags?: string[];
    publishDate?: Date;
    author?: {
        id: string;
        name?: string;
    };
}

// All top-level properties are now required
type MyRequiredBlogPost = Required<BlogPost>;
```

The `MyRequiredBlogPost` type is equivalent to:

```
type MyRequiredBlogPost = {
    title: string;
    content: string;
    tags: string[];
    publishDate: Date;
    author: {
        id: string;
        name?: string;
    };
};
```

Notice that:

- The **top-level** properties `tags`, `publishDate`, and `author` are now required.
- The `name` property inside `author` is still optional, because `Required<T>` only affects the properties directly on `T` itself.

Just like `Partial<T>`, `Required<T>` is **not recursive**. It does a single level of transformation, leaving any nested objects unchanged. Later sections will explore more advanced patterns when you need deeper control over nested structures.

Readonly Utility Type

`Readonly<T>` creates a new type where all top-level properties of `T` are marked as `readonly`, preventing them from being reassigned after initialization.

```
interface UserProfile {
    id: string;
    name: string;
    preferences: {
        readonly theme: "light" | "dark";
        notifications: boolean;
    };
}

type ConstantUserProfile = Readonly<UserProfile>;

// This is the same as:
```

```
// type ConstantUserProfile = {
//   readonly id: string;
//   readonly name: string;
//   readonly preferences: {
//     readonly theme: "light" | "dark";
//     notifications: boolean;
//   };
// };
```

`Readonly` is **shallow**: it makes the top-level properties of `UserProfile` readonly, but it does not automatically make every nested property readonly. In this example, the `preferences` object itself is readonly (you can't reassign `preferences`), but inside it only `theme` is readonly because it was declared that way in `UserProfile`.

Use `Readonly<T>` when you want to:

- expose immutable views of otherwise mutable objects
- prevent accidental reassignment of configuration or settings objects
- make API return values read-only to consumers

Record Utility Type

`Record<K, T>` creates an object type with keys of type `K` and values of type `T`.

It is especially useful when you:

- want a dictionary-like object with consistent value types
- need to ensure that **all** members of a union of keys are present
- want TypeScript to prevent extra keys from being added

Basic dictionary usage

```
// Using string as the key type
type StringKeyDictionary = Record<string, number>;

const karateScores: StringKeyDictionary = {
    "Ralph Macchio": 60,
    "William Zabka": 100,
    "Jackie Chan": 82,
};

// We can add any string key
karateScores["Pat Morita"] = 85;

// But values must be numbers
// Type 'string' is not assignable to type 'number'
karateScores["Eve"] = "A+";
```

Using unions for required keys

One of the most powerful uses of `Record` is with a union of literal types. This ensures that **every key in the union exists** in the object, and that no extra keys are allowed.

```
// Using a union of literal types as keys
type PlayerRole = "tank" | "healer" | "dps";
type RoleCapacity = Record<PlayerRole, number>;

const partyRequirements: RoleCapacity = {
```

```
    tank: 1,
    healer: 2,
    dps: 3,
};

// TypeScript error if any role is missing
const invalidRequirements: RoleCapacity = {
    tank: 1,
    dps: 3,
    // Property 'healer' is missing in type '{ tank: number; dps: number; }'.
};

// We can't add additional keys not in the union
// Property 'support' does not exist on type 'RoleCapacity'.
partyRequirements["support"] = 1;
```

Lookup tables and configuration

`Record` shines for exhaustive lookup tables and configuration objects where you want every possible key to be handled.

```
type HttpStatusCode = 200 | 201 | 400 | 401 | 403 | 404 | 500;

const statusMessages: Record<HttpStatusCode, string> = {
    200: "OK",
    201: "Created",
    400: "Bad Request",
    401: "Unauthorized",
    403: "Forbidden",
    404: "Not Found",
    500: "Internal Server Error",
};

function getStatusMessage(code: HttpStatusCode): string {
    return statusMessages[code];
}

getStatusMessage(404); // "Not Found"
```

By combining `Record` with unions of keys, you get strongly typed, exhaustive maps that are easy to refactor and hard to misuse.

Pick Utility Type

`Pick<T, K>` creates a new type by selecting a subset of properties from an existing type `T`.

```
interface Product {
    id: string;
    name: string;
    price: number;
    description: string;
    category: string;
    inStock: boolean;
    images: string[];
    reviews: { user: string; rating: number; text: string }[];
}
```

```ts
type ProductSummary = Pick<Product, "id" | "name" | "price">;

const productList: ProductSummary[] = [
    { id: "p1", name: "Headphones", price: 79.99 },
    { id: "p2", name: "Mouse", price: 49.99 },
];

const invalidProduct: ProductSummary = {
    id: "p3",
    name: "Keyboard",
    price: 99.99,
    // TypeScript error:
    // Object literal may only specify known properties,
    // and 'description' does not exist in type 'ProductSummary'.
    description: "Noise cancelling headphones",
};
```

Because `ProductSummary` is built with `Pick`, it only allows the `id`, `name`, and `price` properties from `Product`. Any extra properties (like `description`) cause a type error.

`Pick` is handy for creating focused "views" of a larger type, such as:

- return types from APIs that don't need to expose every field
- DTOs or lightweight objects passed between layers
- narrowing a type for UI components that only read a few properties

Omit Utility Type

`Omit<T, K>` is the opposite of `Pick<T, K>`. Instead of choosing a set of properties to **keep**, it creates a new type by **excluding** a set of properties `K` from an existing type `T`.

This is especially handy when you want to hide sensitive fields (like passwords) or remove implementation details before sending data over an API or into another layer of your application.

```ts
interface DatabaseUser {
    id: string;
    username: string;
    email: string;
    passwordHash: string;
    createdAt: Date;
    updatedAt: Date;
}

// Create a safe user representation without sensitive or internal data
type PublicUser = Omit<DatabaseUser, "passwordHash" | "updatedAt">;

function getUserProfile(userId: string): PublicUser {
    // Imagine this was fetched from a database
    const dbUser: DatabaseUser = {
        id: userId,
        username: "predragmilanovic",
        email: "predrag@example.com",
        passwordHash: "${2a$12$...}",
        createdAt: new Date("2026-01-28"),
        updatedAt: new Date(),
    };
```

```
    // Convert to PublicUser (only allowed properties from PublicUser)
    const publicUser: PublicUser = {
        id: dbUser.id,
        username: dbUser.username,
        email: dbUser.email,
        createdAt: dbUser.createdAt,
    };

    return publicUser;
}

// If you accidentally try to include an omitted property,
// TypeScript will flag it as an error:
const invalidPublicUser: PublicUser = {
    id: "123",
    username: "predragmilanovic",
    email: "predrag@example.com",
    createdAt: new Date("2026-01-28"),
    // TypeScript error:
    // Object literal may only specify known properties,
    // and 'passwordHash' does not exist in type 'PublicUser'.
    passwordHash: "${2a$12$...}",
};
```

Because `PublicUser` is built with `Omit`, it **cannot** have the `passwordHash` or `updatedAt` properties from `DatabaseUser`. Trying to include them produces a type error, which helps prevent accidentally leaking sensitive information.

Use `Omit` when you want to:

- expose a "safe" version of a type without internal or sensitive fields
- remove fields not needed in a specific context (e.g., UI view models)
- derive input or response DTOs from a richer domain model while sharing the same base shape

Utility Types

As a TypeScript codebase grows, it's easy to end up with many slightly different versions of the same shape – "user without id", "user for updates", "user response", and so on. Instead of copy-pasting these variants, TypeScript's **utility types** let you **derive** them from a single source of truth.

This chapter walks through the most commonly used built-in utilities and shows how they help you keep your types DRY, expressive, and safe.

- **Single Source of Truth** – introduces the idea of defining core types once and building everything else from them.
- `Partial<T>` – makes all properties optional, perfect for "update" or "patch" shapes derived from an existing type.
- `Required<T>` – does the opposite of `Partial`, forcing all top-level properties to be present.
- `Readonly<T>` – turns all top-level properties into `readonly`, great for immutable views and configuration.
- `Record<K, T>` – builds strongly-typed dictionaries and lookup tables from a key type `K` and value type `T`.
- `Pick<T, K>` – creates a focused view of a type by selecting a subset of its properties.
- `Omit<T, K>` – the inverse of `Pick`, producing a type by excluding specific properties (for example, to strip sensitive fields before returning API responses).

Together, these utilities form a toolkit for transforming existing types into the exact shapes you need, without sacrificing type safety or maintainability.

Generics

Generics are one of TypeScript's most powerful features. They let you write reusable logic that works with **many types** instead of just one, while still preserving precise type information.

For example, a data structure like a queue or stack shouldn't care *what* kind of values it stores:

- `NumberQueue`
- `StringQueue`
- `UserQueue`
- etc.

Rather than re-implementing each of these, we can write a single `Queue<T>` that works for any type `T`. When we use it as `Queue<number>` or `Queue<User>`, TypeScript keeps track of that concrete type everywhere we interact with the queue.

Generics are how we reuse behavior **across types** without falling back to `any`.

You've already seen many generics in the standard library and earlier chapters:

- `Array<T>` – arrays of `T` (`Array<number>`, `Array<string>`, etc.)
- `Promise<T>` – an async computation that eventually yields a `T`
- Utility types like `Partial<T>`, `Required<T>`, `Readonly<T>`, `Record<K, T>`, `Pick<T, K>`, and `Omit<T, K>`

In all of these, the `<T>` (or `<K, T>`) part is called a **generic type parameter**. `T` is just a variable name for "some type" – we could call it anything – but `T` is a very common convention.

Creating Custom Generic Functions

Let's build a simple but realistic example. Imagine some client-side code that makes lots of `fetch` requests to a backend. We might start with a helper like this:

```
async function fetchFromApi(url: string) {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("Network response was not ok");
        }
        return await response.json();
    } catch (error) {
        console.error("Error fetching data:", error);
        return undefined;
    }
}
```

This works, but TypeScript infers the return type as `Promise<any>` (or `Promise<unknown>`), which doesn't give us much help when we use it.

We can improve this by making the function **generic** over the type of data we expect back:

```
async function fetchFromApi<T>(url: string): Promise<T | undefined> {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("Network response was not ok");
```

```
        }
        return (await response.json()) as T;
    } catch (error) {
        console.error("Error fetching data:", error);
        return undefined;
    }
}
```

Here:

- `<T>` declares a **type parameter** `T` for the function.
- The return type `Promise<T | undefined>` says "this returns a `T` (or `undefined` if something goes wrong)."

Now, whenever we call `fetchFromApi`, we specify the type we expect to receive:

```
type Comment = { id: string; body: string };
type User = { id: string; name: string };
type Post = { id: string; title: string };

const comments = await fetchFromApi<Comment[]>(
    "https://api.example.com/posts/1/comments",
);

const user = await fetchFromApi<User>(
    "https://api.example.com/user/1",
);

const posts = await fetchFromApi<Post[]>(
    "https://api.example.com/posts",
);
```

In each call, `T` is inferred from the explicit type argument:

- `fetchFromApi<Comment[]>` → `T` is `Comment[]`
- `fetchFromApi<User>` → `T` is `User`
- `fetchFromApi<Post[]>` → `T` is `Post[]`

Everywhere we use `comments`, `user`, or `posts`, TypeScript now knows their exact shapes. That's the core value of generics: **reusable logic plus precise types**, without sacrificing safety or falling back to `any`.

Multiple Type Parameters

Type parameters are just parameters. You don't have to stop at a single one — a generic function or type can use as many type parameters as it needs (within reason).

In examples, you'll often see short names like `T`, `U`, or `V`. Those are only conventions: you can use longer, more descriptive names if you prefer.

A Transform Function with Two Type Parameters

Let's build a function that "transforms" its inputs. It should:

- Take an array of items of type `InputType`.
- Take a function that converts an `InputType` into an `OutputType`.
- Return a new array of `OutputType` values.

```
function transform<InputType, OutputType>(
    inputs: InputType[],
```

```
    update: (item: InputType) => OutputType,
): OutputType[] {
    const outputs: OutputType[] = [];

    for (const input of inputs) {
        const output = update(input);
        outputs.push(output);
    }

    return outputs;
}
```

That signature is a bit long — which is why many people prefer single-letter names like `T` and `U` for type parameters in real code.

If you've used JavaScript's `Array.prototype.map`, you might recognize this pattern: `transform` is essentially a typed reimplementation of `map`.

Using `transform` with Objects

First, define a `Human` type and an array of humans:

```
type Human = {
    name: string;
    age: number;
};

const humans: Human[] = [
    { name: "Eren", age: 15 },
    { name: "Mikasa", age: 16 },
    { name: "Armin", age: 15 },
];
```

Now create a transformer that turns a `Human` into a `string`:

```
const titanTransformer = (human: Human): string =>
    `${human.name} is a titan!`;

const titanNames = transform<Human, string>(humans, titanTransformer);
console.log(titanNames);
// ['Eren is a titan!', 'Mikasa is a titan!', 'Armin is a titan!']
```

Here, TypeScript checks that: - The `inputs` array is `Human[]`. - The `update` function takes a `Human` and returns a `string`. - The return type of `transform` is `string[]`.

Reusing `transform` with Other Types

We don't need to change `transform` to work with completely different data — we only change the type arguments and the `update` function.

```
const numbers = [1, 2, 3, 4, 5];

const double = (num: number): number => num * 2;

const doubledNumbers = transform<number, number>(numbers, double);
console.log(doubledNumbers);
// [2, 4, 6, 8, 10]
```

By using multiple type parameters, `transform` stays flexible and type-safe no matter what kinds of values you pass in and return.

Generic Constraints

Sometimes you need a generic function to know **something** about the types it operates on. In the examples so far, the type parameter could be anything:

```
async function fetchFromApi<T>(url: string): Promise<T | undefined>;
```

Here `T` is completely unconstrained. TypeScript doesn't assume any properties or methods on `T`.

Generic **constraints** let you say, "`T` must at least have these members." That way, your implementation can safely use those members while still being generic over everything else.

Constraining with `extends`

Constraints are usually expressed with interfaces (or object types) and the `extends` keyword on a type parameter.

First, define a constraint type:

```
interface HasCost {
    cost: number;
}
```

Then constrain a generic type parameter to types that satisfy that interface:

```
function applyDiscount<T extends HasCost>(
    vals: T[],
    discount: number,
): T[] {
    const arr: T[] = [];

    for (const val of vals) {
        val.cost *= discount;
        arr.push(val);
    }

    return arr;
}
```

`T extends HasCost` means: - `T` can be any type **as long as** it has a numeric `cost` property. - Inside the function, `val.cost` is safe to access and mutate. - The function still returns `T[]`, so you don't lose type information.

Using the Constrained Function

Because `HasCost` is just "anything with a `cost: number`," lots of different shapes can work.

```
const shoes = [
    {
        size: 12.5,
        country: "BIH",
        cost: 120,
    },
    {
        size: 12.5,
```

```
        country: "US",
        cost: 110,
    },
];

const tvs = [
    {
        framerate: 120,
        brand: "Samsung",
        cost: 500,
    },
    {
        framerate: 240,
        brand: "Vizio",
        cost: 300,
    },
];

const people = [
    {
        name: "Predrag",
    },
    {
        name: "Breanna",
    },
];

const discountedShoes = applyDiscount(shoes, 0.3);
const discountedTVs = applyDiscount(tvs, 0.5);

// Error:
// Argument of type '{ name: string; }[]' is not assignable to
// parameter of type 'HasCost[]'.
// (Also, you can't buy people!)
const discountedPeople = applyDiscount(people, 0.2);
```

`shoes` and `tvs` are fine because every element has a `cost` property, so they satisfy `HasCost`. `people` fails the constraint because there is no `cost` property.

Generic constraints are a key tool for writing reusable functions that still make **safe assumptions** about the shapes of the values they work with.

Type Parameters for Types

Type parameters aren't just for functions and methods. You can also use them to create **generic object types** with interfaces or type aliases.

A Generic Store Type

Here's a simple `Store` interface that is generic over the kind of value it stores:

```
interface Store<T> {
    get(id: string): T;
    save(id: string, item: T): void;
    list(): T[];
```

```
}

// The same idea works with a type alias:
// type Store<T> = {
//   get(id: string): T;
//   save(id: string, item: T): void;
//   list(): T[];
// };
```

`Store<T>` describes any value that has `get`, `save`, and `list` methods. The concrete type of `T` is left open so different callers can plug in whatever they need.

Using a Generic Store in a Function

We can now write a function that works with **any** `Store<T>` regardless of what `T` is:

```
function addAndGetItems<T>(
    store: Store<T>,
    id: string,
    newItem: T,
): T[] {
    store.save(id, newItem);
    return store.list();
}
```

`addAndGetItems` doesn't care what is stored; it just uses the operations guaranteed by the `Store<T>` interface. TypeScript will infer `T` from the `store` and `newItem` arguments.

A Store for Products

First, define a `Product` type, and then create an object whose shape matches `Store<Product>`:

```
type Product = {
    name: string;
    price: number;
};

const productStore = {
    products: {} as Record<string, Product>,

    get(id: string): Product {
        return this.products[id];
    },

    save(id: string, item: Product): void {
        this.products[id] = item;
    },

    list(): Product[] {
        return Object.values(this.products);
    },
};
```

Because `productStore` has `get`, `save`, and `list` methods with the right types, it is compatible with `Store<Product>`.

We can now call `addAndGetItems` and TypeScript will infer `T` as `Product`:

```
const newStore = addAndGetItems(productStore, "laneslaptop", {
    name: "Laptop",
    price: 999,
});
console.log(newStore);
// [{ name: "Laptop", price: 999 }]

const finalStore = addAndGetItems(productStore, "allanstoaster", {
    name: "Toaster",
    price: 50,
});
console.log(finalStore);
// [
//   { name: "Laptop", price: 999 },
//   { name: "Toaster", price: 50 },
// ]
```

A Store for Something Else

The real power of generic types is reuse. We can define a completely different type and create another store that still works with `addAndGetItems`:

```
type Homunculus = {
    title: string;
    abilities: string[];
};

const homunculusStore = {
    homunculi: {} as Record<string, Homunculus>,

    get(id: string): Homunculus {
        return this.homunculi[id];
    },

    save(id: string, item: Homunculus): void {
        this.homunculi[id] = item;
    },

    list(): Homunculus[] {
        return Object.values(this.homunculi);
    },
};

const newHomunculus = addAndGetItems(homunculusStore, "laneslaptop", {
    title: "Laptop",
    abilities: ["fast", "strong"],
});
console.log(newHomunculus);
// [{ title: "Laptop", abilities: ["fast", "strong"] }]
```

Here, `addAndGetItems` is reused with `Homunculus` values simply because `homunculusStore` has the same `Store<T>` shape, but with `T` being `Homunculus` instead of `Product`.

This is the core idea of **type parameters for types**: you define one generic structure and let callers plug in the concrete type they need.

Generic Type Inference

When you call a generic function, you can usually **skip** writing the type arguments yourself. TypeScript can infer them from the values you pass in.

Let's revisit our "titan transformer" example:

```
function transform<InputType, OutputType>(
    inputs: InputType[],
    update: (item: InputType) => OutputType,
): OutputType[] {
    const outputs: OutputType[] = [];

    for (const input of inputs) {
        const output = update(input);
        outputs.push(output);
    }

    return outputs;
}

type Human = {
    name: string;
    age: number;
};

const humans: Human[] = [
    { name: "Eren", age: 15 },
    { name: "Mikasa", age: 16 },
    { name: "Armin", age: 15 },
];

const titanTransformer = (human: Human): string => `${human.name} is a titan!`;
```

Explicit Type Arguments

We *can* call `transform` by explicitly specifying the type parameters:

```
const titanNames = transform<Human, string>(humans, titanTransformer);
console.log(titanNames);
//    ^? string[]
```

Here we are telling TypeScript: `InputType` is `Human` and `OutputType` is `string`.

Letting TypeScript Infer the Types

But we don't have to. In most real code you just write:

```
const titanNames = transform(humans, titanTransformer);
//    ^? string[]
```

TypeScript looks at the arguments you passed in and figures out the type parameters:

- `humans` is `Human[]`, so `InputType` is `Human`.
- `titanTransformer` takes a `Human` and returns a `string`, so `OutputType` is `string`.

Once the type parameters are inferred, the rest of the function body and the return type are all fully typed as if you had written `<Human, string>` yourself.

In practice, **generic type inference** means you usually:

- Declare the type parameters on the function.
- Let TypeScript infer the actual type arguments at call sites.

You only need to write the type arguments manually when inference can't figure them out or when you want to override what it would infer.

Generic Classes

By now you've seen that you can bolt type parameters onto almost anything in TypeScript: functions, type aliases, interfaces, and more.

Classes are no exception – they can be generic too.

In this section we'll combine a few ideas:

- `InMemoryRepository` is a **generic class**.
- It implements a **generic interface** (`Repository<T>`).
- The type parameter `T` is **constrained** to have an `id` property.

A Generic Repository Interface

First, a plain generic interface that describes the behavior we want from a repository:

```
interface Repository<T> {
    getAll(): T[];
    getById(id: string): T | undefined;
    save(item: T): void;
}
```

- `Repository<T>` doesn't care what `T` is.
- It just says: "Give me some type `T`, and I'll describe how to get and save values of that type."

A Generic Class With a Constraint

Now let's implement this interface with a class that stores everything in memory:

```
class InMemoryRepository<T extends { id: string }>
    implements Repository<T>
{
    private items: T[] = [];

    getAll(): T[] {
        return [...this.items];
    }

    getById(id: string): T | undefined {
        return this.items.find((item) => item.id === id);
    }

    save(item: T): void {
        const index = this.items.findIndex((i) => i.id === item.id);

        if (index >= 0) {
            this.items[index] = item;
        } else {
            this.items.push(item);
```

```
        }
    }
}
```

There are a few things to notice here:

- `InMemoryRepository` is a **generic class**: it has a type parameter `<T>`.
- `InMemoryRepository<T>` **implements** `Repository<T>`, so TypeScript makes sure all the methods and their types line up correctly. If we forget a method or change a return type, we'll get a type error.
- The type parameter has a **constraint**: `T extends { id: string }`.
  - Any `Repository<T>` is free to choose whatever `T` is.
  - But this particular implementation needs an `id` so it can store and look things up.

In other words: *any* object type can go into `InMemoryRepository`, as long as it has an `id` property of type `string`. All the implementation logic is shared for all those different possible types.

Using the Generic Class

Let's use `InMemoryRepository` with a specific type:

```
interface Shinigami {
    id: string;
    name: string;
}

const deathNoteRepo = new InMemoryRepository<Shinigami>();

deathNoteRepo.save({ id: "1", name: "Ryuk" });
deathNoteRepo.save({ id: "2", name: "Rem" });

console.log(deathNoteRepo.getAll());
//    ^? Shinigami[]
```

TypeScript keeps track of the type argument you pass in:

- `deathNoteRepo` is inferred as `InMemoryRepository<Shinigami>`.
- `getAll()` returns `Shinigami[]`.
- `getById("1")` returns `Shinigami | undefined`.
- `save` only accepts values of type `Shinigami`.

Violating the Constraint

If you try to create an `InMemoryRepository` for a type that **doesn't** have an `id` property, TypeScript will complain:

```
interface CharacterWithoutId {
    name: string;
}

// Error:
// Type 'CharacterWithoutId' does not satisfy the constraint '{ id: string; }'.
//   Property 'id' is missing in type 'CharacterWithoutId' but required in type '{ id: string; }'.
const characterRepo = new InMemoryRepository<CharacterWithoutId>();
```

This is exactly what we want: the constraint on `T` means you **can't** accidentally create an `InMemoryRepository` for a type that doesn't have the data it needs to function.

Why Use `implements` With Generics?

In this example, the `implements` keyword is doing important work:

- It guarantees that `InMemoryRepository<T>` can be used anywhere a `Repository<T>` is expected.
- It keeps the implementation and the interface in sync, even as you refactor.
- It scales nicely: you can have multiple implementations (e.g. `SqlRepository<T>`, `ApiRepository<T>`) all implementing the same `Repository<T>` interface with different internals.

Generic classes like this are a great fit whenever you have **shared behavior** that should work the same way for many different types, but still be **fully type-safe**.

Generics

Generics are how you write **reusable, type-safe code** in TypeScript.

Instead of hard-coding a single type into a function, class, or object type, you add one or more **type parameters** (like `T` or `InputType`) and let callers plug in the concrete types they need. TypeScript then tracks those types all the way through your code.

You've already used generics in the standard library:

- `Array<T>` – arrays of `T` (e.g. `Array<number>`)
- `Promise<T>` – async values that eventually yield a `T`
- Utility types like `Partial<T>`, `Readonly<T>`, `Record<K, T>`, `Pick<T, K>`, `Omit<T, K>`

This chapter walks through how to **create and use your own generics**.

What You'll Learn

- **01 – Generics**
  Introduces generic functions and type parameters using a `fetchFromApi<T>` helper. You'll see how `<T>` lets you keep precise result types (`Comment[]`, `User`, `Post[]`, etc.) without copying logic or falling back to `any`.
- **02 – Multiple Type Parameters**
  Shows how to use more than one type parameter with a `transform<InputType, OutputType>` function (a typed `map`). You'll reuse the same function for different input/output combos (e.g. `Human → string`, `number → number`).
- **03 – Generic Constraints**
  Adds constraints like `T extends HasCost` so your generic functions can safely assume certain properties exist. You'll see how this lets you write helpers (like `applyDiscount`) that work for any type "with a `cost`", while rejecting incompatible shapes.
- **04 – Type Parameters for Types**
  Moves beyond functions to **generic object types**. You'll build a `Store<T>` interface and reuse it for different domains (`Product`, `Homunculus`, …) while keeping implementations and helpers like `addAndGetItems` fully type-safe.
- **05 – Generic Type Inference**
  Shows that you rarely need to write type arguments at call sites. Using the `transform` example, you'll see how TypeScript infers `InputType` and `OutputType` from the arguments you pass, and when you might still specify them explicitly.
- **06 – Generic Classes**
  Applies the same ideas to classes. You'll build an `InMemoryRepository<T>` that implements a generic `Repository<T>` interface, with a constraint `T extends { id: string }`. This demonstrates reusable, type-safe behavior across many object types.

Taken together, these sections show how generics let you:

- Extract common patterns into **reusable building blocks**.
- Preserve **rich, specific types** instead of falling back to `any`.
- Scale your codebase without duplicating logic for every new type that comes along.

Conditional Types

We're now getting into some pretty advanced stuff that, while very useful in tricky modelling situations, is not something you'll usually need in everyday application code.

As a general rule, advanced TypeScript features are more useful in **library code** that needs to be flexible, abstract, and reusable. Application-level TypeScript code is generally much simpler and more concrete…

Conditional types let us create **new types based on conditions** inside the type system itself.

Basic Syntax

The general form of a conditional type looks like this:

```
type NewType = SomeType extends OtherType ? TrueType : FalseType;
```

You can read this just like a ternary expression in JavaScript:

> "If `SomeType` extends (satisfies) `OtherType`, then `NewType` is `TrueType`; otherwise, it's `FalseType`."

Simple Example: `IsString`

Here's a very small but clear example:

```
type IsString<T> = T extends string ? true : false;

// Usage
type Result1 = IsString<"hello">; // true
type Result2 = IsString<42>;      // false
type Result3 = IsString<string>;  // true
```

In this example, `IsString` is a conditional type that checks whether the type parameter `T` extends `string`. If it does, the resulting type is `true`; otherwise, it's `false`.

This might not look immediately useful on its own, but this same pattern powers a lot of the more advanced utility types in the standard library.

Built-in Conditional Utility Types

TypeScript actually ships with some built-in conditional types. Here are three important ones (simplified):

```
type Extract<T, U> = T extends U ? T : never;
type Exclude<T, U> = T extends U ? never : T;
type NonNullable<T> = T extends null | undefined ? never : T;
```

Roughly speaking:

- `Extract<T, U>` keeps only the parts of `T` that are assignable to `U`.
- `Exclude<T, U>` removes from `T` anything that is assignable to `U`.
- `NonNullable<T>` removes `null` and `undefined` from `T`.

We'll focus on `Extract` for a practical example.

Practical Example: Filtering Event Unions

As usual, the question is: **"When the heck is this useful?"**

Imagine we have some events that can fire in a front-end application:

```
type ClickEvent = { type: "click"; x: number; y: number };
type KeyEvent = { type: "key"; key: string };
type MouseMoveEvent = { type: "mousemove"; x: number; y: number };
type FormEvent = { type: "submit"; formId: string };

type Event = ClickEvent | KeyEvent | MouseMoveEvent | FormEvent;
```

It might be useful to **dynamically create a type** that only includes the "mouse-related" events – the ones that have both `x` and `y` properties. We can use the `Extract` conditional type to do that.

First, here's the (simplified) implementation of `Extract` again, just for reference:

```
type Extract<T, U> = T extends U ? T : never;
```

Now we can use it to filter our `Event` union:

```
type MouseRelatedEvents = Extract<Event, { x: number; y: number }>;
```

`MouseRelatedEvents` is now the same as:

```
type MouseRelatedEvents = ClickEvent | MouseMoveEvent;
```

The key difference is that it's **dynamic**. If we add more events to the `Event` union in the future, `MouseRelatedEvents` will automatically include them as long as they match the condition (i.e., they have `x` and `y` properties).

This is where conditional types really shine: they let you express relationships *between* types so that when one type changes, the others automatically stay in sync.

Infer

The `infer` keyword is a special feature you can only use **inside a conditional type**. It lets you *capture* a type from the `true` branch and give it a name so you can reuse it.

The classic example is extracting the **return type** of a function:

```
type GetReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
```

You can read this as:

> "If `T` is a function type that takes any arguments and returns some type `R`, then `GetReturnType<T>` is `R`; otherwise it's `never`."

Using `GetReturnType`

```
function greet() {
    return "Hello, world!";
}

function sum(a: number, b: number) {
    return a + b;
}

type GreetReturnType = GetReturnType<typeof greet>; // string
type SumReturnType = GetReturnType<typeof sum>;     // number
```

In real-world code you'd normally use the built-in `ReturnType<T>` utility, but `GetReturnType` is a nice, small example of how `infer` works.

Why `infer R` instead of just `R`?

You might be thinking: *"Why do we write `infer R` instead of just `R`?"* The short answer is: **because TypeScript's syntax requires it.**

The conditional type is checking whether `T` matches this function shape:

```
T extends (...args: any[]) => any
```

But we don't want the return type to be `any` – we want to **capture** whatever that return type actually is and reuse it in the `true` branch of the conditional.

So instead of writing `any`, we write:

```
T extends (...args: any[]) => infer R
```

Here's what `infer R` is doing:

- It **introduces a new type variable** `R` right inside the conditional type.
- If `T` really is a function type, TypeScript **infers** what `R` should be from its return type.
- That inferred `R` is then available only in the `true` branch (`? R : never`).

So you can think of `infer R` as saying:

> "If this conditional is true, create a new type variable `R` for the part of the type I've marked, and let me use `R` on the right-hand side of the `?`."

This same pattern works in more complex situations too – for example, inferring element types from arrays, parameter types from functions, and so on – but the basic idea is always the same: **use `infer` to name a piece of a matched type so you can reuse it.**

Mapped Types

Remember **dynamic properties**?

```
type UserMetrics = {
    [key: string]: number;
};
```

This kind of type is called an **index signature** – it says:

> "For *any* string key, the value will be a `number`."

Mapped types are a more precise and powerful version of this idea.

Instead of allowing *any* key, a mapped type lets us create **new types with dynamic properties based on the keys of an existing type**.

From `Soldier` to `OptionalSoldier`

Say we have a `Soldier` type:

```
type Soldier = {
    name: string;
    age: number;
    branch: "garrison" | "military police" | "survey corps";
};
``;
```

Now imagine we want a new type with **the same properties**, but where **all of them are optional**. We

Instead, we can use a **mapped type**:

```ts
type OptionalSoldier = {
    [K in keyof Soldier]?: Soldier[K];
};
```

Let's break that down:

- `keyof Soldier` gets the **union of keys** of `Soldier` → `"name" | "age" | "branch"`.
- `K in ...` iterates over each of those keys.
- The `?` after the property name makes each property **optional**.
- `Soldier[K]` looks up the **value type** for each key in `Soldier`.

The result is exactly the same as writing:

```ts
type OptionalSoldier = {
    name?: string;
    age?: number;
    branch?: "garrison" | "military police" | "survey corps";
};
```

The big win is that if we **add, remove, or change** properties on `Soldier`, the `OptionalSoldier` type **automatically stays in sync**.

Changing the Value Types

Mapped types are often used to make properties `optional` or `readonly`, but they can also **change the value type entirely**.

For example, we can create a version of `Soldier` where every property is a `string`:

```ts
type StringifiedSoldier = {
    [K in keyof Soldier]: string;
};
```

Which is equivalent to:

```ts
type StringifiedSoldier = {
    name: string;
    age: string;
    branch: string;
};
```

Again, this stays **automatically up to date** if `Soldier` changes, because it's defined in terms of `keyof Soldier` and its keys.

Generic Mapped Types

In real-world code, we usually turn patterns like this into **reusable generic helpers**:

```ts
// Make all properties of T optional
type Optional<T> = {
    [K in keyof T]?: T[K];
};

// Turn all property values into strings
type Stringified<T> = {
```

```
    [K in keyof T]: string;
};

type OptionalSoldier2 = Optional<Soldier>;
type StringifiedSoldier2 = Stringified<Soldier>;
```

Many of TypeScript's built-in utility types (like `Partial<T>` and `Readonly<T>`) are built using this same **mapped type** pattern.

Mapped Types With Conditionals

If mapped types are a step toward type-level wizardry, **conditional mapped types** are one step further.

They're powerful and genuinely useful in some situations – but they can also become hard to read if you're not careful. As always with advanced type features: use them when they clearly improve correctness or maintainability, not just because they're clever.

Starting Point: `OptionalSoldier`

From the previous section, recall our `Soldier` and `OptionalSoldier` types:

```
type Soldier = {
    name: string;
    age: number;
    branch: "garrison" | "military police" | "survey corps";
};

type OptionalSoldier = {
    [K in keyof Soldier]?: Soldier[K];
};
```

`OptionalSoldier` is a **mapped type** that:

- Iterates over each key in `Soldier` with `K in keyof Soldier`.
- Uses `Soldier[K]` to get the value type for that key.
- Marks each property as optional with `?`.

Filtering Properties by Value Type

What if, instead of making properties optional, we wanted to **filter out any non-string properties**?

In other words, we want a type that:

- Keeps only the properties whose value type is assignable to `string`.
- Drops everything else.

That's exactly what **conditional mapped types** are for:

```
type FilteredSoldier = {
    [K in keyof Soldier]: Soldier[K] extends string ? Soldier[K] : never;
};
```

Here's what's happening inside the mapped type:

- For each key `K` in `keyof Soldier`:
  - We check the value type `Soldier[K]`.
  - If `Soldier[K] extends string`, we keep it as `Soldier[K]`.
  - Otherwise, we use `never`.

Because `never` is treated as "no possible value" in many mapped-type contexts, properties with `never` often end up effectively **filtered out** in downstream usage (for example, when indexing by the value type or combining with other mapped types).

The resulting type is:

```
type FilteredSoldier = {
    name: string;
    // age: never;
    branch: "garrison" | "military police" | "survey corps";
};
```

Two important details:

- `age` becomes `never` because `number` is not assignable to `string`.
- The `branch` property **keeps its specific union type**, not just `string`, because we checked `Soldier[K] extends string` and then *reused that same `Soldier[K]`* in the `true` branch of the conditional.

This pattern – using a conditional inside a mapped type and then reusing the more specific type – is very common in advanced utility types in the TypeScript standard library.

Extracting keys from types

Mapped types don't just let you build new object types – you can also use them together with conditional types to *extract* keys that match some condition.

Start with a simple object type:

```
type Soldier = {
    name: string;
    age: number;
    branch: "garrison" | "military police" | "survey corps";
};
```

Imagine you want a union of just the keys whose values are `string`. One way to do this is to first build a helper mapped type that keeps the key name when the value type matches, and otherwise produces `never`:

```
type StringKeys<T> = {
    [K in keyof T]: T[K] extends string ? K : never;
};

type Result = StringKeys<Soldier>;
/*
type Result = {
    name: "name";
    age: never;
    branch: "branch";
}
*/
```

Now we can turn that object type into a union of its values by indexing it with `keyof T`:

```
type StringKeyUnion<T> = StringKeys<T>[keyof T];

type Keys = StringKeyUnion<Soldier>;
//     ^? "name" | "branch"
```

This pattern – *"map, then index"* – is a common way to use conditional and mapped types together:

- The mapped type `StringKeys<T>` walks over every property key `K` in `T` and conditionally keeps it or replaces it with `never`.

- Indexing with `keyof T` (`StringKeys<T>[keyof T]`) collects all of those property values into a single union.

By changing the condition inside `StringKeys<T>`, you can extract keys for numbers, booleans, arrays, and more.

Conditional Types

Conditional types are one of TypeScript's most powerful **type-level building blocks**. They let you say things like *"if this type extends that type, then produce this other type"* entirely within the type system.

You'll mostly see them in **libraries and reusable helpers** (often alongside generics and mapped types), but understanding the basics will also help you read and debug real-world TypeScript code.

What You'll Learn

- **01 – Conditional Types**
  Introduces the core syntax `T extends U ? X : Y`, shows how to read conditional types like a ternary expression, and walks through practical helpers such as an `IsString<T>` check and filtering unions with utilities like `Extract`, `Exclude`, and `NonNullable`.
- **02 – Infer**
  Explains the special `infer` keyword, which you can only use inside conditional types. You'll build a `GetReturnType<T>` helper (a simplified `ReturnType<T>`) and see how `infer R` lets you capture and reuse parts of a matched type, like a function's return type.
- **03 – Mapped Types**
  Builds up from index signatures to mapped types that transform object shapes. Using examples like `Soldier`, `OptionalSoldier`, and `Stringified<T>`, you'll see how `[K in keyof T]` and `T[K]` let you create reusable helpers such as `Optional<T>` and `Stringified<T>`.
- **04 – Mapped Types With Conditionals**
  Combines mapped types with conditional logic to selectively keep or change properties. You'll write types like `FilteredSoldier` that keep only properties whose values satisfy a constraint (for example, all properties whose type extends `string`).
- **05 – Extracting Keys From Types**
  Shows how to use the "map, then index" pattern—first creating a mapped type that marks some keys as `never`, then indexing with `[keyof T]`—to produce unions like *"all keys whose values are strings"*. This ties together conditionals, mapped types, and indexing to derive key unions from existing object types.

Quick Reference

Basic Conditional Type

```ts
// General form
type NewType<T> = T extends Condition ? TrueType : FalseType;

// Example
type IsString<T> = T extends string ? true : false;

type A = IsString<string>;  // true
type B = IsString<number>;  // false
```

Inferring Inside Conditionals

```
// Extract a function's return type
type GetReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

function greet() {
  return "hello";
}

type GreetReturn = GetReturnType<typeof greet>; // string
```

Mapped Types and Conditionals

```
// Make all properties optional
type Optional<T> = {
  [K in keyof T]?: T[K];
};

// Keep only string-valued properties
type StringProperties<T> = {
  [K in keyof T]: T[K] extends string ? T[K] : never;
};

// Turn matching properties into a key union
type StringPropertyKeys<T> = StringProperties<T>[keyof T];
```

When to Reach for Conditional Types

- When you need **derived types** that stay in sync with a source type (for example, all mouse events, all string properties, or non-nullable fields).
- When you're building **reusable helpers or libraries** that must work across many different input types.
- When the built-in utilities (`Extract`, `Exclude`, `NonNullable`, `ReturnType`, `Partial`, etc.) get you partway there but you need a custom variation.

They are incredibly powerful, but they can also become hard to read. Prefer the **simplest type that clearly expresses your intent**, and reach for conditional types (and `infer`) when they give you a clear correctness or maintainability win.

Install TypeScript

In this chapter we will move from running TypeScript in the browser to running it directly on your machine.

This guide assumes that you already have **Node.js** and **npm** installed. If you do not, install them first from the official Node.js website or your system package manager.

Global installation

To install the TypeScript compiler globally, run the following command:

```
npm install -g typescript
```

After this completes, you should be able to use the TypeScript compiler via the `tsc` command.

Verifying the installation

Check that `tsc` is available and see which version you have installed:

```
tsc -v
```

You should see a version string such as:

```
Version 5.x.x
```

If you do not see a valid version, either the installation failed or the location where `npm` installs global binaries is not in your `PATH`.

On many systems, `npm` installs global binaries into a directory such as:

- macOS (with nvm): `/Users/<you>/.nvm/versions/node/<node-version>/bin`
- Linux (with nvm): `/home/<you>/.nvm/versions/node/<node-version>/bin`

Make sure that the appropriate directory is included in your `PATH` environment variable. Once `tsc` is available on your `PATH`, you are ready to compile TypeScript from the command line.

tsconfig.json

The `tsconfig.json` file configures how the TypeScript compiler behaves. It controls what JavaScript gets emitted, which libraries are available, and what kinds of errors the compiler will report.

Because there are so many configuration options, it is rarely accurate to say that "TypeScript will always error on X". In practice, whether something is an error or not often depends on how your `tsconfig.json` is set up.

A simple tsconfig.json

Here is a minimal example that is reasonable for a brand-new project:

```
{
    "compilerOptions": {
        "lib": ["esnext"],
        "target": "esnext"
    }
}
```

This configuration sets two important options under `compilerOptions`:

- `lib` specifies which JavaScript library definitions are included during compilation. This determines which built-in APIs (such as `Promise`, `Map`, or `Array.prototype.find`) TypeScript knows about and can type-check.
- `target` specifies which ECMAScript version TypeScript should emit. This affects the shape of the JavaScript that gets written to disk.

Using `"esnext"` for both `lib` and `target` is convenient when you are starting a new project and want to write against the latest JavaScript features. However, before you ship a 1.0 release, it is a good idea to pin these to a specific version that matches the runtime environments you intend to support. For example:

```
{
    "compilerOptions": {
        "lib": ["es2024"],
        "target": "es2024"
    }
}
```

Locking your configuration to a specific ECMAScript year makes your build more predictable and avoids surprises if the meaning of `"esnext"` changes as the language evolves.

More tsconfig.json options

The full list of `tsconfig.json` options is quite large, so you will usually refer to the official documentation when you need something specific. That said, there are a few high-value `compilerOptions` that are worth understanding early on.

Here is a sketch of what a more fleshed-out configuration might look like:

```
{
    "compilerOptions": {
        "lib": ["es2024", "dom", "dom.iterable"],
        "target": "es2024",
        "strict": true,
        "skipLibCheck": true,
        "verbatimModuleSyntax": true,
        "esModuleInterop": true,
        "moduleDetection": "force",
        "noUncheckedIndexedAccess": true
    }
}
```

Some of the most common and useful options are:

- `lib`: Adds built-in library definitions. If you are writing browser code, include `"dom"` and `"dom.iterable"` (note the lowercase) so that TypeScript knows about the DOM APIs like `document`, `window`, and friends.
- `strict`: When `true`, enables all strict type-checking options. This catches many bugs at compile time. It is strongly recommended for new codebases, though you may temporarily disable it when migrating a large existing JavaScript project.
- `skipLibCheck`: When `true`, TypeScript skips type-checking `.d.ts` declaration files (for example, everything in `node_modules`). This can drastically reduce compile times without meaningfully affecting type safety for most projects.
- `verbatimModuleSyntax`: When `true`, TypeScript preserves your import and export syntax more literally and requires that type-only imports and exports use the `import type` / `export type` form. This removes some of the historical quirks around how imports are transformed and makes it clearer which imports exist only at type-checking time.
- `esModuleInterop`: When `true`, makes it easier to interoperate with CommonJS modules by allowing default-style imports (`import fs from "fs"`) from modules that were originally written for `require`. This is especially handy when working with Node-style packages.
- `moduleDetection`: When set to `"force"`, treats every file as a module, even if it does not explicitly use `import` or `export`. This avoids some legacy script-mode behavior and is generally what you want for modern TypeScript projects.
- `noUncheckedIndexedAccess`: When `true`, any indexed access like `obj[key]` or `arr[index]` gets `undefined` added to its type. This nudges you to handle missing values explicitly and can prevent subtle runtime errors.

You do not need to memorize all of these at once, but having a mental model of what they do will help you understand example `tsconfig.json` files you encounter in the wild.

Declaration Files

If you've ever seen funky-looking `.d.ts` files and wondered what they are, they're declaration files. A declaration file only contains type information – no runtime code is allowed. They're extremely useful for describing the types of JavaScript code that exists in your app but doesn't ship with its own types.

TypeScript uses these files to understand the shape of values at compile time. At runtime, declaration files are completely erased.

Why declaration files exist

Declaration files are most commonly used to:

- Add types for plain JavaScript libraries (especially ones loaded from a CDN).
- Describe globals that are provided by some script but not declared anywhere in your TypeScript source.
- Share type information across a project without bundling any extra JavaScript.

They let you keep the convenience of existing JavaScript libraries while still enjoying TypeScript's static checking and editor tooling.

Example: typing a global `google` object

For example, at Boot.dev we support login with Google. The codebase is written in TypeScript, but we include Google's JavaScript library in our HTML as per their documentation.

Because we want static type hints in our editors, we add a `globals.d.ts` file to our project that describes the global `google` object on `window`:

```
declare global {
    interface Window {
        google: Google;
    }
}

interface Google {
    accounts: {
        id: {
            renderButton: (
                element: HTMLElement,
                options: {
                    type?: string;
                    theme?: string;
                    size?: string;
                    text?: string;
                    shape?: string;
                    width?: number;
                },
            ) => void;
            prompt: () => void;
            cancel: () => void;
            initialize: (options: { client_id: string; callback: () => void }) => void;
            disableAutoSelect: () => void;
            revoke: (clientId: string, callback: () => void) => void;
        };
    };
}

export {};
```

This declaration file effectively says:

> "There's a global variable called `google` on the `window` object, and it has this shape."

Now when we use `window.google` in our TypeScript code, the compiler and our editor know exactly what properties and methods exist, and can provide autocomplete, type-checking, and inline documentation. The declaration file doesn't change how the code runs at all, but it makes our lives much easier when writing and maintaining the code.

Using JavaScript Libraries

When you start a new TypeScript project, it's easy to imagine that you'll have perfect type safety everywhere.

Then reality hits: you're asked to `npm install` a JavaScript library that doesn't ship with type definitions.

What do you do when that happens?

You generally have three options:

1. **Let `any` flow through your code** (fast, but you lose type safety).
2. **Create your own type definitions** for the library.
3. **Use community-maintained types** from DefinitelyTyped.

DefinitelyTyped is a large, community-driven repository of type definitions for popular JavaScript libraries. In practice, your first step should be to check there, usually via an `@types/...` package on npm.

However, this handbook is about understanding how things work under the hood, so let's focus on **writing your own type definitions** for an existing JavaScript library.

Creating a `.d.ts` file for an external module

Suppose you're using a JavaScript package called `pregnantgoku` that is published on npm but has no type declarations.

You can teach TypeScript about this package by adding a declaration file alongside your code. For example, create a file called `pregnantgoku.d.ts` somewhere in your project that's picked up by TypeScript (e.g. inside `src` or another included folder) and write:

```
declare module "pregnantgoku" {
    export function kamehameha(target: [number, number]): void;

    export type Saiyan = {
        name: string;
        monthsAlong: number;
        powerLevel: number;
    };
}
```

Now when you write code like this:

```
import { kamehameha, Saiyan } from "pregnantgoku";

const goku: Saiyan = {
    name: "Goku",
    monthsAlong: 9,
    powerLevel: 9001,
};

kamehameha([0, 1]);
```

TypeScript will use the declarations from `pregnantgoku.d.ts` to type-check your usage.

Declaration files for internal JavaScript modules

Sometimes you're not consuming a third-party package from `node_modules`, but rather a JavaScript file that lives inside your own project, for example `pregnantgoku.js`.

In that case, you can create a sibling declaration file called `pregnantgoku.d.ts` that exports the same surface area as your JavaScript module:

```ts
export function kamehameha(target: [number, number]): void;

export type Saiyan = {
    name: string;
    monthsAlong: number;
    powerLevel: number;
};
```

Because these are **file-based** declarations, you don't need the `declare module "..." {}` wrapper. TypeScript will automatically match `pregnantgoku.d.ts` to `pregnantgoku.js` and use the types whenever you import from that file.

Summary

- Prefer existing types from DefinitelyTyped when they're available.
- When they aren't, you can write your own `.d.ts` files to describe JavaScript libraries.
- Use `declare module "name" { ... }` for packages from `node_modules`.
- Use plain `export` declarations in a `.d.ts` file that sits next to a JavaScript file in your own project.

TypeScript Language Server

So far we've used `tsc` from the command line to compile TypeScript, but in most projects that's not how you interact with TypeScript day-to-day.

Most of the time, TypeScript is running as a **language server** inside your editor, powering features like:

- Auto-completion and inline suggestions
- Type checking with underlined errors and quick fixes
- Go to definition / find references / rename symbol

This is why people sometimes joke that TypeScript is a "glorified linter" – you might never manually run `tsc`, but you rely on its analysis constantly through your editor.

The important thing to understand is that your **editor tooling** and your **build tooling** are separate things that both use TypeScript, and they can drift out of sync.

For example:

- Your editor might be using TypeScript 4.x and one `tsconfig.json`.
- Your build (Vite, Webpack, `tsc`, etc.) might be using TypeScript 5.x and a different `tsconfig.json`.

If that happens, you can end up in situations where:

- The editor shows no errors, but the build fails.
- The editor reports errors, but the build is totally fine.

You want your **editor** and **build** to agree about:

- The TypeScript version
- Which files are in the project
- The compiler options in `tsconfig.json`

Which TypeScript Version Is the Editor Using?

Most editors will **prefer the TypeScript version installed in your project**. If you have a `node_modules` folder with `typescript` installed, the language server should use that version.

If your editor can't find a workspace-local TypeScript, it will fall back to something else, usually one of:

- A globally installed `typescript` on your machine
- A TypeScript version bundled with the editor itself

- A TypeScript version explicitly configured in editor settings (for example: `.vscode/settings.json`, `.zed/settings.json`, etc.)

To avoid confusion:

- **Pin `typescript` in your project** (for example in `package.json`).
- **Open the editor at the project root**, so it can find `node_modules/typescript` and your `tsconfig.json`.
- **Check your editor's TS version view** (most have a command or status bar item to show which TS it's using).

Keeping your editor and build on the **same TypeScript version and config** will save you from a lot of "why does CI see an error that I don't?" style bugs.

Restarting the TypeScript Server

Language servers are notorious for sometimes getting stuck or out of sync. If you notice that:

- Errors aren't updating
- Auto-completion stops working
- Go to definition behaves strangely

...then step one is often simply: **restart the TypeScript language server**.

A few examples:

- **VS Code**: `Cmd/Ctrl+Shift+P` → `TypeScript: Restart TS Server`
- **Zed**: `Cmd/Ctrl+Shift+P` → `Restart Language Server`
- **Neovim (built on LSP)**: `:LspRestart`

If something looks wrong in your TypeScript tooling, it's always worth checking:

1. Am I using the TypeScript version from this project?
2. Is the right `tsconfig.json` being picked up?
3. Have I tried restarting the language server?

Those three checks solve a surprising number of problems.

TypeScript Ignore Directives

I try really hard to avoid using these, but sometimes they are the least-bad option.

TypeScript has a few special comments that can suppress type errors:

```
// @ts-ignore: Ignores the next line's errors.

// @ts-ignore
const x: number = "not a number"; // Error suppressed

// @ts-nocheck: Disables type checking for the entire file.

// @ts-nocheck

const y: number = "not a number"; // No error

function sum(x: number, y: number): string {
    return x + y; // No error
}
```

These comments do exactly what they say: they **turn off TypeScript's help**.

- `@ts-ignore` hides all errors on the *next* line.
- `@ts-nocheck` disables type checking for the *entire file*.

Use them **very sparingly**, or ideally, not at all.

When you feel tempted to reach for one of these, consider instead:

- Narrowing or refining the types so the error is genuinely fixed.
- Adding an explicit type assertion (with a comment explaining why) if you truly know more than the compiler.
- Isolating unsafe code behind a small, well-documented helper.

If you do end up using `@ts-ignore` or `@ts-nocheck`, treat it as technical debt:

- Add a short justification comment next to it.
- Prefer `@ts-ignore` on a single line over `@ts-nocheck` for an entire file.
- Revisit these directives periodically and remove them once the underlying issue is resolved.

These tools are escape hatches, not everyday instruments.

Vanilla Vite

So far we've built a small TypeScript script and compiled it manually with the TypeScript compiler (`tsc`). Now let's scrap that setup and see what it looks like to scaffold a front-end project using Vite instead.

These days it's uncommon to build a front-end application without some kind of build tool. You certainly *can* — we just did — but in real projects you'll almost always reach for a tool, especially since most front-end frameworks (React, Vue, Svelte, and others) ship with their own build setups.

What Is Vite?

Vite is a relatively new build tool that focuses on speed and simplicity. It works great for framework projects, but it's also a solid choice for "vanilla" TypeScript apps.

Vite gives you a few important pieces out of the box:

- **Fast tooling** – Vite uses `esbuild` (a super-fast bundler written in Go) for much of the heavy lifting during development, which is a big part of why it feels so snappy. For production builds it uses Rollup under the hood.
- **Simple configuration** – For a JS/TS tool, Vite's configuration surface is small and approachable. Compared to older tools like Webpack, it's usually much easier to get started.
- **Built-in dev server** – Vite runs a development server for you, serving your files and handling hot module replacement (HMR) so that changes appear in the browser almost instantly.
- **On-the-fly compilation** – During development, Vite compiles and transforms your code on demand, so you don't have to run `tsc` manually every time you make a change.

Local Development

In projects, you'll run TypeScript directly on your machine, wire it into a build tool, and rely on your editor's integration to give you fast feedback.

This section walks through the practical pieces you need to be productive with TypeScript in a real codebase.

What you'll learn

- **Install TypeScript** – How to install the TypeScript compiler on your machine, verify the `tsc` command works, and understand where npm puts global binaries.

- **tsconfig.json** – How to configure the TypeScript compiler, choose `lib` and `target` settings, and enable high-value options like `strict`, `skipLibCheck`, and `verbatimModuleSyntax`.
- **Declaration Files** – How `.d.ts` files describe the shape of values without shipping any runtime code, and how to use them to type globals and other JavaScript that doesn't have built-in types.
- **Using JavaScript Libraries** – Strategies for working with untyped JavaScript packages, from reaching for `@types` on DefinitelyTyped to writing your own declaration files for external and internal modules.
- **TypeScript Language Server** – How your editor uses TypeScript under the hood, how it can drift from your build configuration, and how to make sure both use the same TypeScript version and `tsconfig.json`.
- **TypeScript Ignore Directives** – The escape hatches (`@ts-ignore`, `@ts-nocheck`) that suppress type errors, when they might be necessary, and why they should be treated as technical debt.
- **Vanilla Vite** – How to move from manually running `tsc` to using Vite as a fast, modern build tool for a vanilla TypeScript front-end project.

Quick setup checklist

If you're starting a new TypeScript project on your machine, a sensible sequence is:

1. **Install Node.js and npm**, then **install TypeScript** so the `tsc` command is available (see *Install TypeScript*).
2. **Create a `tsconfig.json`** with reasonable defaults for your environment – start with modern `lib` and `target` values, then enable stricter options as you're comfortable (see *tsconfig.json*).
3. **Open the project in your editor** and make sure the **language server is using the workspace TypeScript version** from `node_modules`, not some global or bundled version (see *TypeScript Language Server*).
4. **Decide on a build tool**. For front-end work, scaffolding a project with Vite is a great default (see *Vanilla Vite*). For smaller scripts, running `tsc` directly might be enough.
5. **Add types for JavaScript libraries** you depend on – prefer `@types` packages from DefinitelyTyped, and fall back to writing your own `.d.ts` files when necessary (see *Declaration Files* and *Using JavaScript Libraries*).
6. **Avoid turning off the type checker**. Reach for `@ts-ignore` or `@ts-nocheck` only as a last resort, with comments explaining why they're there and a plan to remove them later (see *TypeScript Ignore Directives*).