TYPESCRIPT

Intermediate

Advanced Types · Generics Deep Dive · Decorators & Metadata
Async Patterns · Module Systems & Namespaces · Advanced OOP
Type Utilities & Inference · Error Handling & Validation
Design Patterns in TypeScript

Philip Osztromok

Generated with Claude

Table of Contents

TypeScript Intermediate — 9 Chapters

🔷 Advanced Types

Now that you've mastered TypeScript fundamentals, it's time to level up. Advanced types — unions, intersections, type guards, conditionals, and mapped types — are the tools that separate basic type safety from true type mastery. These patterns unlock the full power of TypeScript's type system.

Union Types: Multiple Possibilities

A union type represents a value that could be one of several types. Use the `|` (pipe) operator:

```
type Status = "success" | "error" | "pending";
function handleResponse(status: Status) {
  if (status === "success") {
    console.log("All good!");
  }
}

type ID = string | number;
const userId: ID = "user-123";  // ✅ Both valid
const postId: ID = 456;         // ✅ Both valid
```

String Literals

Restrict to specific string values. Great for enums-like behavior without the syntax overhead.

Union of Types

Mix primitives: `string | number`, or objects with different shapes.

Intersection Types: Combine Constraints

An intersection type combines multiple types into one. Use the `&` operator:

```
type HasName = { name: string };
type HasAge = { age: number };
type Person = HasName & HasAge;
const alice: Person = {
  name: "Alice",
  age: 30
}; // ✅ Must have both name and age
```

Union vs Intersection

Union (|)

Or: type can be one thing **or** another

```
type Result = string | number;
// value is either string OR number
```

Intersection (&)

And: type must have properties from both

```
type Combined = A & B;
// value must satisfy A AND B
```

Type Guards: Narrowing Union Types

When working with unions, TypeScript doesn't know which type you have. Type guards narrow the type so you can safely use type-specific operations:

```
type Input = string | number;
function process(value: Input) {
  // Problem: TypeScript doesn't know if it's string or number
// value.toUpperCase();  // ❌ Error: number doesn't have toUpperCase
// Solution: Type guard with typeof
if (typeof value === "string") {
    console.log(value.toUpperCase());  // ✅ Now safe
  } else {
    console.log(value + 10);  // ✅ Safe arithmetic
  }
}
```

`typeof` Guard

Check primitive types: `string`, `number`, `boolean`, `undefined`.

`instanceof` Guard

Check if a value is an instance of a class: `obj instanceof Error`.

Custom Guard

Write a function with return type `value is Type` for complex checks.

`in` Operator

Check if an object has a property: `"email" in user`.

Custom Type Guard Example

```
interface Dog { bark(): void; }
interface Cat { meow(): void; }
// Custom type guard function
function isDog(pet: any): pet is Dog {
  return typeof pet.bark === "function";
}

const myPet: Dog | Cat = getDog();

if (isDog(myPet)) {
  myPet.bark();  // ✅ TypeScript knows it's a Dog
} else {
  myPet.meow();  // ✅ TypeScript knows it's a Cat
}
```

Conditional Types: Types That Decide

Conditional types use the ternary operator to assign a type based on a condition:

```
type IsString<T> = T extends string ? true : false;
type A = IsString<"hello">;  // true
type B = IsString<number>;    // false
// Practical example: get the return type of a function
type GetReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
const add = (a: number, b: number) => a + b;
type AddReturn = GetReturnType<typeof add>;  // number
```

Key keyword: `infer` captures a type that will be inferred during type checking.

Mapped Types: Transform Types Systematically

Mapped types let you create new types by transforming properties of existing types:

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

// Make all properties optional
type UserPartial = {
  [K in keyof User]?: User[K];
};

// Result is equivalent to:
// {
//   id?: number;
//   name?: string;
//   email?: string;
// }
```

`keyof`

Gets all property names of a type as a union. `keyof User` → `"id" | "name" | "email"`

`in`

Iterates over the union. Creates a new property for each.

`[K]`

Accesses the type of a property. `User[K]` gets the type of property `K`.

Utility Types

TypeScript provides built-in mapped types: `Partial<T>`, `Readonly<T>`, `Record<K, V>`

Practical Mapped Type: Read-Only Config

```
interface AppConfig {
  apiUrl: string;
  timeout: number;
  retries: number;
}

// Make all config properties readonly and add getters
type ReadOnlyConfig = {
readonly [K in keyof AppConfig]: AppConfig[K];
};

const config: ReadOnlyConfig = {
  apiUrl: "https://api.example.com",
  timeout: 5000,
  retries: 3
};

// config.timeout = 10000;  // ❌ Error: readonly property
```

🏗️ Real-World Patterns

Pattern 1: Discriminated Unions (Tagged Unions)

Use a common property to distinguish between union members:

```
type Result<T> =
  | { status: "success"; data: T; }
  | { status: "error"; error: string; }
function handle<T>(result: Result<T>) {
  if (result.status === "success") {
    console.log(result.data);  // ✅ TypeScript knows data exists
  } else {
    console.log(result.error);  // ✅ TypeScript knows error exists
  }
}
```

Pattern 2: Extract Properties with Conditional Types

```
// Get only string properties from a type
type StringPropertiesOnly<T> = {
  [K in keyof T]: T[K] extends string ? K : never;
}[keyof T];

interface User {
  name: string;
  age: number;
  email: string;
}

type StringKeys = StringPropertiesOnly<User>;  // "name" | "email"
```

💻 Coding Challenges

Challenge 1: Union & Type Guard

Write a function that accepts either a `User` object or a user `id` (number). Inside, use a type guard to determine which was passed and handle each case differently.

Goal: Practice unions and type narrowing with `typeof`.

→ Solution

Challenge 2: Intersection & Custom Guard

Create two interfaces (e.g., `Admin` and `Employee`), then a type for a value that is both. Write a custom type guard function to check if something is an `Admin`.

Goal: Practice intersections and custom type predicates.

→ Solution

Challenge 3: Mapped Type

Take any interface and create a mapped type that makes all properties optional **and** readonly. Test it with a sample object.

Goal: Build comfort with `keyof`, `in`, and property transformations.

→ Solution

⚠️ Gotcha: Overly Broad Unions

Avoid unions that are too broad — they defeat the purpose of type safety. For example, `string | number | boolean | object | any` basically gives up on type checking. Use discriminated unions and type guards to narrow aggressively. The more specific your types, the more the compiler can help you.

🎯 What's Next

Now that you've mastered advanced types, we'll explore **Generics Deep Dive** — how to write reusable, type-safe code that works across different types. This is where TypeScript truly shines.

🔷 Generics Deep Dive

Generics are TypeScript's superpower for writing reusable, type-safe code. You've seen basic generics (`Array`, `Promise`), but true mastery comes from understanding constraints, variance, and how to compose generic types. This chapter teaches you to think in generics.

Generic Constraints: Restricting Type Parameters

By default, a type parameter `T` can be anything. Constraints let you say "T must satisfy this requirement":

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
  return obj[key];  // ✅ TypeScript knows key exists on T
}

const user = { name: "Alice", age: 30 };
const result = getProperty(user, "name");  // ✅ "name" is valid
// getProperty(user, "email");  // ❌ "email" is not a key of user
```

Key syntax: `K extends keyof T` means "K must be a key that exists on T."

Extends a Type

`T extends string` — T must be a string or a subtype of string.

Extends a Union

`T extends string | number` — T must be one of these types.

Extends keyof

`K extends keyof T` — K must be a property name of T.

Multiple Constraints

Chain them: `<T extends { length: number }>` — T must have a length property.

Example: Pick from an Array with Constraints

```
// Only works on objects with a specific property type
function pickByType<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

interface Config {
  port: number;
  host: string;
}

const config: Config = { port: 3000, host: "localhost" };
const port = pickByType(config, "port");  // ✅ type is number
// Type inference knows the exact return type based on which key you pick!
```

Default Type Parameters

Just like function parameters, type parameters can have defaults:

```
type Result<T = string> = {
  success: boolean;
  data: T;
};

type StringResult = Result;        // T defaults to string
type NumberResult = Result<number>; // T is explicitly number
const response: StringResult = {
  success: true,
  data: "hello" // No need to specify <string>
};
```

Variance: Covariance & Contravariance

Variance describes how generic types relate when their type parameters are subtypes of each other. This is subtle but crucial:

```
class Animal {}
class Dog extends Animal {}
// Arrays are COVARIANT in their type parameter
const dogs: Dog[] = [new Dog()];
const animals: Animal[] = dogs;  // ✅ Dog[] is assignable to Animal[]
// Functions are CONTRAVARIANT in their parameter type
const feedAnimal = (a: Animal) => { };
const feedDog = (d: Dog) => { };

const callback: (animal: Animal) => void = feedDog;  // ❌ Error!
// A function expecting a Dog can't be used where we pass any Animal
```

Covariance

If `Dog extends Animal`, then `Box<Dog>` extends `Box<Animal>`. Read-only types.

Contravariance

If `Dog extends Animal`, then `Callback<Animal>` extends `Callback<Dog>`. Function parameters.

Invariance

`Box<Dog>` is not assignable to `Box<Animal>`. Mutable types.

Why It Matters

Prevents type errors. Contravariance protects function parameter safety.

keyof and typeof: Type Introspection

`keyof` gets property names; `typeof` gets the type of a value:

```
const user = { name: "Alice", age: 30 };

// typeof gets the type of the value
type UserType = typeof user;  // { name: string; age: number }
// keyof gets all property names
type UserKeys = keyof UserType;  // "name" | "age"
// Combine them: create a type that maps keys to their values
type UserValues = UserType[keyof UserType];  // string | number
// Practical: generic object property getter
function createGetter<T, K extends keyof T>(obj: T, key: K) {
  return () => obj[key];
}
```

Generic Utility Types: The Toolkit

TypeScript provides built-in generic utilities for common transformations:

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

// Partial: make all properties optional
type PartialUser = Partial<User>;
// Pick: select specific properties
type UserPreview = Pick<User, "name" | "email">;
// Omit: exclude specific properties
type UserWithoutId = Omit<User, "id">;
// Record: map keys to a value type
type UserRoles = Record<"admin" | "user" | "guest", User[]>;
// Readonly: make all properties readonly
type ReadOnlyUser = Readonly<User>;
// Extract: types assignable to U from union T
type StringOrNumber = string | number | boolean;
type Strings = Extract<StringOrNumber, string>;  // string
// Exclude: opposite of Extract
type NonStrings = Exclude<StringOrNumber, string>;  // number | boolean
```

Partial / Required

Toggle optionality. `Partial<T>` makes all optional; `Required<T>` makes all required.

Pick / Omit

Slice a type. `Pick` selects properties; `Omit` excludes them.

Record

Build a type from keys. `Record<K, V>` creates an object with keys K, values V.

Extract / Exclude

Filter unions. `Extract` keeps matching types; `Exclude` removes them.

Building Your Own Generic Utilities

Chain utilities and constraints to build powerful reusable patterns:

Example: Extracting Function Parameters

```
// Get all parameter types from a function
type Parameters<T extends (...args: any[]) => any> =
  T extends (...args: infer P) => any ? P : never;
const add = (a: number, b: number) => a + b;

type AddParams = Parameters<typeof add>;  // [number, number]
// Now you can build around it
function callWithLogging<T extends (...args: any[]) => any>(
  fn: T,
  ...args: Parameters<T>
) {
  console.log("Calling with", args);
  return fn(...args);
}
```

Recursive Generics: Types That Call Themselves

Generics can reference themselves for deeply nested structures:

```
// Make a type recursive: useful for nested data
type NestedArray<T> = T | NestedArray<T>[];
const deep: NestedArray<number> = [
  1,
  [2, 3],
  [4, [5, 6]] // ✅ Arbitrary nesting is valid
];

// Practical: make object values deeply partial
type DeepPartial<T> = T extends object ? {
  [K in keyof T]?: DeepPartial<T[K]>
} : T;

interface Config {
  db: {
    host: string;
    port: number;
    ssl: {
      cert: string;
    }
  }
}

type PartialConfig = DeepPartial<Config>;  // All properties at all levels optional
```

🧪 Real-World Pattern: Generic API Response Handler

Type-Safe Fetch Wrapper

```
interface ApiResponse<T> {
  status: "success" | "error";
  data?: T;
  error?: string;
}

async function fetchJson<T>(url: string): Promise<T> {
  const response = await fetch(url);
  return response.json();
}

async function getUser(id: number) {
  const user = await fetchJson<{ id: number; name: string }>(
    `/api/users/${id}`
  );
  console.log(user.name);  // ✅ TypeScript knows name exists
}
```

💻 Coding Challenges

Challenge 1: Generic with Constraints

Write a function that takes an object and a key, and returns the value. Use constraints to ensure the key exists on the object. Bonus: make the return type exactly match the property type.

Goal: Practice `keyof` constraints and type inference.

→ Solution

Challenge 2: Build a Utility Type

Create a utility type `GetType<T, K>` that takes an object type and a key, and returns the type of that property. Test it on a sample interface.

Goal: Combine `keyof`, conditional types, and generics.

→ Solution

Challenge 3: Recursive Generic

Create a type `Flatten<T>` that "unwraps" nested arrays. For example, `Flatten<[1, [2, 3]]>` should be `1 | 2 | 3`.

Goal: Build recursive type logic with array handling.

→ Solution

⚠️ Gotcha: Over-Constraining Generics

Constraints are powerful but can make code rigid. Before constraining a type parameter, ask: "Does this really need to be restricted?" Sometimes accepting `any` (though not ideal) is simpler than a complex constraint. Use constraints to enforce safety, not just to feel thorough.

🎯 What's Next

With generics mastered, we'll explore **Decorators & Metadata** — a powerful (and experimental) feature for attaching type information and behavior to classes and properties.

✨ Decorators & Metadata

Decorators are TypeScript's way of adding metadata and behavior to classes, methods, properties, and parameters. They're powerful (and experimental), enabling frameworks like Angular to work their magic. This chapter teaches you how to read, write, and reason about decorators.

> ⚠️ **Note:** Decorators are an experimental feature. To use them, enable `"experimentalDecorators": true` in `tsconfig.json`. They're currently a TC39 proposal but already widely used in production.

What Are Decorators?

A decorator is a function that modifies a class, method, property, or parameter. It runs at class definition time and can inspect or alter the target. Decorators are prefixed with `@`:

```
@Log
class User {
  @Validate
  name: string;

  @Deprecated
oldMethod() { }
}

// Decorators are syntactic sugar for function wrapping:
// @Log class User { } is equivalent to:
class User { }
User = Log(User);
```

Key insight: A decorator is just a function that receives the target (class, method, etc.) and can return a modified version of it.

🏛️ Class Decorators

A class decorator receives the constructor function and can replace or enhance it:

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

@Sealed
class User {
  name = "Alice";
}

// Now you can't add properties to User or User.prototype
// User.newProp = 123;  // ❌ Error
```

Practical: Class Decorator for Logging

```
function Log(constructor: Function) {
  return class extends constructor {
    constructor(...args: any[]) {
      console.log(`Creating instance of ${constructor.name}`);
      super(...args);
    }
  };
}

@Log
class Database {
  constructor(url: string) {
    console.log(`Connecting to ${url}`);
  }
}

new Database("postgres://...");
// Output: Creating instance of Database
//         Connecting to postgres://...
```

🏷️ Property Decorators

A property decorator receives the class prototype and the property name. It can't return anything (it modifies in place):

```
function Uppercase(target: any, propertyKey: string) {
 let value: string;

 const getter = () => value;
 const setter = (newValue: string) => {
  value = newValue.toUpperCase();
 };

 Object.defineProperty(target, propertyKey, {
  get: getter,
  set: setter,
  enumerable: true,
  configurable: true
 });
}

class User {
 @Uppercase
 name: string = "";
}

const user = new User();
user.name = "alice";
console.log(user.name); // "ALICE" ✅
```

Signature

`(target: any, propertyKey: string)`

Use Cases

Validation, transformation, metadata attachment, lazy initialization.

Limitation

Can't return anything. Modifies the class prototype in place.

Runs When?

At class definition time, not when instances are created.

🎯 Method Decorators

A method decorator receives the prototype, method name, and a property descriptor. It can wrap or replace the method:

```
function TimeIt(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function(...args: any[]) {
    const start = Date.now();
    const result = originalMethod.apply(this, args);
    const elapsed = Date.now() - start;
    console.log(`${propertyKey} took ${elapsed}ms`);
    return result;
  };

  return descriptor;
}

class Calculator {
  @TimeIt
fibonacci(n: number): number {
    if (n <= 1) return n;
    return this.fibonacci(n - 1) + this.fibonacci(n - 2);
  }
}

const calc = new Calculator();
calc.fibonacci(10);  // Logs execution time
```

Signature

```
(target: any, propertyKey: string, descriptor: PropertyDescriptor)
```

Descriptor.value

The original method function. Replace or wrap it to modify behavior.

Common Patterns

Logging, timing, caching, error handling, authentication checks.

Return Value

Return the (modified) descriptor to finalize the change.

🔗 Parameter Decorators

A parameter decorator receives the prototype, method name, and parameter index. Typically used for metadata (requires reflecting metadata):

```
function Required(target: any, propertyKey: string | undefined, parameterIndex: number) {
  // Store metadata about this parameter
const existingMetadata = Reflect.getOwnMetadata("required", target, propertyKey) || [];
  existingMetadata[parameterIndex] = true;
  Reflect.defineMetadata("required", existingMetadata, target, propertyKey);
}

class User {
  create(@Required name: string, @Required email: string) {
    console.log(`Creating user: ${name} (${email})`);
  }
}

// Later, you could validate based on the metadata
const metadata = Reflect.getOwnMetadata("required", User.prototype, "create");
// metadata is [true, true] — both parameters are required
```

> **Note:** Parameter decorators alone can't enforce behavior. They're typically used with method decorators that read the metadata and validate.

⚙️ Decorator Factories: Parameterized Decorators

To pass options to a decorator, wrap it in a factory function:

```
function MinLength(min: number) {
  return function(target: any, propertyKey: string) {
    let value: string;

    const setter = (newValue: string) => {
      if (newValue.length < min) {
        throw new Error(`${propertyKey} must be at least ${min} characters`);
      }
      value = newValue;
    };

    Object.defineProperty(target, propertyKey, { set: setter });
  };
}

class User {
  @MinLength(3)
  name: string = "";
}

const user = new User();
user.name = "ab";  // ❌ Error: must be at least 3 characters
```

🪜 Real-World Pattern: Validation Decorator

Complete Validation System

```
function Validate(rules: { [key: string]: (value: any) => boolean }) {
  return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;

    descriptor.value = function(...args: any[]) {
      for (const [param, rule] of Object.entries(rules)) {
        const value = args[0];
        if (!rule(value)) {
          throw new Error(`Validation failed for ${param}`);
        }
      }
      return originalMethod.apply(this, args);
    };

    return descriptor;
  };
}

class UserService {
  @Validate({
    email: (v) => typeof v === "string" && v.includes("@")
  })
  createUser(email: string) {
    console.log(`User created: ${email}`);
  }
}

const service = new UserService();
service.createUser("alice@example.com");  // ✅
service.createUser("invalid");            // ❌ Error: Validation failed
```

💻 Coding Challenges

Challenge 1: Method Logging Decorator

Create a method decorator that logs when a method is called and what it returns. Apply it to a sample class method.

Goal: Practice method decorators and descriptor manipulation.

→ Solution

Challenge 2: Property Validator Decorator

Create a property decorator that enforces a constraint (e.g., email must be a valid format). Use it on a class property and test it.

Goal: Build property decorators with validation logic.

→ Solution

Challenge 3: Decorator Factory

Create a decorator factory (decorator with parameters) that limits method calls to N times. Apply it to a method and verify it stops after the limit.

Goal: Understand how decorator factories work and parameterize decorators.

→ Solution

⚠️ Gotcha: Decorators Run at Definition Time

Decorators execute when the class is defined, not when instances are created. If you need per-instance behavior, wrap the original method inside a decorator factory or use a method decorator that returns a wrapped function. Also, always remember: TypeScript's decorator output depends on your tsconfig settings — test with `experimentalDecorators` enabled.

🎯 What's Next

With decorators mastered, we'll explore **Async Patterns** — mastering Promises, async/await, error handling, and concurrent operations for real-world applications.

⚡ Async Patterns

Modern JavaScript is asynchronous. APIs don't return instantly; databases take time; user interactions are unpredictable. This chapter teaches you to write type-safe async code that handles delays, errors, and concurrent operations without callback hell.

Promises: The Foundation

A Promise represents a value that will (eventually) resolve or reject:

```
const promise: Promise<string> = new Promise<string>((resolve, reject) => {
  setTimeout(() => {
    resolve("Success!");
  }, 1000);
});

promise
  .then((result) => console.log(result))  // Logs "Success!" after 1s
  .catch((error) => console.error(error));   // Catches rejection
```

Pending

Initial state. The async operation is still running.

Fulfilled

Operation succeeded. The Promise holds a value.

Rejected

Operation failed. The Promise holds a reason (error).

Settled

Either fulfilled or rejected. No longer pending.

Type annotation: `Promise<T>` means "a Promise that resolves to T".

🎯 Async/Await: Synchronous-Looking Code

Async/await is syntactic sugar for Promises. It makes async code look like sync code:

Promises vs Async/Await

Promise .then() chain

```
function fetchUser(id: number) {
 return fetch(`/api/users/${id}`)
  .then(r => r.json())
  .then(user => {
   console.log(user);
   return user;
  })
  .catch(err => {
   console.error(err);
   throw err;
  });
}
```

Async/await

```
async function fetchUser(id: number) {
 try {
  const response = await fetch(`/api/users/${id}`);
  const user = await response.json();
  console.log(user);
  return user;
 } catch (err) {
  console.error(err);
  throw err;
 }
}
```

Key syntax:

- `async` before the function declares it returns a Promise
- `await` pauses execution until a Promise settles
- `try/catch` handles rejections like sync errors

Typing Async Functions

```
// Async function always returns a Promise
async function getUser(id: number): Promise<{ id: number; name: string }> {
  const response = await fetch(`/api/users/${id}`);
  return response.json();
}

// You DON'T write "Promise<Promise<T>>" even though await unwraps
// TypeScript automatically unwraps for you
const user = await getUser(1);  // type: { id: number; name: string }
// If you don't await, you get the Promise:
const promise = getUser(1);  // type: Promise<{ id: number; name: string }>
```

🛡️ Error Handling in Async Code

Always wrap async code in try/catch:

```
async function processData() {
 try {
  const data = await fetch("/api/data").then(r => r.json());
  if (!data) throw new Error("No data");
  return data;
 } catch (error) {
  if (error instanceof TypeError) {
   console.error("Network error:", error.message);
  } else if (error instanceof Error) {
   console.error("Error:", error.message);
  }
  throw error;  // Re-throw to propagate
 }
}
```

> 💡 **Always Catch or Re-throw**
>
> Unhandled Promise rejections crash silently. Either catch the error or re-throw it to propagate up the stack. If you intentionally ignore an error, at least add a `.catch(() => {})` to make it explicit.

🚀 Concurrent Operations: Running Multiple Promises

`Promise.all()` waits for all Promises to resolve. If any reject, it rejects:

```
async function fetchMultiple() {
 try {
  const [user, posts, comments] = await Promise.all([
   fetch("/api/user").then(r => r.json()),
   fetch("/api/posts").then(r => r.json()),
   fetch("/api/comments").then(r => r.json())
  ]);
  // All three fetches ran concurrently ✅
return { user, posts, comments };
 } catch (error) {
  console.error("One or more requests failed", error);
  throw error;
 }
}
```

Promise.all()

All succeed → returns array. Any fails → rejects immediately.

Promise.allSettled()

Waits for all, regardless of success/failure. Returns array of outcomes.

Promise.race()

First to settle (success or failure) wins. Returns first result.

Promise.any()

First to succeed wins. If all fail, rejects with aggregate error.

Example: Resilient Concurrent Requests

```
async function fetchAll() {
 const results = await Promise.allSettled([
  fetch("/api/critical"),    // Must succeed
fetch("/api/optional")    // Nice to have
 ]);

 const [critical, optional] = results;

 if (critical.status === "fulfilled") {
  console.log("Critical data:", critical.value);
 } else {
  throw Error("Critical request failed");
 }

 if (optional.status === "fulfilled") {
  console.log("Optional data:", optional.value);
 } else {
  console.warn("Optional request failed, continuing anyway");
 }
}
```

Async Iteration: for await...of

For loops that wait on each iteration:

```
async function * fetchPages(pageCount: number) {
  for (let i = 1; i <= pageCount; i++) {
    const data = await fetch(`/api/page/${i}`).then(r => r.json());
    yield data;
  }
}

async function processPages() {
  for await (const page of fetchPages(5)) {
    console.log("Processing page:", page);
  }
}
```

🪜 Real-World Pattern: Retry with Exponential Backoff

Resilient API Call with Retries

```
async function fetchWithRetry<T>(
  url: string,
  maxRetries: number = 3
): Promise<T> {
  for (let attempt = 0; attempt <= maxRetries; attempt++) {
    try {
      const response = await fetch(url);
      if (!response.ok) {
        throw new Error(`HTTP ${response.status}`);
      }
      return response.json();
    } catch (error) {
      if (attempt === maxRetries) throw error;

      // Exponential backoff: 100ms, 200ms, 400ms, ...
const delay = 100 * Math.pow(2, attempt);
      console.warn(`Retry ${attempt + 1}/${maxRetries} after ${delay}ms`);
      await new Promise(resolve => setTimeout(resolve, delay));
    }
  }
}

// Usage
const data = await fetchWithRetry<{ id: number; name: string }>("/api/data");
```

💻 Coding Challenges

Challenge 1: Async Function with Error Handling

Write an async function that fetches data from a URL, parses JSON, and handles network errors separately from parse errors. Return the data or throw an appropriate error.

Goal: Practice async/await and error handling.

→ Solution

Challenge 2: Concurrent Requests

Write an async function that fetches data from 3 different URLs concurrently using `Promise.all()`. Handle the case where one fails and all fail.

Goal: Understand concurrent operations and failure modes.

→ Solution

Challenge 3: Retry Logic

Build a function that retries a failing async operation (simulated by a random success/failure) up to N times before giving up.

Goal: Implement retry patterns with loops and timing.

→ Solution

⚠️ Gotcha: Forgetting to Await

Forgetting `await` returns a Promise instead of the value. The code still runs (no error), but you're working with a Promise when you expected the value. Always use `await` inside async functions when you need the result, or use `.then()` if not in an async context.

🎯 What's Next

With async patterns mastered, we'll explore **Module Systems & Namespaces** — how to organize code across files and manage dependencies in TypeScript.

📦 Module Systems & Namespaces

As codebases grow, organization matters. This chapter covers ES modules (the modern standard), CommonJS (Node.js legacy), path mapping (clean imports), barrel exports (organized re-exports), and namespaces (TypeScript-specific grouping). You'll learn how to structure code across files without chaos.

🌎 ES Modules: The Standard

ES modules (ESM) are the modern standard. Files are modules; imports/exports are explicit:

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

export const PI = 3.14159;

// app.ts
import { add, PI } from "./math";  // .ts extension is optional

console.log(add(2, 3));    // 5
console.log(PI);           // 3.14159
```

Named Exports

`export function foo()` — export by name. Import with `{ foo }`.

Default Export

`export default class User` — one default per file. Import as `import User from`.

Re-export

`export { foo } from "./module"` — pass-through import/export (barrel exports).

Namespace Import

`import * as math from "./math"` — import all as `math.add()`.

📼 CommonJS: Node.js Legacy

CommonJS (CJS) is Node.js's original module system. Still widely used, but ES modules are preferred:

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

module.exports = { add };

// app.ts (CommonJS style)
const { add } = require("./math");
console.log(add(2, 3));
```

ES Modules vs CommonJS

CommonJS (require/module.exports)

```
const math = require("./math");
module.exports = { myFunc };
```

- Dynamic (runtime)
- Synchronous
- Node.js built-in

ES Modules (import/export)

```
import math from "./math";
export { myFunc };
```

- Static (compile-time)
- Asynchronous
- JavaScript standard

Modern projects: Use ES modules (set `"module": "ESNext"` in tsconfig.json). CommonJS is legacy but still needed for older Node.js versions.

🗺️ Path Mapping: Cleaner Imports

Deep nested paths are messy. Path mapping lets you create short aliases:

```
// tsconfig.json
{
 "compilerOptions": {
  "baseUrl": ".",
  "paths": {
   "@/*": ["src/*"],
   "@utils/*": ["src/utils/*"],
   "@components/*": ["src/components/*"]
  }
 }
}

// Instead of:
import { formatDate } from "../../../../utils/format";

// Write:
import { formatDate } from "@utils/format";
```

> 💡 **Path Alias Conventions**
>
> `@` prefix is common in modern projects (similar to npm scopes). Use `@utils`, `@types`, `@components` for clarity. Keep the mapping shallow so paths stay readable.

📦 Barrel Exports: Organized Re-exports

A barrel export (index.ts) re-exports from submodules for cleaner public APIs:

```
// src/utils/string.ts
export function capitalize(s: string) {
  return s.charAt(0).toUpperCase() + s.slice(1);
}

// src/utils/math.ts
export function sum(a: number, b: number) {
  return a + b;
}

// src/utils/index.ts (barrel export)
export * from "./string";
export * from "./math";

// app.ts — clean, single import
import { capitalize, sum } from "@utils";
```

Without barrel: You'd need two imports. **With barrel:** One import gets everything.

📋 Ambient Declarations: Types for External Code

Sometimes you use libraries without type definitions. Ambient declarations (`.d.ts` files) provide types:

```
// types/global.d.ts
// Declare a global variable
declare const DEBUG: boolean;

// Declare a module (for libraries without types)
declare module "my-untyped-library" {
  export function doSomething(x: any): string;
}

// In app.ts
console.log(DEBUG);  // ✅ TypeScript knows about it
import { doSomething } from "my-untyped-library";
const result = doSomething(42);  // ✅ Typed
```

🔷 TypeScript Namespaces (Legacy Grouping)

Namespaces group related types without ES modules. They're pre-module, but still used in legacy codebases:

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

  export const PI = 3.14159;
}

// app.ts
const result = Math.add(2, 3);  // Access via namespace
```

> **Avoid namespaces in new code.** Use ES modules instead. Namespaces are kept for backward compatibility but modules are cleaner and standard.

Real-World Pattern: Well-Organized Project

Clean Project Structure with Path Aliases & Barrels

```
// Directory structure:
// src/
//   ├── types/        ← TypeScript types
//   │   ├── index.ts   (barrel)
//   │   └── user.ts
//   ├── utils/        ← Utility functions
//   │   ├── index.ts   (barrel)
//   │   ├── format.ts
//   │   └── validate.ts
//   ├── services/     ← Business logic
//   │   ├── index.ts   (barrel)
//   │   └── api.ts
//   └── app.ts        ← Entry point
// tsconfig.json
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@types/*": ["src/types/*"],
      "@utils/*": ["src/utils/*"],
      "@services/*": ["src/services/*"]
    }
  }
}

// src/types/index.ts
export * from "./user";

// src/utils/index.ts
export * from "./format";
export * from "./validate";

// src/app.ts — clean imports
import { User } from "@types";
import { formatDate, validateEmail } from "@utils";
import { UserService } from "@services";

const user: User = { id: 1, email: "alice@example.com" };
```

```
console.log(formatDate(new Date()));
```

💻 Coding Challenges

Challenge 1: Create a Barrel Export

Create three utility modules (math.ts, string.ts, array.ts) with exported functions. Then create an index.ts barrel that re-exports all. Test importing from the barrel.

Goal: Practice barrel exports and namespace organization.

→ Solution

Challenge 2: Set Up Path Aliases

Create a tsconfig.json with path aliases (@utils, @types, @services). Set up dummy modules under each and import using the aliases.

Goal: Configure and use path mapping.

→ Solution

Challenge 3: Ambient Declaration

Create a .d.ts file that declares a global DEBUG constant and a module type for an external library. Write code that uses both.

Goal: Practice ambient declarations for untyped external code.

→ Solution

⚠️ Gotcha: Circular Dependencies

Be careful with circular imports: A imports B, B imports A. TypeScript won't error, but at runtime it can cause undefined values. Refactor to break the cycle—move shared types to a third file both can import from.

🎯 What's Next

With module systems mastered, we'll explore **Advanced OOP** — abstract classes, access modifiers, static members, and design patterns in TypeScript.

🏛️ Advanced OOP

Object-oriented programming in TypeScript goes beyond basic classes. This chapter covers abstract classes (enforce contracts), access modifiers (control visibility), static members (class-level data), and getters/setters (computed properties). These patterns scale code from hundreds to millions of lines.

🔷 Abstract Classes: Enforce Contracts

An abstract class can't be instantiated. It defines a contract that subclasses must implement:

```
abstract class Animal {
  abstract makeSound(): void;  // Subclasses MUST implement
sleep() {
    console.log("Zzz...");  // Concrete method (optional to override)
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("Woof!");  // ✅ Implements abstract method
  }
}

const animal = new Animal();  // ❌ Error: can't instantiate abstract class
const dog = new Dog();        // ✅ OK
dog.makeSound();          // "Woof!"
```

Abstract Methods

`abstract methodName();` — no body. Subclasses must implement.

Abstract Properties

`abstract prop: Type;` — subclasses must define.

Concrete Methods

Regular methods with bodies. Subclasses can override or use as-is.

Use Case

Define a template. Subclasses fill in the details.

🔒 Access Modifiers: Control Visibility

TypeScript provides `public`, `private`, and `protected` to control who can access members:

```
class BankAccount {
  public accountHolder: string;       // Anyone can read/write
private balance: number = 0;     // Only this class can access
protected log: string[] = [];   // This class and subclasses
constructor(holder: string) {
    this.accountHolder = holder;
  }

  public deposit(amount: number) {
    this.balance += amount;
    this.recordLog(`Deposited ${amount}`);
  }

  private recordLog(message: string) {
    this.log.push(message);
  }

  protected getBalance() {
    return this.balance;  // Only subclasses can call
  }
}

const account = new BankAccount("Alice");
account.deposit(100);           // ✅ public
console.log(account.accountHolder);  // ✅ public
// account.balance;         // ❌ private — error
// account.recordLog("..."); // ❌ private — error
```

Access Levels

public (default)

```
public prop: string;
// Accessible everywhere
```

private

```
private prop: string;
// Only inside this class
```

protected

```
protected prop: string;
// This class + subclasses
```

readonly

```
readonly prop: string;
// Can't reassign after init
```

📌 Static Members: Class-Level Data

Static members belong to the class itself, not instances:

```
class Counter {
  static count: number = 0;

  static reset() {
    Counter.count = 0;
  }

  constructor() {
    Counter.count++;  // Increment class-level counter
  }
}

const c1 = new Counter();
const c2 = new Counter();

console.log(Counter.count);  // 2 — shared across all instances
Counter.reset();
console.log(Counter.count);  // 0
```

Static Properties

`static prop: Type;` — shared by all instances.

Static Methods

`static methodName() { }` — called on the class, not instances.

Access

Use `ClassName.property`, not `instance.property`.

Use Cases

Configuration, factories, utility functions, singletons.

🎯 Getters & Setters: Computed Properties

Getters and setters let you use properties while running custom logic:

```
class User {
  private _age: number = 0;

  get age(): number {
    return this._age;
  }

  set age(value: number) {
    if (value < 0) {
      throw new Error("Age can't be negative");
    }
    this._age = value;
  }
}

const user = new User();
user.age = 25;       // ✅ Calls setter with validation
console.log(user.age);  // ✅ Calls getter
user.age = -5;       // ❌ Error: Age can't be negative
```

Why use getters/setters?

- **Validation:** Check constraints before setting
- **Computed values:** Calculate on-the-fly
- **Side effects:** Log, notify, update cache
- **Clean API:** Property syntax instead of methods

Real-World Pattern: Sealed Class with Private

State

Robust Configuration Class

```
abstract class BaseConfig {
  abstract validate(): boolean;
}

class AppConfig extends BaseConfig {
  private _port: number;
  private _host: string;
  readonly version: string;
  static instance: AppConfig;

  private constructor(port: number, host: string) {
    super();
    this._port = port;
    this._host = host;
    this.version = "1.0.0";
  }

  static create(port: number, host: string): AppConfig {
    if (!AppConfig.instance) {
      AppConfig.instance = new AppConfig(port, host);
    }
    return AppConfig.instance;
  }

  get port(): number {
    return this._port;
  }

  set port(value: number) {
    if (value < 1 || value > 65535) {
      throw new Error("Invalid port");
    }
    this._port = value;
  }

  validate(): boolean {
    return this._port > 0 && this._host.length > 0;
  }
}

// Usage: factory pattern with singleton
```

```
const config = AppConfig.create(3000, "localhost");
console.log(config.port);  // 3000
```

💻 Coding Challenges

Challenge 1: Abstract Class with Multiple Subclasses

Create an abstract `Vehicle` class with abstract methods. Implement two subclasses (`Car`, `Bike`) that satisfy the contract.

Goal: Practice abstract classes and polymorphism.

→ Solution

Challenge 2: Access Control & Encapsulation

Create a `BankAccount` class with private balance, public deposit/withdraw methods, and a getter for balance. Ensure balance can't go negative.

Goal: Understand private/public boundaries and encapsulation.

→ Solution

Challenge 3: Static Singleton Pattern

Create a `Database` class with a static `getInstance()` method that returns a single instance (singleton). Add static initialization logic.

Goal: Implement the singleton pattern with static members.

→ Solution

⚠️ Gotcha: Private at Compile Time Only

TypeScript's `private` is compile-time only. In the compiled JavaScript, `private` fields are just regular properties—they're not truly inaccessible at runtime. For true privacy, use JavaScript's `#` private fields. But for most purposes, TypeScript's `private` is sufficient discipline.

🎯 What's Next

With advanced OOP patterns mastered, we'll explore **Type Utilities & Inference** — leveraging TypeScript's powerful type system to build reusable, composable type utilities.

🔧 Type Utilities & Inference

TypeScript's type system is powerful enough to build reusable type transformations. This chapter covers built-in utility types (Partial, Pick, Omit, Record, etc.), how to compose them, extract types from values, infer return types, and write type predicates. These tools unlock the full potential of the type system.

📚 Built-In Utility Types

TypeScript provides a standard library of utility types for common transformations:

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

// Partial — all properties optional
type PartialUser = Partial<User>;
// { id?: number; name?: string; email?: string; age?: number; }
// Required — all properties required
type RequiredUser = Required<PartialUser>;
// { id: number; name: string; email: string; age: number; }
// Readonly — all properties readonly
type ReadOnlyUser = Readonly<User>;
// { readonly id: number; readonly name: string; ... }
// Pick — select specific properties
type UserPreview = Pick<User, "name" | "email">;
// { name: string; email: string; }
// Omit — exclude specific properties
type UserWithoutPassword = Omit<User, "password">;
// { id: number; name: string; email: string; age: number; }
// Record — build object with specific keys
type UserRoles = Record<"admin" | "user" | "guest", User[]>;
// { admin: User[]; user: User[]; guest: User[]; }
```

Partial & Required

Toggle optionality. `Partial<T>` makes all optional; `Required<T>` makes all required.

Readonly

Make all properties immutable. Can't reassign after creation.

Pick & Omit

`Pick` selects properties; `Omit` excludes them. Slice a type.

Record

Build an object type from keys. `Record<K, V>` creates `{ [key in K]: V }`.

🔍 Extract & Exclude: Filter Unions

Work with unions by keeping or removing types:

```
type StringOrNumber = string | number | boolean;
// Extract — keep types assignable to U
type Strings = Extract<StringOrNumber, string>;
// string
// Exclude — remove types assignable to U
type NonStrings = Exclude<StringOrNumber, string>;
// number | boolean
// Practical: filter event types
type Events = "click" | "focus" | "blur" | "change";
type FocusEvents = Extract<Events, "focus" | "blur">;
// "focus" | "blur"
```

Extract Types From Functions

Infer function signatures to build related types:

```
function getUser(id: number): { id: number; name: string } {
  return { id, name: "Alice" };
}

// ReturnType — extract return type
type UserReturn = ReturnType<typeof getUser>;
// { id: number; name: string }
// Parameters — extract parameter types
type GetUserParams = Parameters<typeof getUser>;
// [id: number]
// ConstructorParameters — for class constructors
class User {
  constructor(name: string, age: number) { }
}

type UserConstructorParams = ConstructorParameters<typeof User>;
// [name: string, age: number]
```

🎯 Type Predicates: Smart Narrowing

Custom type guards that narrow types intelligently:

```
interface Dog {
  bark(): void;
}

interface Cat {
  meow(): void;
}

// Type predicate: "value is Dog" tells TypeScript to narrow
function isDog(animal: Dog | Cat): animal is Dog {
  return typeof (animal as Dog).bark === "function";
}

const pet: Dog | Cat = getPet();

if (isDog(pet)) {
  pet.bark();  // ✅ TypeScript knows pet is Dog
} else {
  pet.meow();  // ✅ TypeScript knows pet is Cat
}
```

🔗 Composing Utility Types

Chain utilities to build complex transformations:

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

// Exclude password, make all optional, make readonly
type SafeUserPreview = Readonly<Partial<Omit<User, "password">>>;
// Equivalent to:
// {
//   readonly id?: number;
//   readonly name?: string;
//   readonly email?: string;
// }
// Use it for API responses
const preview: SafeUserPreview = {
  name: "Alice",
  // email is optional, id is optional
};
```

Real-World Pattern: Generic API Wrapper

Build Type-Safe CRUD Operations

```
interface Entity {
  id: number;
  createdAt: Date;
  updatedAt: Date;
}

interface ApiResponse<T extends Entity> {
  status: "success" | "error";
  data?: T;
  error?: string;
}

class Repository<T extends Entity> {
  // Create uses partial (no id, timestamps auto-generated)
async create(data: Omit<T, "id" | "createdAt" | "updatedAt">): Promise<T> {
    // Simulate API call
return {
      ...data,
      id: Math.random(),
      createdAt: new Date(),
      updatedAt: new Date()
    } as T;
  }

  // Update uses partial (update only what changed)
async update(id: number, data: Partial<Omit<T, "id" | "createdAt">>): Promise<T> {
    // Update logic here
return {} as T;
  }
}

// Usage with User type
interface User extends Entity {
  name: string;
  email: string;
}

const userRepo = new Repository<User>();

// Create requires name and email, NOT id/timestamps
```

```
await userRepo.create({ name: "Alice", email: "alice@example.com" });

// Update can be partial
await userRepo.update(1, { name: "Bob" });  // ✅ Only name
```

💻 Coding Challenges

Challenge 1: Build Custom Utilities

Create utility types: `GetType<T, K>` (property type), `Flatten<T>` (array), `Nullable<T>` (add null to all properties).

Goal: Build reusable utility types using conditional, mapped, and utility type composition.

→ Solution

Challenge 2: Type Predicates

Write custom type guard functions: `isString()`, `isArray<T>()`, `isRecord<T>()`. Test them with discriminated unions.

Goal: Master type predicates and smart narrowing.

→ Solution

Challenge 3: Compose Utilities

Use Pick, Omit, Partial, Readonly together to build derived types. Create safe API response types from your domain types.

Goal: Combine utilities to solve real-world problems.

→ Solution

⚠️ Gotcha: Deep Transformations

Most built-in utilities (Partial, Readonly, etc.) work only at the top level. If you need to make nested properties optional, you'll build recursive utilities. That's advanced, but it's possible using conditional types and mapped types.

🎯 What's Next

With type utilities mastered, we'll explore **Error Handling & Validation** — custom error types, runtime validation, and exhaustiveness checking for bullet-proof code.

🛡️ Error Handling & Validation

Errors are inevitable. This chapter teaches you to design robust error hierarchies, validate data defensively, handle errors gracefully, and use TypeScript's type system to catch errors at compile time. Good error handling separates amateur from professional code.

⚠️ Custom Error Classes: Structured Errors

Extend `Error` to create domain-specific error types:

```
class ValidationError extends Error {
  constructor(public field: string, message: string) {
    super(message);
    this.name = "ValidationError";
  }
}

class NetworkError extends Error {
  constructor(public statusCode: number, message: string) {
    super(message);
    this.name = "NetworkError";
  }
}

class DatabaseError extends Error {
  constructor(public query: string, message: string) {
    super(message);
    this.name = "DatabaseError";
  }
}

// Usage: throw with context
throw new ValidationError("email", "Invalid email format");
throw new NetworkError(404, "User not found");
```

Structured Data

Attach context: which field failed, what status code, what query.

Error.name

Set `this.name` so you can identify error types later.

Type Discrimination

Use `instanceof` or `error.name` to handle different errors differently.

Call super()

Always call `super(message)` to set the error message.

🎯 Handling Errors: Try-Catch Patterns

```
try {
  // Risky operation
const user = await fetchUser(id);
} catch (error) {
  // Always type-guard: error could be anything
if (error instanceof ValidationError) {
    console.error(`Field validation failed: ${error.field}`);
  } else if (error instanceof NetworkError) {
    console.error(`Server error: ${error.statusCode}`);
  } else if (error instanceof Error) {
    console.error(`Unexpected: ${error.message}`);
  } else {
    console.error("Unknown error");
  }
}
```

> ⚠️ **Important:** In TypeScript, the caught error is of type `unknown`, not `Error`. Always type-guard before using it.

✅ Validation: Fail Fast or Collect All?

Two Validation Strategies

Fail Fast (throw on first error)

```
function validate(data: User) {
 if (!data.email) {
  throw new ValidationError("email", "Required");
 }
 if (data.age < 18) {
  throw new ValidationError("age", "Must be 18+");
 }
}
```

Collect All (gather all errors)

```
function validate(data: User): ValidationError[] {
 const errors: ValidationError[] = [];

 if (!data.email) {
  errors.push(new ValidationError("email", "Required"));
 }
 if (data.age < 18) {
  errors.push(new ValidationError("age", "Must be 18+"));
 }

 return errors;
}
```

🔄 Result Type: No Exceptions

Instead of throwing, return success or error as data:

```
type Result<T, E = Error> =
  | { ok: true; value: T }
  | { ok: false; error: E };
async function parseJSON<T>(text: string): Result<T, SyntaxError> {
  try {
    const value = JSON.parse(text) as T;
    return { ok: true, value };
  } catch (error) {
    return { ok: false, error: error as SyntaxError };
  }
}

// Usage: no try-catch needed
const result = parseJSON<User>(jsonString);

if (result.ok) {
  console.log("Parsed:", result.value);  // ✅ User type
} else {
  console.error("Parse failed:", result.error.message);
}
```

🔍 Exhaustiveness: Catch Missing Cases

Use discriminated unions + the `never` type to ensure all cases are handled:

```
type FileOperation =
  | { type: "read"; path: string }
  | { type: "write"; path: string; content: string }
  | { type: "delete"; path: string };
function handleOperation(op: FileOperation): void {
  switch (op.type) {
    case "read":
      console.log(`Reading ${op.path}`);
      break;
    case "write":
      console.log(`Writing to ${op.path}`);
      break;
    case "delete":
      console.log(`Deleting ${op.path}`);
      break;
    default:
      // If you forget a case, TypeScript errors here
const _exhaustive: never = op;
      return _exhaustive;
  }
}
```

The trick: If you add a new case to the union but don't handle it, the `op` value has type `never`, which causes a compile error. Forces you to update all handlers.

Real-World Pattern: Robust API Call

Handle All Failure Modes

```
async function fetchUserSafe(id: number) {
  try {
    // 1. Validate input
if (id <= 0) {
      throw new ValidationError("id", "Must be positive");
    }

    // 2. Network call with timeout
const controller = new AbortController();
    const timeout = setTimeout(() => controller.abort(), 5000);

    const response = await fetch(`/api/users/${id}`, {
      signal: controller.signal
    });

    clearTimeout(timeout);

    // 3. Check status
if (!response.ok) {
      throw new NetworkError(response.status, response.statusText);
    }

    // 4. Parse and validate
const data = await response.json();
    if (!data.id || !data.name) {
      throw new Error("Invalid response shape");
    }

    return { ok: true, data };
  } catch (error) {
    if (error instanceof ValidationError) {
      return { ok: false, reason: "validation", error };
    } else if (error instanceof NetworkError) {
      return { ok: false, reason: "network", error };
    } else {
      return { ok: false, reason: "unknown", error };
    }
  }
```

```
}
```

💻 Coding Challenges

Challenge 1: Custom Error Hierarchy

Create an error base class and 3 subclasses (ValidationError, NetworkError, AuthError). Write a function that catches each type and handles differently.

Goal: Practice structured error handling.

→ Solution

Challenge 2: Validation with Error Collection

Write a validator that collects all errors instead of throwing on first. Return an array of errors or empty array on success.

Goal: Implement collect-all validation pattern.

→ Solution

Challenge 3: Result Type & Exhaustiveness

Implement a `Result<T, E>` type. Add a discriminated union operation type. Use exhaustiveness checking to ensure all operations are handled.

Goal: Combine Result type with exhaustiveness checking.

→ Solution

⚠️ Gotcha: The unknown Caught Error

In TypeScript, `catch(error)` has type `unknown`, not `Error`. Even if you only throw `Error` subclasses, something could throw a string or null. Always guard with `instanceof` or `typeof` before using the error.

🎯 What's Next

With error handling mastered, we'll explore the final chapter: **Design Patterns in TypeScript** — bringing together everything you've learned to build robust, scalable applications.

Design Patterns in TypeScript

Design patterns are proven solutions to common problems. This final chapter brings together everything you've learned—generics, decorators, async, error handling, type utilities—to build real-world patterns with TypeScript's type system. Master these patterns and you'll write professional-grade code.

👤 Singleton: One Instance

Ensure only one instance of a class exists globally:

```
class Logger {
  private static instance: Logger | null = null;
  private logs: string[] = [];

  private constructor() {}

  static getInstance(): Logger {
    if (Logger.instance === null) {
      Logger.instance = new Logger();
    }
    return Logger.instance;
  }

  log(message: string) {
    this.logs.push(message);
    console.log(`[${new Date().toISOString()}] ${message}`);
  }

  getLogs(): string[] {
    return [...this.logs];
  }
}

// Usage: always the same instance
const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

logger1 === logger2;  // true
logger1.log("Hello");
console.log(logger2.getLogs());  // ["Hello"]
```

🏭 Factory: Flexible Creation

Create objects without exposing creation logic. Useful for supporting multiple implementations:

```
interface Database {
  query(sql: string): Promise<any[]>;
  close(): Promise<void>;
}

class PostgreSQL implements Database {
  async query(sql: string) {
    console.log(`PostgreSQL: ${sql}`);
    return [];
  }
  async close() {}
}

class MySQL implements Database {
  async query(sql: string) {
    console.log(`MySQL: ${sql}`);
    return [];
  }
  async close() {}
}

// Factory: create the right DB without exposing the type
function createDatabase(type: "postgres" | "mysql"): Database {
  if (type === "postgres") {
    return new PostgreSQL();
  } else {
    return new MySQL();
  }
}

// Usage: caller doesn't know which DB they have
const db: Database = createDatabase("postgres");
await db.query("SELECT * FROM users");
```

👁️ Observer: Reactive Updates

Notify multiple objects when state changes (event-driven architecture):

```
interface Observer {
  update(data: unknown): void;
}

class Subject {
  private observers: Observer[] = [];

  attach(observer: Observer) {
    this.observers.push(observer);
  }

  detach(observer: Observer) {
    this.observers = this.observers.filter(o => o !== observer);
  }

  notify(data: unknown) {
    this.observers.forEach(o => o.update(data));
  }
}

// Concrete observers
class Logger implements Observer {
  update(data: unknown) {
    console.log(`[Log] ${JSON.stringify(data)}`);
  }
}

class EmailNotifier implements Observer {
  update(data: unknown) {
    console.log(`[Email] Sending notification for ${data}`);
  }
}

// Usage
const subject = new Subject();
subject.attach(new Logger());
subject.attach(new EmailNotifier());

subject.notify({ event: "user.created", userId: 1 });
```

⚙️ Strategy: Swappable Algorithms

Encapsulate algorithms so they're interchangeable at runtime:

```
interface SortStrategy {
  sort(arr: number[]): number[];
}

class BubbleSort implements SortStrategy {
  sort(arr: number[]): number[] {
    console.log("Sorting with Bubble Sort");
    // Bubble sort implementation
return [...arr].sort();
  }
}

class QuickSort implements SortStrategy {
  sort(arr: number[]): number[] {
    console.log("Sorting with Quick Sort");
    // Quick sort implementation
return [...arr].sort();
  }
}

class Sorter {
  constructor(private strategy: SortStrategy) {}

  execute(arr: number[]): number[] {
    return this.strategy.sort(arr);
  }
}

// Usage: swap strategies at runtime
const sorter1 = new Sorter(new BubbleSort());
console.log(sorter1.execute([3, 1, 2]));  // [1, 2, 3]
const sorter2 = new Sorter(new QuickSort());
console.log(sorter2.execute([3, 1, 2]));  // [1, 2, 3]
```

💉 Dependency Injection: Loose Coupling

Pass dependencies explicitly instead of hardcoding. Makes code testable and flexible:

```
// BAD: hardcoded dependency
class UserService_Bad {
  private db = new PostgreSQL();  // Tightly coupled
}

// GOOD: inject dependency
class UserService {
  constructor(private db: Database) {}

  async getUser(id: number) {
    return await this.db.query(`SELECT * FROM users WHERE id = ${id}`);
  }
}

// Usage: can swap DB implementation
const service = new UserService(new PostgreSQL());
await service.getUser(1);

// Testing: use mock
class MockDB implements Database {
  async query() { return [{ id: 1, name: "Test User" }]; }
  async close() {}
}

const testService = new UserService(new MockDB());
// Now testable without real database!
```

💻 Coding Challenges

Challenge 1: Singleton Logger

Implement a Logger singleton with methods to log at different levels (info, warn, error). Ensure only one instance exists across the app.

Goal: Practice singleton pattern with application-level state.

→ Solution

Challenge 2: Factory with Type Safety

Create a factory that builds different notification strategies (Email, SMS, Push). Use generics to ensure type safety across implementations.

Goal: Combine factory pattern with generics.

→ Solution

Challenge 3: Observer with Type-Safe Events

Implement an event bus with typed events (discriminated union). Add/remove listeners for specific events. Emit with type safety.

Goal: Combine Observer pattern with type utilities.

→ Solution

💡 Choose Patterns Carefully

Not every problem needs a design pattern. Start simple, refactor into patterns when complexity demands it. The best code is often the simplest code that solves the problem. Patterns are tools for specific situations, not dogma.

🎯 Course Complete!

You've mastered TypeScript Intermediate. You now understand advanced types, generics, decorators, async patterns, modules, OOP, type utilities, error handling, and design patterns. You're ready to build professional-grade TypeScript applications. Consider exploring Advanced TypeScript (Course 3) for even deeper mastery—or apply these skills to real projects, where the best learning happens.