TypeScript Complete Guide 📚

A comprehensive TypeScript learning guide organized into digestible chapters, from beginner to advanced concepts.

📖 Table of Contents

🟢 Beginner Level

🟡 Intermediate Level

🔴 Advanced Level

⚙️ Tooling and Configuration

🚀 Best Practices and Patterns

🎯 Learning Path Recommendations

For Complete Beginners

1. Start with Chapters 1-8 (Beginner Level)
2. Practice with small projects
3. Move to Chapters 9-15 (Intermediate Level)
4. Build a medium-sized project
5. Tackle Advanced chapters as needed

For JavaScript Developers

1. Quick read: Chapters 1-2
2. Focus on: Chapters 3-8, 11-13
3. Deep dive: Chapters 16-23
4. Tooling: Chapters 24-28
5. Best practices: Chapters 29-34

For Framework-Specific Learning

- **React Developers**: Chapters 1-8, 11-13, 25, 27-28
- **Node.js Developers**: Chapters 1-8, 11-15, 26-28
- **Library Authors**: Chapters 1-23, 30, 32-34

🗀 Repository Structure

```
TypeScript-Complete-Guide/
├── README.md                 # This file
├── chapters/                 # Individual chapter files
│   ├── 01-introduction.md
│   ├── 02-setup.md
│   └── ...
├── examples/                 # Code examples
│   ├── beginner/
│   ├── intermediate/
│   └── advanced/
├── configs/                  # Sample configuration files
│   ├── tsconfig-frontend.json
│   ├── tsconfig-backend.json
│   └── tsconfig-library.json
└── projects/                 # Sample projects
```

```
        ├── todo-app/
        ├── express-api/
        └── react-components/
```

🤝 How to Use This Guide

1. **Sequential Learning**: Follow chapters in order for comprehensive understanding
2. **Topic-Specific**: Jump to specific chapters based on your needs
3. **Reference**: Use as a quick reference for TypeScript concepts
4. **Practice**: Each chapter includes practical examples and exercises

📝 Prerequisites

- Basic JavaScript knowledge
- Familiarity with ES6+ features
- Understanding of programming concepts (variables, functions, objects)
- Node.js installed on your system

🎉 Getting Started

Ready to begin your TypeScript journey? Start with Chapter 1: Introduction to TypeScript!

This guide is designed to be a comprehensive resource for learning TypeScript. Each chapter builds upon previous concepts while remaining accessible for reference purposes.

Introduction to TypeScript

> Understanding what TypeScript is, why it matters, and how it improves JavaScript development

What is TypeScript?

TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. Developed by Microsoft, it adds optional static type checking to JavaScript, making code more predictable, maintainable, and easier to debug.

Why Choose TypeScript Over JavaScript?

🛡️ Type Safety

Catch errors at compile time instead of runtime, preventing many common bugs before they reach production.

```javascript
// JavaScript - Runtime error
function greet(name) {
  return "Hello, " + name.toUppercase(); // TypeError: name.toUppercase is not a function
}
```

```
// TypeScript - Compile-time error
function greet(name: string): string {
  return "Hello, " + name.toUppercase(); // Error: Property 'toUppercase' does not exist
  // Correct: name.toUpperCase()
}
```

🖌️ Enhanced IDE Support

- **Intelligent autocomplete**: Get suggestions based on actual types
- **Refactoring tools**: Rename variables, functions, and properties safely
- **Navigation**: Jump to definitions and find all references
- **Real-time error detection**: See errors as you type

📚 Self-Documenting Code

Types serve as inline documentation, making code intent clearer:

```
// Clear function signature tells you exactly what to expect
function calculateTax(price: number, taxRate: number): number {
  return price * (1 + taxRate);
}

// vs JavaScript where you need to guess or read documentation
function calculateTax(price, taxRate) {
  return price * (1 + taxRate);
}
```

🔄 Easier Refactoring

Confident code changes with comprehensive type checking across your entire codebase.

🚀 Modern JavaScript Features

Access to latest ECMAScript features with backward compatibility through compilation.

TypeScript vs JavaScript: Key Differences

Feature	JavaScript	TypeScript
Type checking	Runtime	Compile-time
Error detection	Runtime	Development
IDE support	Basic	Advanced
Learning curve	Lower	Higher
File extension	`.js`	`.ts`

Feature	JavaScript	TypeScript
Compilation	Not required	Required

Real-World Benefits

Large Codebases

TypeScript shines in large applications where:

- Multiple developers work on the same code
- Code complexity increases over time
- Refactoring becomes frequent
- API contracts need to be enforced

Team Collaboration

```
// Clear interface definitions help team members understand data structures
interface User {
  id: number;
  name: string;
  email: string;
  isActive: boolean;
  createdAt: Date;
  preferences?: UserPreferences;
}

interface UserPreferences {
  theme: "light" | "dark";
  notifications: boolean;
  language: string;
}
```

API Development

```
// Type-safe API responses
interface ApiResponse<T> {
  data: T;
  status: number;
  message: string;
  errors?: string[];
}

// Usage
const userResponse: ApiResponse<User> = await fetchUser(id);
```

When to Use TypeScript

✅ Great for:

- Large applications
- Team projects
- Long-term maintenance
- Complex business logic
- API development
- Library development

⚠ Consider carefully for:

- Small scripts
- Rapid prototyping
- Simple websites
- Learning JavaScript basics

Migration Strategy

You don't need to migrate everything at once:

1. **Start small**: Add TypeScript to new files
2. **Gradual adoption**: Convert existing files one by one
3. **Mixed codebase**: JavaScript and TypeScript can coexist
4. **Incremental typing**: Start with `any` and add specific types over time

```
// Start with any for quick migration
let userData: any = fetchUserData();

// Gradually add specific types
interface UserData {
  id: number;
  name: string;
}
let userData: UserData = fetchUserData();
```

TypeScript Ecosystem

Popular Frameworks with TypeScript Support

- **React**: Excellent TypeScript support
- **Angular**: Built with TypeScript
- **Vue.js**: First-class TypeScript support
- **Node.js**: Great for backend development
- **Next.js**: Full-stack TypeScript applications

Development Tools

- **VS Code**: Best-in-class TypeScript support

- **WebStorm**: Advanced TypeScript features
- **ESLint**: Code quality and style
- **Prettier**: Code formatting

Getting Started Checklist

- □ Install TypeScript globally or in your project
- □ Set up `tsconfig.json` configuration
- □ Configure your IDE for TypeScript
- □ Start with simple type annotations
- □ Learn about interfaces and types
- □ Explore advanced features gradually

Next Steps

Now that you understand what TypeScript is and why it's valuable, let's move on to setting up your development environment and writing your first TypeScript code.

Continue to: Setting Up TypeScript Development Environment

TypeScript Setup and Configuration

> Complete guide to installing TypeScript, setting up your development environment, and configuring your first project

Installation Methods

Global Installation

Install TypeScript globally to use the `tsc` command anywhere:

```
# Using npm
npm install -g typescript

# Using yarn
yarn global add typescript

# Using pnpm
pnpm add -g typescript

# Verify installation
tsc --version
```

Project-Specific Installation (Recommended)

Install TypeScript as a development dependency in your project:

```
# Using npm
npm install --save-dev typescript

# Using yarn
yarn add --dev typescript

# Using pnpm
pnpm add -D typescript
```

Why Project-Specific Installation?

- **Version consistency**: Different projects can use different TypeScript versions
- **Team collaboration**: Everyone uses the same TypeScript version
- **CI/CD compatibility**: Build systems use the exact version specified

Basic Project Setup

1. Initialize Your Project

```
# Create project directory
mkdir my-typescript-project
cd my-typescript-project

# Initialize package.json
npm init -y

# Install TypeScript
npm install --save-dev typescript

# Install Node.js types (for Node.js projects)
npm install --save-dev @types/node
```

2. Create TypeScript Configuration

```
# Generate tsconfig.json
npx tsc --init
```

3. Project Structure

```
my-typescript-project/
├── src/
│   ├── index.ts
│   └── utils/
│       └── helpers.ts
├── dist/
├── node_modules/
├── package.json
├── tsconfig.json
└── README.md
```

TypeScript Configuration (tsconfig.json)

Basic Configuration

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist", "**/*.test.ts"]
}
```

Key Configuration Options

Compilation Options

```
{
  "compilerOptions": {
    // Target JavaScript version
    "target": "ES2020", // ES5, ES6, ES2017, ES2018, ES2019, ES2020, ES2021,
ESNext

    // Module system
    "module": "commonjs", // commonjs, amd, es6, es2015, es2020, esnext

    // Output directory
    "outDir": "./dist",

    // Root directory of source files
    "rootDir": "./src",

    // Library files to include
    "lib": ["ES2020", "DOM"],

    // Module resolution strategy
    "moduleResolution": "node"
  }
}
```

Type Checking Options

```
{
  "compilerOptions": {
    // Enable all strict type checking options
    "strict": true,

    // Individual strict options (enabled by "strict")
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictBindCallApply": true,
    "strictPropertyInitialization": true,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "alwaysStrict": true,

    // Additional checks
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "exactOptionalPropertyTypes": true,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
    "noUncheckedIndexedAccess": true
  }
}
```

Development Options

```
{
  "compilerOptions": {
    // Generate source maps for debugging
    "sourceMap": true,

    // Generate declaration files
    "declaration": true,
    "declarationMap": true,

    // Remove comments from output
    "removeComments": false,
    // Import helpers from tslib
    "importHelpers": true,

    // Enable experimental decorators
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

Compilation Commands

Basic Compilation

```
# Compile all files
npx tsc

# Compile specific file
npx tsc src/index.ts

# Compile with custom config
npx tsc --project tsconfig.prod.json
```

Watch Mode

```
# Watch for changes and recompile
npx tsc --watch

# Watch with custom config
npx tsc --watch --project tsconfig.dev.json
```

Build Scripts

Add scripts to your `package.json`:

```
{
  "scripts": {
    "build": "tsc",
    "build:watch": "tsc --watch",
    "build:prod": "tsc --project tsconfig.prod.json",
    "clean": "rm -rf dist",
    "dev": "ts-node src/index.ts",
    "start": "node dist/index.js"
  }
}
```

Development Tools Setup

ts-node for Development

Run TypeScript files directly without compilation:

```
# Install ts-node
npm install --save-dev ts-node

# Run TypeScript file directly
npx ts-node src/index.ts

# With nodemon for auto-restart
npm install --save-dev nodemon
```

Create `nodemon.json`:

```
{
  "watch": ["src"],
  "ext": "ts",
  "exec": "ts-node src/index.ts"
}
```

VS Code Configuration

Create `.vscode/settings.json`:

```
{
  "typescript.preferences.importModuleSpecifier": "relative",
  "typescript.suggest.autoImports": true,
  "typescript.updateImportsOnFileMove.enabled": "always",
  "editor.codeActionsOnSave": {
    "source.organizeImports": true
  },
  "files.exclude": {
    "**/node_modules": true,
    "**/dist": true
  }
}
```

Create `.vscode/tasks.json` for build tasks:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "typescript",
      "tsconfig": "tsconfig.json",
      "option": "watch",
      "problemMatcher": ["$tsc-watch"],
      "group": "build",
      "label": "TypeScript: Watch"
    }
  ]
}
```

Environment-Specific Configurations

Development Configuration (tsconfig.dev.json)

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "sourceMap": true,
    "removeComments": false,
    "noUnusedLocals": false,
    "noUnusedParameters": false
  },
  "include": ["src/**/*", "tests/**/*"]
}
```

Production Configuration (tsconfig.prod.json)

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "sourceMap": false,
    "removeComments": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true
  },
  "exclude": ["node_modules", "tests", "**/*.test.ts", "**/*.spec.ts"]
}
```

Path Mapping

Simplify imports with path mapping:

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

Usage:

```
// Instead of
import { helper } from "../../../utils/helper";

// Use
import { helper } from "@utils/helper";
```

Common Setup Issues and Solutions

Issue: Module Not Found

```
# Install missing type definitions
npm install --save-dev @types/node
npm install --save-dev @types/express
```

Issue: Cannot Find Global Types

Add to `tsconfig.json`:

```
{
  "compilerOptions": {
    "types": ["node"],
    "typeRoots": ["./node_modules/@types"]
  }
}
```

Issue: Import/Export Errors

Ensure proper module configuration:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true
  }
}
```

Next Steps

With your TypeScript environment set up, you're ready to start learning about type annotations and basic TypeScript syntax.

Continue to: Basic Type Annotations

Basic Type Annotations

> Learn how to add type annotations to variables, parameters, and return values in TypeScript

What are Type Annotations?

Type annotations are explicit declarations that tell TypeScript what type of value a variable, parameter, or function return should be. They provide compile-time type checking and better IDE support.

```
// Without type annotation (JavaScript)
let message = "Hello World";

// With type annotation (TypeScript)
let message: string = "Hello World";
```

Primitive Types

String Type

```
let firstName: string = "John";
let lastName: string = "Doe";
let fullName: string = `${firstName} ${lastName}`; // Template literals

// String methods are fully typed
let upperName: string = firstName.toUpperCase();
let nameLength: number = firstName.length;
```

Number Type

```
let age: number = 25;
let price: number = 99.99;
let hexValue: number = 0xff; // Hexadecimal
let binaryValue: number = 0b1010; // Binary
let octalValue: number = 0o744; // Octal
```

```
// Number methods are typed
let rounded: number = price.toFixed(2);
let parsed: number = parseInt("42");
```

Boolean Type

```
let isActive: boolean = true;
let isComplete: boolean = false;
let hasPermission: boolean = age >= 18; // Expression result

// Boolean operations
let canAccess: boolean = isActive && hasPermission;
let shouldShow: boolean = !isComplete;
```

Null and Undefined

```
let nullValue: null = null;
let undefinedValue: undefined = undefined;

// Often used in union types
let optionalName: string | null = null;
let maybeAge: number | undefined = undefined;

// Strict null checks (when strictNullChecks is enabled)
let name: string = "John";
// name = null; // Error: Type 'null' is not assignable to type 'string'
```

Special Types

Any Type

The `any` type disables type checking - use sparingly!

```
let anything: any = "hello";
anything = 42;
anything = true;
anything = { name: "John" };
anything = [1, 2, 3];

// No type checking - can call any method
anything.foo.bar.baz; // No error, but might crash at runtime

// When to use any:
// 1. Migrating from JavaScript
// 2. Working with dynamic content
// 3. Third-party libraries without types
```

Unknown Type

Safer alternative to `any` - requires type checking before use:

```
let userInput: unknown;
userInput = "hello";
userInput = 42;
userInput = true;

// Must check type before using
if (typeof userInput === "string") {
  console.log(userInput.toUpperCase()); // OK: TypeScript knows it's a string
}

// Type assertion (use carefully)
let strValue: string = userInput as string;

// Type guard function
function isString(value: unknown): value is string {
  return typeof value === "string";
}

if (isString(userInput)) {
  console.log(userInput.length); // OK: TypeScript knows it's a string
}
```

Void Type

Used for functions that don't return a value:

```
function logMessage(message: string): void {
  console.log(message);
  // No return statement, or return; (without value)
}

function processData(data: any[]): void {
  data.forEach((item) => console.log(item));
  return; // OK: returning without value
}

// Variables of type void can only be undefined or null
let voidValue: void = undefined;
```

Never Type

Represents values that never occur:

```
// Function that never returns (throws error)
function throwError(message: string): never {
  throw new Error(message);
}

// Function with infinite loop
function infiniteLoop(): never {
  while (true) {
    // Do something forever
  }
}

// Exhaustive checking in switch statements
type Color = "red" | "green" | "blue";

function getColorCode(color: Color): string {
  switch (color) {
    case "red":
      return "#FF0000";
    case "green":
      return "#00FF00";
    case "blue":
      return "#0000FF";
    default:
      // This should never be reached
      const exhaustiveCheck: never = color;
      throw new Error(`Unhandled color: ${exhaustiveCheck}`);
  }
}
```

Type Inference

TypeScript can automatically infer types in many cases:

```
// Type inference - no annotation needed
let message = "Hello"; // Inferred as string
let count = 42; // Inferred as number
let isReady = false; // Inferred as boolean

// Function return type inference
function add(a: number, b: number) {
  return a + b; // Return type inferred as number
}

// Array type inference
let numbers = [1, 2, 3]; // Inferred as number[]
let mixed = ["hello", 42, true]; // Inferred as (string | number | boolean)[]
```

```
// Object type inference
let person = {
  name: "John",
  age: 30,
}; // Inferred as { name: string; age: number; }
```

When to Use Type Annotations

Always Annotate

```
// Function parameters (cannot be inferred)
function greet(name: string, age: number): string {
  return `Hello ${name}, you are ${age} years old`;
}

// When initial value doesn't match intended type
let userId: string | number = "user123";
// Later: userId = 456;

// When declaring without initialization
let userName: string;
// Later: userName = "John";
```

Optional Annotations

```
// Clear from context - annotation optional
let message: string = "Hello"; // Could be: let message = "Hello";

// When you want to be explicit
let price: number = 99.99; // Makes intent clear

// For better documentation
function calculateTax(amount: number): number {
  // Clear what function does
  return amount * 0.1;
}
```

Variable Declaration Patterns

Multiple Variables

```
// Same type
let firstName: string, lastName: string;
firstName = "John";
lastName = "Doe";
```

```
// Different types
let name: string = "John";
let age: number = 30;
let isActive: boolean = true;

// Destructuring with types
let [x, y]: [number, number] = [10, 20];
let { name: userName, age: userAge }: { name: string; age: number } = {
  name: "John",
  age: 30,
};
```

Const Assertions

```
// Regular const - type is widened
const colors = ["red", "green", "blue"]; // Type: string[]

// Const assertion - type is narrowed
const colorsConst = ["red", "green", "blue"] as const; // Type: readonly ["red",
"green", "blue"]

// Object const assertion
const config = {
  apiUrl: "https://api.example.com",
  timeout: 5000,
} as const; // Properties become readonly and literal types
```

Type Annotations in Practice

API Response Handling

```
// Define expected response structure
interface ApiResponse {
  data: any;
  status: number;
  message: string;
}

function handleApiResponse(response: ApiResponse): void {
  if (response.status === 200) {
    console.log("Success:", response.message);
    processData(response.data);
  } else {
    console.error("Error:", response.message);
  }
}

function processData(data: any): void {
  // Process the data
  console.log("Processing:", data);
}
```

Form Handling

```
// Form data types
interface UserForm {
  email: string;
  password: string;
  rememberMe: boolean;
}

function validateForm(formData: UserForm): boolean {
  const emailValid: boolean = formData.email.includes("@");
  const passwordValid: boolean = formData.password.length >= 8;

  return emailValid && passwordValid;
}

function submitForm(formData: UserForm): void {
  if (validateForm(formData)) {
    console.log("Form is valid, submitting...");
  } else {
    console.log("Form validation failed");
  }
}
```

Common Mistakes and Best Practices

❌ Common Mistakes

```
// Over-annotating when inference works
let message: string = "Hello"; // Unnecessary
let message = "Hello"; // Better

// Using any too liberally
let data: any = fetchData(); // Loses type safety

// Forgetting function parameter types
function greet(name) {
  // Error: Parameter 'name' implicitly has an 'any' type
  return `Hello ${name}`;
}
```

✅ Best Practices

```
// Let TypeScript infer when obvious
let count = 0; // Clear it's a number

// Annotate when intent isn't clear
let userId: string | number; // Will be assigned later

// Always annotate function parameters
function greet(name: string): string {
  return `Hello ${name}`;
}

// Use specific types over general ones
type Status = "pending" | "approved" | "rejected"; // Better than string
let orderStatus: Status = "pending";
```

Type Annotation Checklist

- ☐ Function parameters are always annotated
- ☐ Return types are annotated for public functions
- ☐ Variables are annotated when type isn't obvious
- ☐ Avoid `any` unless absolutely necessary
- ☐ Use `unknown` instead of `any` when possible
- ☐ Let TypeScript infer types when they're obvious
- ☐ Use specific types over general ones

Next Steps

Now that you understand basic type annotations, let's explore more complex type structures like interfaces and type aliases.

Continue to: Interfaces and Type Aliases

Interfaces and Type Aliases

> Learn how to define custom types using interfaces and type aliases, and understand when to use each

What are Interfaces?

Interfaces define the structure of objects, specifying what properties and methods an object should have. They act as contracts that ensure objects conform to a specific shape.

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

```
// Object must match the interface
const user: User = {
  id: 1,
  name: "John Doe",
  email: "john@example.com",
};
```

Basic Interface Syntax

Simple Interface

```
interface Product {
  id: number;
  name: string;
  price: number;
  description: string;
}

function displayProduct(product: Product): void {
  console.log(`${product.name}: $${product.price}`);
}

const laptop: Product = {
  id: 1,
  name: "MacBook Pro",
  price: 1999,
  description: "Powerful laptop for professionals",
};

displayProduct(laptop);
```

Optional Properties

```
interface UserProfile {
  id: number;
  username: string;
  email: string;
  avatar?: string; // Optional property
  bio?: string;
  website?: string;
}

// Valid - optional properties can be omitted
const user1: UserProfile = {
  id: 1,
  username: "johndoe",
  email: "john@example.com",
};
```

```
// Also valid - optional properties included
const user2: UserProfile = {
  id: 2,
  username: "janedoe",
  email: "jane@example.com",
  avatar: "avatar.jpg",
  bio: "Software developer",
};
```

Readonly Properties

```
interface Config {
  readonly apiUrl: string;
  readonly version: string;
  timeout: number; // Can be modified
}

const appConfig: Config = {
  apiUrl: "https://api.example.com",
  version: "1.0.0",
  timeout: 5000,
};

// appConfig.apiUrl = "new-url"; // Error: Cannot assign to 'apiUrl' because it is a read-only property
appConfig.timeout = 10000; // OK: timeout is not readonly
```

Interface Extension

Basic Extension

```
interface Animal {
  name: string;
  age: number;
}

interface Dog extends Animal {
  breed: string;
  bark(): void;
}

interface Cat extends Animal {
  color: string;
  meow(): void;
}

const myDog: Dog = {
  name: "Buddy",
  age: 3,
  breed: "Golden Retriever",
  bark() {
    console.log("Woof!");
  },
};
```

Multiple Extension

```
interface Flyable {
  fly(): void;
  altitude: number;
}

interface Swimmable {
  swim(): void;
  depth: number;
}

// Extending multiple interfaces
interface Duck extends Animal, Flyable, Swimmable {
  quack(): void;
}

const duck: Duck = {
  name: "Donald",
  age: 2,
  altitude: 0,
  depth: 0,
  fly() {
    this.altitude = 100;
    console.log("Flying!");
  },
  swim() {
    this.depth = 5;
    console.log("Swimming!");
  },
  quack() {
    console.log("Quack!");
  },
};
```

Function Types in Interfaces

Method Signatures

```
interface Calculator {
  // Method signature
  add(a: number, b: number): number;
  subtract(a: number, b: number): number;
  // Property with function type
  multiply: (a: number, b: number) => number;

  // Optional method
  divide?(a: number, b: number): number;
}

const calc: Calculator = {
  add(a, b) {
    return a + b;
  },
  subtract(a, b) {
    return a - b;
  },
  multiply: (a, b) => a * b,
  // divide is optional, so we can omit it
};
```

Event Handler Interfaces

```
interface EventHandler {
  onClick(event: MouseEvent): void;
  onKeyPress(event: KeyboardEvent): void;
  onSubmit?(event: SubmitEvent): void;
}

interface ButtonProps {
  text: string;
  disabled?: boolean;
  handler: EventHandler;
}

function createButton(props: ButtonProps): HTMLButtonElement {
  const button = document.createElement("button");
  button.textContent = props.text;
  button.disabled = props.disabled || false;

  button.addEventListener("click", props.handler.onClick);
  button.addEventListener("keypress", props.handler.onKeyPress);

  return button;
}
```

Index Signatures

String Index Signatures

```
interface StringDictionary {
  [key: string]: string;
}

const translations: StringDictionary = {
  hello: "Hola",
  goodbye: "Adiós",
  thanks: "Gracias",
};

// Can add any string key
translations.welcome = "Bienvenido";
```

Number Index Signatures

```
interface NumberArray {
  [index: number]: number;
  length: number; // Can have named properties too
}

const scores: NumberArray = {
  0: 95,
  1: 87,
  2: 92,
  length: 3,
};
```

Mixed Index Signatures

```
interface MixedDictionary {
  [key: string]: string | number;
  name: string; // Must be compatible with index signature
  age: number;
}

const person: MixedDictionary = {
  name: "John",
  age: 30,
  city: "New York",
  zipCode: 10001,
};
```

Type Aliases

Type aliases create new names for existing types, including complex type combinations.

Basic Type Aliases

```
// Primitive type alias
type UserID = string;
type Age = number;
type IsActive = boolean;

// Using type aliases
function getUser(id: UserID): User | null {
  // Implementation
  return null;
}

function updateAge(userId: UserID, newAge: Age): void {
  // Implementation
}
```

Object Type Aliases

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

type Rectangle = {
  topLeft: Point;
  bottomRight: Point;
};

function calculateArea(rect: Rectangle): number {
  const width = rect.bottomRight.x - rect.topLeft.x;
  const height = rect.bottomRight.y - rect.topLeft.y;
  return width * height;
}
```

Union Types

```
type Status = "pending" | "approved" | "rejected" | "cancelled";
type Theme = "light" | "dark" | "auto";
type Size = "small" | "medium" | "large";

// Union of different types
type ID = string | number;
type Response = string | { error: string } | { data: any };

function handleResponse(response: Response): void {
  if (typeof response === "string") {
    console.log("Message:", response);
  } else if ("error" in response) {
    console.error("Error:", response.error);
  } else {
    console.log("Data:", response.data);
  }
}
```

Function Type Aliases

```
// Function type aliases
type EventCallback = (event: Event) => void;
type Validator<T> = (value: T) => boolean;
type Transformer<T, U> = (input: T) => U;

// Using function type aliases
const emailValidator: Validator<string> = (email) => {
  return email.includes("@");
};

const stringToNumber: Transformer<string, number> = (str) => {
  return parseInt(str, 10);
};

function addEventListener(
  element: HTMLElement,
  event: string,
  callback: EventCallback
): void {
  element.addEventListener(event, callback);
}
```

Intersection Types

```
type Name = {
  firstName: string;
  lastName: string;
};

type Contact = {
  email: string;
  phone: string;
};

type Address = {
  street: string;
  city: string;
  zipCode: string;
};

// Intersection type - combines all properties
type Person = Name & Contact & Address;
```

```
const person: Person = {
  firstName: "John",
  lastName: "Doe",
  email: "john@example.com",
  phone: "555-1234",
  street: "123 Main St",
  city: "Anytown",
  zipCode: "12345",
};

// Intersection with interfaces
interface Timestamped {
  createdAt: Date;
  updatedAt: Date;
}

type UserWithTimestamp = User & Timestamped;
```

Interface vs Type Alias: When to Use Which?

Use Interfaces When:

```
// 1. Defining object shapes
interface User {
  id: number;
  name: string;
}

// 2. You need declaration merging
interface Window {
  customProperty: string;
}

// Later in another file...
interface Window {
  anotherProperty: number;
}
// Window now has both properties

// 3. Extending classes
class BaseUser implements User {
  constructor(public id: number, public name: string) {}
}

// 4. When you might extend later
interface AdminUser extends User {
  permissions: string[];
}
```

Use Type Aliases When:

```
// 1. Union types
type Status = "loading" | "success" | "error";

// 2. Intersection types
type UserWithRole = User & { role: string };

// 3. Computed properties
type Keys = "name" | "email";
type UserSubset = {
  [K in Keys]: string;
};

// 4. Complex type manipulations
type Optional<T> = {
  [K in keyof T]?: T[K];
};

// 5. Function types
type EventHandler = (event: Event) => void;

// 6. Primitive aliases
type UserID = string;
```

Advanced Interface Patterns

Generic Interfaces

```
interface Repository<T> {
  findById(id: string): T | null;
  save(entity: T): T;
  delete(id: string): boolean;
  findAll(): T[];
}

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

class UserRepository implements Repository<User> {
  private users: User[] = [];

  findById(id: string): User | null {
    return this.users.find((user) => user.id === id) || null;
  }

  save(user: User): User {
    this.users.push(user);
    return user;
  }

  delete(id: string): boolean {
    const index = this.users.findIndex((user) => user.id === id);
    if (index > -1) {
      this.users.splice(index, 1);
      return true;
    }
    return false;
  }

  findAll(): User[] {
    return [...this.users];
  }
}
```

Conditional Properties

```
interface BaseConfig {
  apiUrl: string;
  timeout: number;
}

interface DevConfig extends BaseConfig {
  debug: true;
  logLevel: "verbose" | "info" | "warn" | "error";
}

interface ProdConfig extends BaseConfig {
  debug: false;
  compression: boolean;
}

type Config = DevConfig | ProdConfig;

function createLogger(config: Config): void {
  if (config.debug) {
    // TypeScript knows this is DevConfig
    console.log(`Log level: ${config.logLevel}`);
  } else {
    // TypeScript knows this is ProdConfig
    console.log(`Compression: ${config.compression}`);
  }
}
```

Best Practices

✅ Good Practices

```typescript
// Use descriptive names
interface UserAccount {
  id: string;
  email: string;
}

// Group related properties
interface UserProfile {
  personal: {
    firstName: string;
    lastName: string;
    dateOfBirth: Date;
  };
  contact: {
    email: string;
    phone?: string;
  };
  preferences: {
    theme: "light" | "dark";
    notifications: boolean;
  };
}

// Use readonly for immutable data
interface ApiResponse {
  readonly data: any;
  readonly status: number;
  readonly timestamp: Date;
}
```

❌ Avoid

```typescript
// Don't use overly generic names
interface Data {
  value: any;
}

// Don't make everything optional
interface BadUser {
  id?: string;
  name?: string;
  email?: string;
}

// Don't use any when you can be specific
interface BadResponse {
  data: any; // Be more specific about the data structure
}
```

Next Steps

Now that you understand interfaces and type aliases, let's explore how to add types to functions, including parameters, return types, and overloads.

Continue to: Functions and Type Safety

Functions and Type Safety

> Master function typing in TypeScript, including parameters, return types, overloads, and advanced function patterns

Basic Function Typing

Function Declarations

```
// Basic function with typed parameters and return type
function add(a: number, b: number): number {
  return a + b;
}

// Function with no return value
function logMessage(message: string): void {
  console.log(message);
}

// Function that never returns
function throwError(message: string): never {
  throw new Error(message);
}
```

Function Expressions

```
// Arrow function
const multiply = (a: number, b: number): number => a * b;

// Function expression
const divide = function (a: number, b: number): number {
  return a / b;
};

// Function with explicit type annotation
const subtract: (a: number, b: number) => number = (a, b) => a - b;
```

Function Type Aliases

```
// Define function type
type MathOperation = (a: number, b: number) => number;
type StringProcessor = (input: string) => string;
type EventHandler = (event: Event) => void;

// Use function types
const add: MathOperation = (a, b) => a + b;
const toUpperCase: StringProcessor = (str) => str.toUpperCase();

// Function that accepts another function
function calculate(operation: MathOperation, x: number, y: number): number {
  return operation(x, y);
}

const result = calculate(add, 5, 3); // 8
```

Parameter Types

Optional Parameters

```
// Optional parameters must come after required ones
function greet(name: string, greeting?: string): string {
  return `${greeting || "Hello"}, ${name}!`;
}

greet("John"); // "Hello, John!"
greet("John", "Hi"); // "Hi, John!"

// Multiple optional parameters
function createUser(name: string, age?: number, email?: string): User {
  return {
    id: Math.random().toString(),
    name,
    age: age || 0,
    email: email || "",
  };
}
```

Default Parameters

```
// Default parameter values
function createConnection(
  host: string = "localhost",
  port: number = 3000,
  ssl: boolean = false
): Connection {
  return new Connection(host, port, ssl);
}
```

```
// Can call with any number of arguments
createConnection(); // Uses all defaults
createConnection("api.example.com"); // Custom host, default port and ssl
createConnection("api.example.com", 443, true); // All custom

// Default parameters can reference earlier parameters
function buildUrl(
  protocol: string = "https",
  host: string,
  path: string = "/"
): string {
  return `${protocol}://${host}${path}`;
}
```

Rest Parameters

```
// Rest parameters for variable arguments
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}

sum(1, 2, 3, 4, 5); // 15
sum(); // 0

// Rest parameters with other parameters
function logWithPrefix(prefix: string, ...messages: string[]): void {
  messages.forEach((message) => {
    console.log(`${prefix}: ${message}`);
  });
}

logWithPrefix("INFO", "Server started", "Database connected");

// Typed rest parameters
function combineObjects<T>(...objects: T[]): T {
  return Object.assign({}, ...objects);
}
```

Destructured Parameters

```
// Object destructuring in parameters
interface UserInfo {
  name: string;
  age: number;
  email: string;
}

function displayUser({ name, age, email }: UserInfo): string {
  return `${name} (${age}) - ${email}`;
}

// With default values
function createApiClient({
  baseUrl = "https://api.example.com",
  timeout = 5000,
  retries = 3,
}: {
  baseUrl?: string;
  timeout?: number;
  retries?: number;
} = {}): ApiClient {
  return new ApiClient(baseUrl, timeout, retries);
}

// Array destructuring
function getCoordinates([x, y]: [number, number]): string {
  return `(${x}, ${y})`;
}
```

Return Types

Explicit Return Types

```
// Explicit return type annotation
function getUser(id: string): User | null {
  // Implementation
  return users.find((user) => user.id === id) || null;
}

// Promise return types
function fetchUserData(id: string): Promise<User> {
  return fetch(`/api/users/${id}`).then((response) => response.json());
}

// Async function return types
async function getUserAsync(id: string): Promise<User | null> {
  try {
    const response = await fetch(`/api/users/${id}`);
    return await response.json();
  } catch (error) {
    return null;
  }
}
```

Return Type Inference

```
// TypeScript infers return types
function multiply(a: number, b: number) {
  return a * b; // Inferred as number
}

function getUsers() {
  return [
    { id: 1, name: "John" },
    { id: 2, name: "Jane" },
  ]; // Inferred as { id: number; name: string; }[]
}

// Complex inference
function processData(data: string[]) {
  return data
    .filter((item) => item.length > 0)
    .map((item) => ({ value: item, length: item.length }));
  // Inferred as { value: string; length: number; }[]
}
```

Function Overloads

Function overloads allow you to define multiple function signatures for the same function.

Basic Overloads

```
// Overload signatures
function format(value: string): string;
function format(value: number): string;
function format(value: boolean): string;

// Implementation signature (must be compatible with all overloads)
function format(value: string | number | boolean): string {
  return String(value);
}

// Usage
const str1 = format("hello"); // string
const str2 = format(42); // string
const str3 = format(true); // string
```

Complex Overloads

```
// Different return types based on parameters
function createElement(tag: "div"): HTMLDivElement;
function createElement(tag: "span"): HTMLSpanElement;
function createElement(tag: "button"): HTMLButtonElement;
function createElement(tag: string): HTMLElement;
function createElement(tag: string): HTMLElement {
  return document.createElement(tag);
}

// TypeScript knows the specific return type
const div = createElement("div"); // HTMLDivElement
const button = createElement("button"); // HTMLButtonElement

// Conditional overloads
function get(url: string): Promise<string>;
function get(url: string, options: { json: true }): Promise<object>;
function get(
  url: string,
  options?: { json?: boolean }
): Promise<string | object> {
  // Implementation
  return fetch(url).then((response) =>
    options?.json ? response.json() : response.text()
  );
}
```

Method Overloads

```
class DataProcessor {
  // Method overloads
  process(data: string): string;
  process(data: number): number;
  process(data: string[]): string[];

  process(data: string | number | string[]): string | number | string[] {
    if (typeof data === "string") {
      return data.toUpperCase();
    } else if (typeof data === "number") {
      return data * 2;
    } else {
      return data.map((item) => item.toUpperCase());
    }
  }
}

const processor = new DataProcessor();
const result1 = processor.process("hello"); // string
const result2 = processor.process(42); // number
const result3 = processor.process(["a", "b"]); // string[]
```

Generic Functions

Basic Generic Functions

```
// Generic function
function identity<T>(arg: T): T {
  return arg;
}

// Usage with explicit type
const stringResult = identity<string>("hello");
const numberResult = identity<number>(42);

// Usage with type inference
const autoString = identity("hello"); // T inferred as string
const autoNumber = identity(42); // T inferred as number
```

Generic Functions with Constraints

```
// Constraint: T must have a length property
function getLength<T extends { length: number }>(arg: T): number {
  return arg.length;
}

getLength("hello"); // OK: string has length
getLength([1, 2, 3]); // OK: array has length
// getLength(42); // Error: number doesn't have length

// Multiple constraints
function merge<T extends object, U extends object>(obj1: T, obj2: U): T & U {
  return { ...obj1, ...obj2 };
}

const merged = merge({ name: "John" }, { age: 30 });
// Type: { name: string } & { age: number }
```

Generic Functions with Multiple Type Parameters

```
// Multiple type parameters
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}

const stringNumberPair = pair("hello", 42); // [string, number]
const booleanArrayPair = pair(true, [1, 2, 3]); // [boolean, number[]]

// Generic function with conditional logic
function convert<T, U>(value: T, converter: (input: T) => U): U {
  return converter(value);
}
```

```
const stringToNumber = convert("42", parseInt); // number
const numberToString = convert(42, String); // string
```

Higher-Order Functions

Functions that Return Functions

```
// Function factory
function createValidator<T>(predicate: (value: T) => boolean) {
  return function (value: T): boolean {
    return predicate(value);
  };
}

const isPositive = createValidator<number>((n) => n > 0);
const isNotEmpty = createValidator<string>((s) => s.length > 0);

// Curried functions
function add(a: number) {
  return function (b: number): number {
    return a + b;
  };
}

const add5 = add(5);
const result = add5(3); // 8

// Generic curried function
function curry<T, U, V>(fn: (a: T, b: U) => V) {
  return function (a: T) {
    return function (b: U): V {
      return fn(a, b);
    };
  };
}
```

Callback Functions

```
// Typed callbacks
function processArray<T, U>(
  array: T[],
  callback: (item: T, index: number) => U
): U[] {
  return array.map(callback);
}

const numbers = [1, 2, 3, 4, 5];
const doubled = processArray(numbers, (n, i) => n * 2); // number[]
const strings = processArray(numbers, (n, i) => `Item ${i}: ${n}`); // string[]
```

```
// Event handlers
type EventCallback<T> = (event: T) => void;

function addEventListener<T extends Event>(
  element: HTMLElement,
  eventType: string,
  callback: EventCallback<T>
): void {
  element.addEventListener(eventType, callback as EventListener);
}
```

Function Type Guards

Custom Type Guards

```
// Type guard function
function isString(value: unknown): value is string {
  return typeof value === "string";
}

function isNumber(value: unknown): value is number {
  return typeof value === "number";
}

// Using type guards
function processValue(value: unknown): string {
  if (isString(value)) {
    return value.toUpperCase(); // TypeScript knows value is string
  } else if (isNumber(value)) {
    return value.toString(); // TypeScript knows value is number
  } else {
    return "Unknown type";
  }
}

// Generic type guard
function isArrayOf<T>(
  value: unknown,
  guard: (item: unknown) => item is T
): value is T[] {
  return Array.isArray(value) && value.every(guard);
}

const maybeStringArray: unknown = ["a", "b", "c"];
if (isArrayOf(maybeStringArray, isString)) {
  // TypeScript knows maybeStringArray is string[]
  console.log(maybeStringArray.join(", "));
}
```

Async Functions and Promises

Promise-based Functions

```
// Function returning Promise
function fetchUser(id: string): Promise<User> {
  return fetch(`/api/users/${id}`).then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);
    }
    return response.json();
  });
}

// Generic Promise function
function delay<T>(ms: number, value: T): Promise<T> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(value), ms);
  });
}

// Promise utility functions
function timeout<T>(promise: Promise<T>, ms: number): Promise<T> {
  return Promise.race([
    promise,
    delay(ms, null).then(() => {
      throw new Error("Timeout");
    }),
  ]);
}
```

Async/Await Functions

```
// Async function
async function getUserData(id: string): Promise<UserData> {
  try {
    const user = await fetchUser(id);
    const profile = await fetchUserProfile(user.id);
    const preferences = await fetchUserPreferences(user.id);

    return {
      user,
      profile,
      preferences,
    };
  } catch (error) {
    console.error("Failed to fetch user data:", error);
    throw error;
  }
}
```

```typescript
// Async generator function
async function* fetchPages<T>(url: string): AsyncGenerator<T[], void, unknown> {
  let page = 1;
  let hasMore = true;

  while (hasMore) {
    const response = await fetch(`${url}?page=${page}`);
    const data = await response.json();

    yield data.items;

    hasMore = data.hasMore;
    page++;
  }
}
```

Best Practices

✅ Good Practices

```typescript
// Always type function parameters
function processUser(user: User, options: ProcessOptions): ProcessResult {
  // Implementation
}

// Use specific return types for public APIs
function calculateTax(amount: number, rate: number): number {
  return amount * rate;
}

// Use function overloads for different behaviors
function format(value: string): string;
function format(value: number, decimals: number): string;
function format(value: string | number, decimals?: number): string {
  // Implementation
}

// Use generics for reusable functions
function createRepository<T>(entityType: new () => T): Repository<T> {
  return new Repository(entityType);
}
```

❌ Avoid

```typescript
// Don't use any for parameters
function badFunction(data: any): any {
  return data.whatever;
}
```

```
// Don't omit return types for public functions
function publicApi(input: string) {
  // Should specify return type
  return processInput(input);
}

// Don't use function overloads when union types suffice
function unnecessary(value: string): string;
function unnecessary(value: number): string;
// Better: function format(value: string | number): string
```

Function Type Checklist

- □ All function parameters have explicit types
- □ Public functions have explicit return types
- □ Use optional parameters instead of undefined unions when possible
- □ Leverage function overloads for different behaviors
- □ Use generics for reusable functions
- □ Implement proper error handling in async functions
- □ Use type guards for runtime type checking

Next Steps

Now that you understand function typing, let's explore arrays, tuples, and enums to handle collections and fixed sets of values.

Continue to: Arrays, Tuples, and Enums

Arrays, Tuples, and Enums

> Learn how to work with collections, fixed-length arrays, and enumerated values in TypeScript

Arrays in TypeScript

Basic Array Types

```
// Array type annotations
let numbers: number[] = [1, 2, 3, 4, 5];
let names: string[] = ["Alice", "Bob", "Charlie"];
let flags: boolean[] = [true, false, true];

// Alternative syntax using Array<T>
let scores: Array<number> = [95, 87, 92, 88];
let cities: Array<string> = ["New York", "London", "Tokyo"];

// Empty arrays
```

```
let emptyNumbers: number[] = [];
let emptyStrings: Array<string> = [];
```

Array Type Inference

```
// TypeScript infers array types
let fruits = ["apple", "banana", "orange"]; // string[]
let ages = [25, 30, 35, 40]; // number[]
let mixed = ["hello", 42, true]; // (string | number | boolean)[]

// Be careful with empty arrays
let empty = []; // any[] - not very useful
let typedEmpty: string[] = []; // Better - explicitly typed
```

Multi-dimensional Arrays

```
// 2D arrays
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];

// 3D arrays
let cube: number[][][] = [
  [
    [1, 2],
    [3, 4],
  ],
  [
    [5, 6],
    [7, 8],
  ],
];

// Array of objects
interface User {
  id: number;
  name: string;
  email: string;
}

let users: User[] = [
  { id: 1, name: "John", email: "john@example.com" },
  { id: 2, name: "Jane", email: "jane@example.com" },
];
```

Array Methods with Types

```
let numbers: number[] = [1, 2, 3, 4, 5];

// Map - transforms each element
let doubled: number[] = numbers.map((n) => n * 2); // [2, 4, 6, 8, 10]
let strings: string[] = numbers.map((n) => n.toString()); // ["1", "2", "3", "4", "5"]

// Filter - keeps elements that match condition
let evens: number[] = numbers.filter((n) => n % 2 === 0); // [2, 4]

// Reduce - combines all elements into single value
let sum: number = numbers.reduce((acc, n) => acc + n, 0); // 15
let product: number = numbers.reduce((acc, n) => acc * n, 1); // 120

// Find - returns first matching element or undefined
let found: number | undefined = numbers.find((n) => n > 3); // 4

// Some/Every - boolean checks
let hasEven: boolean = numbers.some((n) => n % 2 === 0); // true
let allPositive: boolean = numbers.every((n) => n > 0); // true
```

Generic Array Functions

```
// Generic function for arrays
function getFirst<T>(array: T[]): T | undefined {
  return array[0];
}

function getLast<T>(array: T[]): T | undefined {
  return array[array.length - 1];
}

function chunk<T>(array: T[], size: number): T[][] {
  const chunks: T[][] = [];
  for (let i = 0; i < array.length; i += size) {
    chunks.push(array.slice(i, i + size));
  }
  return chunks;
}

// Usage
const firstNumber = getFirst([1, 2, 3]); // number | undefined
const lastString = getLast(["a", "b", "c"]); // string | undefined
const numberChunks = chunk([1, 2, 3, 4, 5, 6], 2); // number[][]
```

Readonly Arrays

```
// Readonly array - cannot be modified
let readonlyNumbers: readonly number[] = [1, 2, 3, 4, 5];
// readonlyNumbers.push(6); // Error: Property 'push' does not exist
// readonlyNumbers[0] = 10; // Error: Index signature in type 'readonly number[]' only permits reading

// ReadonlyArray<T> type
let readonlyStrings: ReadonlyArray<string> = ["a", "b", "c"];

// Const assertions create readonly arrays
const colors = ["red", "green", "blue"] as const;
// Type: readonly ["red", "green", "blue"]

// Function with readonly parameter
function processNumbers(numbers: readonly number[]): number {
  return numbers.reduce((sum, n) => sum + n, 0);
}

processNumbers([1, 2, 3]); // OK
processNumbers(readonlyNumbers); // OK
```

Tuples

Tuples are arrays with fixed length and specific types for each position.

Basic Tuples

```
// Basic tuple types
let coordinate: [number, number] = [10, 20];
let person: [string, number] = ["John", 30];
let flag: [string, boolean] = ["isActive", true];

// Accessing tuple elements
let x = coordinate[0]; // number
let y = coordinate[1]; // number
let name = person[0]; // string
let age = person[1]; // number

// Tuple assignment
coordinate = [5, 15]; // OK
// coordinate = [5]; // Error: Type '[number]' is not assignable to type '[number, number]'
// coordinate = [5, 15, 25]; // Error: Type '[number, number, number]' is not assignable to type '[number, number]'
```

Optional Tuple Elements

```
// Optional elements (must be at the end)
let optionalTuple: [string, number?] = ["hello"];
optionalTuple = ["hello", 42]; // Also valid

// Multiple optional elements
let userInfo: [string, number?, boolean?] = ["John"];
userInfo = ["John", 30];
userInfo = ["John", 30, true];

// Function returning optional tuple
function parseCoordinate(input: string): [number, number] | [number] {
  const parts = input.split(",").map(Number);
  if (parts.length === 2) {
    return [parts[0], parts[1]];
  } else if (parts.length === 1) {
    return [parts[0]];
  }
  throw new Error("Invalid coordinate format");
}
```

Rest Elements in Tuples

```
// Rest elements
let restTuple: [string, ...number[]] = ["prefix", 1, 2, 3, 4];
let mixedRest: [boolean, ...string[], number] = [true, "a", "b", "c", 42];

// Spread in tuple types
type StringNumberTuple = [string, number];
type ExtendedTuple = [...StringNumberTuple, boolean]; // [string, number, boolean]

// Function with rest tuple parameter
function logWithNumbers(message: string, ...numbers: number[]): void {
  console.log(message, numbers);
}

logWithNumbers("Numbers:", 1, 2, 3, 4, 5);
```

Named Tuple Elements

```
// Named tuple elements (TypeScript 4.0+)
type Point3D = [x: number, y: number, z: number];
type RGB = [red: number, green: number, blue: number];
type UserRecord = [id: number, name: string, email: string, isActive: boolean];

// Usage remains the same, but provides better documentation
let point: Point3D = [10, 20, 30];
let color: RGB = [255, 128, 0];
let user: UserRecord = [1, "John", "john@example.com", true];
```

```
// Destructuring with named tuples
let [userId, userName, userEmail, userActive] = user;
```

Tuple Methods and Operations

```
let tuple: [string, number, boolean] = ["hello", 42, true];

// Length property
let length = tuple.length; // 3

// Destructuring
let [message, count, isEnabled] = tuple;

// Spread operator
let newTuple: [string, number, boolean, string] = [...tuple, "extra"];

// Array methods (that don't change length)
let found = tuple.find((item) => typeof item === "number"); // string | number |
boolean | undefined
let hasString = tuple.some((item) => typeof item === "string"); // boolean

// Converting to array
let asArray: (string | number | boolean)[] = [...tuple];
```

Enums

Enums allow you to define a set of named constants.

Numeric Enums

```
// Basic numeric enum
enum Direction {
  Up, // 0
  Down, // 1
  Left, // 2
  Right, // 3
}

// Custom numeric values
enum HttpStatus {
  OK = 200,
  NotFound = 404,
  InternalServerError = 500,
}

// Mixed numeric enum
enum MixedEnum {
  A, // 0
  B, // 1
  C = 10, // 10
  D, // 11
  E = 20, // 20
  F, // 21
}

// Using numeric enums
function move(direction: Direction): void {
  switch (direction) {
    case Direction.Up:
      console.log("Moving up");
      break;
    case Direction.Down:
      console.log("Moving down");
      break;
    case Direction.Left:
      console.log("Moving left");
      break;
    case Direction.Right:
      console.log("Moving right");
      break;
  }
}

move(Direction.Up);
move(0); // Also valid - numeric enums are bidirectional
```

String Enums

```
// String enum
enum Color {
  Red = "red",
  Green = "green",
  Blue = "blue",
  Yellow = "yellow",
}

// Theme enum
enum Theme {
  Light = "light",
  Dark = "dark",
  Auto = "auto",
}

// Using string enums
function setTheme(theme: Theme): void {
  document.body.className = theme;
}

setTheme(Theme.Dark);
```

```
// setTheme("dark"); // Error: Argument of type '"dark"' is not assignable to
parameter of type 'Theme'

// String enums are not bidirectional
console.log(Color.Red); // "red"
// console.log(Color["red"]); // Error: Element implicitly has an 'any' type
```

Const Enums

```
// Const enum - inlined at compile time
const enum Sizes {
  Small = "small",
  Medium = "medium",
  Large = "large",
}

// Usage
let size = Sizes.Medium; // Compiled to: let size = "medium";

// Benefits: No runtime overhead, smaller bundle size
// Drawbacks: Cannot be used with computed property access

// Regular enum vs const enum compilation:
// Regular: Creates an object at runtime
// Const: Replaces with literal values
```

Heterogeneous Enums

```
// Mixed string and numeric enum (not recommended)
enum BooleanLikeHeterogeneousEnum {
  No = 0,
  Yes = "YES",
}

// Better approach: Use union types
type Status = "pending" | "approved" | "rejected";
type Priority = 1 | 2 | 3 | 4 | 5;
```

Enum Utilities

```
enum UserRole {
  Admin = "admin",
  User = "user",
  Guest = "guest",
  Moderator = "moderator",
}
```

```
// Get all enum values
function getAllRoles(): UserRole[] {
  return Object.values(UserRole);
}

// Check if value is valid enum value
function isValidRole(value: string): value is UserRole {
  return Object.values(UserRole).includes(value as UserRole);
}

// Get enum keys
function getRoleKeys(): string[] {
  return Object.keys(UserRole);
}

// Usage
const roles = getAllRoles(); // ["admin", "user", "guest", "moderator"]
const isValid = isValidRole("admin"); // true
const keys = getRoleKeys(); // ["Admin", "User", "Guest", "Moderator"]
```

Reverse Mapping (Numeric Enums)

```
enum Status {
  Pending,
  Approved,
  Rejected,
}

// Numeric enums have reverse mapping
console.log(Status.Pending); // 0
console.log(Status[0]); // "Pending"
console.log(Status[Status.Pending]); // "Pending"

// Iterate over enum
for (let status in Status) {
  if (isNaN(Number(status))) {
    console.log(status); // "Pending", "Approved", "Rejected"
  }
}

// Get numeric values only
const numericValues = Object.keys(Status)
  .filter((key) => !isNaN(Number(key)))
  .map((key) => Number(key)); // [0, 1, 2]
```

Advanced Patterns

Array and Tuple Utilities

```
// Utility types for arrays and tuples
type Head<T extends readonly unknown[]> = T extends readonly [
  infer H,
  ...unknown[]
]
  ? H
  : never;
type Tail<T extends readonly unknown[]> = T extends readonly [
  unknown,
  ...infer T
]
  ? T
  : never;
type Length<T extends readonly unknown[]> = T["length"];

// Usage
type FirstElement = Head<[string, number, boolean]>; // string
type RestElements = Tail<[string, number, boolean]>; // [number, boolean]
type TupleLength = Length<[string, number, boolean]>; // 3

// Array manipulation utilities
function flatten<T>(arrays: T[][]): T[] {
  return arrays.reduce((acc, arr) => acc.concat(arr), []);
}

function unique<T>(array: T[]): T[] {
  return Array.from(new Set(array));
}

function groupBy<T, K extends keyof any>(
  array: T[],
  keyFn: (item: T) => K
): Record<K, T[]> {
  return array.reduce((groups, item) => {
    const key = keyFn(item);
    (groups[key] = groups[key] || []).push(item);
    return groups;
  }, {} as Record<K, T[]>);
}
```

Enum-like Patterns with Objects

```
// Object as enum alternative
const Theme = {
  Light: "light",
  Dark: "dark",
  Auto: "auto",
} as const;

type Theme = (typeof Theme)[keyof typeof Theme]; // "light" | "dark" | "auto"
```

```typescript
// Benefits: Tree-shakable, no runtime overhead
// Usage
function applyTheme(theme: Theme): void {
  document.body.setAttribute("data-theme", theme);
}

applyTheme(Theme.Dark);
```

Best Practices

✅ Good Practices

```typescript
// Use specific array types
const userIds: number[] = [1, 2, 3]; // Better than any[]

// Use readonly for immutable data
function processItems(items: readonly string[]): string[] {
  return items.map((item) => item.toUpperCase());
}

// Use tuples for fixed-structure data
type Coordinate = [x: number, y: number];
type RGB = [red: number, green: number, blue: number];

// Use string enums for better type safety
enum Status {
  Pending = "pending",
  Approved = "approved",
  Rejected = "rejected",
}

// Use const enums for performance when appropriate
const enum LogLevel {
  Debug = "debug",
  Info = "info",
  Warning = "warning",
  Error = "error",
}
```

❌ Avoid

```typescript
// Don't use any[] when you can be specific
const badArray: any[] = [1, "hello", true]; // Use union types instead

// Don't mutate readonly arrays
function badFunction(items: readonly string[]): void {
  // items.push("new"); // Error - good!
}
```

```
// Don't use heterogeneous enums
enum BadEnum {
  StringValue = "string",
  NumberValue = 42, // Confusing and error-prone
}

// Don't use numeric enums when string enums are clearer
enum BadStatus {
  Pending, // What does 0 mean?
  Approved,
  Rejected,
}
```

Summary Checklist

- □ Use specific array types instead of `any[]`
- □ Consider `readonly` for arrays that shouldn't be modified
- □ Use tuples for fixed-length, heterogeneous data
- □ Prefer string enums over numeric enums for clarity
- □ Use const enums for performance when tree-shaking isn't a concern
- □ Consider union types as alternatives to enums
- □ Use named tuple elements for better documentation

Next Steps

Now that you understand arrays, tuples, and enums, let's explore union and intersection types for more flexible type combinations.

Continue to: Union and Intersection Types

Union and Intersection Types

> Learn how to combine types using unions and intersections to create flexible and powerful type definitions

Union Types

Union types allow a value to be one of several types, using the `|` operator.

Basic Union Types

```
// Basic union types
type StringOrNumber = string | number;
type Status = "pending" | "approved" | "rejected";
type Theme = "light" | "dark" | "auto";
```

```
// Using union types
let id: StringOrNumber = "user123";
id = 456; // Also valid

let currentStatus: Status = "pending";
// currentStatus = "invalid"; // Error: Type '"invalid"' is not assignable

function setTheme(theme: Theme): void {
  document.body.setAttribute("data-theme", theme);
}

setTheme("dark"); // OK
// setTheme("blue"); // Error: Argument of type '"blue"' is not assignable
```

Union Types with Objects

```
// Union of object types
type Cat = {
  type: "cat";
  meow: () => void;
  purr: () => void;
};

type Dog = {
  type: "dog";
  bark: () => void;
  wag: () => void;
};

type Pet = Cat | Dog;

// Function accepting union type
function handlePet(pet: Pet): void {
  // Can only access common properties
  console.log(`Pet type: ${pet.type}`);

  // Need type narrowing for specific properties
  if (pet.type === "cat") {
    pet.meow(); // TypeScript knows this is a Cat
    pet.purr();
  } else {
    pet.bark(); // TypeScript knows this is a Dog
    pet.wag();
  }
}
```

Discriminated Unions

Discriminated unions use a common property to distinguish between union members.

```
// Discriminated union with literal types
type LoadingState = {
  status: "loading";
};

type SuccessState = {
  status: "success";
  data: any;
};

type ErrorState = {
  status: "error";
  error: string;
};

type ApiState = LoadingState | SuccessState | ErrorState;

// Type-safe handling of discriminated unions
function handleApiState(state: ApiState): void {
  switch (state.status) {
    case "loading":
      console.log("Loading...");
      break;
    case "success":
      console.log("Data:", state.data); // TypeScript knows state.data exists
      break;
    case "error":
      console.error("Error:", state.error); // TypeScript knows state.error exists
      break;
    default:
      // Exhaustiveness check
      const exhaustiveCheck: never = state;
      throw new Error(`Unhandled state: ${exhaustiveCheck}`);
  }
}
```

Complex Discriminated Unions

```
// Shape examples
type Circle = {
  kind: "circle";
  radius: number;
};

type Rectangle = {
  kind: "rectangle";
  width: number;
  height: number;
};
```

```
type Triangle = {
  kind: "triangle";
  base: number;
  height: number;
};

type Shape = Circle | Rectangle | Triangle;

// Calculate area with type safety
function calculateArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "rectangle":
      return shape.width * shape.height;
    case "triangle":
      return (shape.base * shape.height) / 2;
    default:
      const exhaustiveCheck: never = shape;
      throw new Error(`Unknown shape: ${exhaustiveCheck}`);
  }
}

// Usage
const circle: Circle = { kind: "circle", radius: 5 };
const rectangle: Rectangle = { kind: "rectangle", width: 10, height: 20 };

console.log(calculateArea(circle)); // 78.54
console.log(calculateArea(rectangle)); // 200
```

Union Types with Functions

```
// Function union types
type EventHandler = (() => void) | ((event: Event) => void);

function addEventListener(
  element: HTMLElement,
  eventType: string,
  handler: EventHandler
): void {
  if (handler.length === 0) {
    // Handler takes no parameters
    element.addEventListener(eventType, handler as () => void);
  } else {
    // Handler takes event parameter
    element.addEventListener(eventType, handler as (event: Event) => void);
  }
}

// Union of different function signatures
type Formatter =
  | ((value: string) => string)
  | ((value: number) => string)
  | ((value: boolean) => string);

const formatters: Formatter[] = [
  (value: string) => value.toUpperCase(),
  (value: number) => value.toFixed(2),
  (value: boolean) => (value ? "Yes" : "No"),
];
```

Type Narrowing with Union Types

Using typeof

```
function processValue(value: string | number | boolean): string {
  if (typeof value === "string") {
    return value.toUpperCase(); // TypeScript knows value is string
  } else if (typeof value === "number") {
    return value.toFixed(2); // TypeScript knows value is number
  } else {
    return value ? "true" : "false"; // TypeScript knows value is boolean
  }
}
```

Using instanceof

```
class Car {
  drive() {
    console.log("Driving car");
  }
}

class Bike {
  ride() {
    console.log("Riding bike");
  }
}

function useVehicle(vehicle: Car | Bike): void {
  if (vehicle instanceof Car) {
    vehicle.drive(); // TypeScript knows vehicle is Car
  } else {
    vehicle.ride(); // TypeScript knows vehicle is Bike
  }
}
```

Using 'in' operator

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

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

Custom Type Guards

```
// Custom type guard functions
function isString(value: unknown): value is string {
  return typeof value === "string";
}

function isNumber(value: unknown): value is number {
  return typeof value === "number";
}

function isUser(obj: any): obj is User {
  return obj && typeof obj.id === "number" && typeof obj.name === "string";
}

// Using custom type guards
function processUnknown(value: unknown): string {
  if (isString(value)) {
    return value.toUpperCase();
  } else if (isNumber(value)) {
    return value.toString();
  } else {
    return "Unknown type";
  }
}
```

Intersection Types

Intersection types combine multiple types into one, using the `&` operator.

Basic Intersection Types

```
// Basic intersection
type Name = {
  firstName: string;
  lastName: string;
};

type Contact = {
  email: string;
  phone: string;
};

type Person = Name & Contact;

// Person has all properties from both types
const person: Person = {
  firstName: "John",
  lastName: "Doe",
  email: "john@example.com",
  phone: "555-1234",
};
```

Intersection with Interfaces

```
interface Serializable {
  serialize(): string;
}

interface Loggable {
  log(): void;
}

// Intersection of interfaces
type SerializableLoggable = Serializable & Loggable;

class DataModel implements SerializableLoggable {
  constructor(private data: any) {}

  serialize(): string {
    return JSON.stringify(this.data);
  }

  log(): void {
    console.log(this.serialize());
  }
}
```

Intersection with Function Types

```
type EventEmitter = {
  on(event: string, callback: Function): void;
  emit(event: string, ...args: any[]): void;
};
```

```
type Disposable = {
  dispose(): void;
};

type EventEmitterWithDisposal = EventEmitter & Disposable;

class MyEventEmitter implements EventEmitterWithDisposal {
  private listeners: Map<string, Function[]> = new Map();

  on(event: string, callback: Function): void {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, []);
    }
    this.listeners.get(event)!.push(callback);
  }

  emit(event: string, ...args: any[]): void {
    const callbacks = this.listeners.get(event) || [];
    callbacks.forEach((callback) => callback(...args));
  }

  dispose(): void {
    this.listeners.clear();
  }
}
```

Merging Object Types

```
// Merging configuration objects
type DatabaseConfig = {
  host: string;
  port: number;
  database: string;
};

type AuthConfig = {
  username: string;
  password: string;
};

type SSLConfig = {
  ssl: boolean;
  cert?: string;
};

type FullConfig = DatabaseConfig & AuthConfig & SSLConfig;

const config: FullConfig = {
  host: "localhost",
  port: 5432,
  database: "myapp",
  username: "admin",
  password: "secret",
  ssl: true,
  cert: "/path/to/cert",
};
```

Advanced Union and Intersection Patterns

Conditional Types with Unions

```
// Conditional type based on union
type ApiResponse<T> = T extends string
  ? { message: T }
  : T extends number
  ? { code: T }
  : { data: T };

type StringResponse = ApiResponse<string>; // { message: string }
type NumberResponse = ApiResponse<number>; // { code: number }
type ObjectResponse = ApiResponse<User>; // { data: User }
```

Distributive Conditional Types

```
// Distributive conditional types with unions
type ToArray<T> = T extends any ? T[] : never;

type StringOrNumberArray = ToArray<string | number>;
// Distributes to: ToArray<string> | ToArray<number>
// Results in: string[] | number[]

// Non-distributive version
type ToArrayNonDistributive<T> = [T] extends [any] ? T[] : never;

type Combined = ToArrayNonDistributive<string | number>;
// Results in: (string | number)[]
```

Utility Types with Intersections

```
// Merge utility type
type Merge<T, U> = {
  [K in keyof T | keyof U]: K extends keyof U
    ? U[K]
    : K extends keyof T
    ? T[K]
    : never;
};
```

```
type A = { a: string; b: number };
type B = { b: string; c: boolean };
type Merged = Merge<A, B>; // { a: string; b: string; c: boolean }

// Override utility type
type Override<T, U> = Omit<T, keyof U> & U;

type Original = { id: number; name: string; email: string };
type Updates = { name: string; age: number };
type Updated = Override<Original, Updates>; // { id: number; email: string; name:
string; age: number }
```

Practical Examples

API Response Handling

```
// API response types
type ApiSuccess<T> = {
  success: true;
  data: T;
  message?: string;
};

type ApiError = {
  success: false;
  error: string;
  code: number;
};

type ApiResponse<T> = ApiSuccess<T> | ApiError;

// Type-safe response handling
function handleApiResponse<T>(response: ApiResponse<T>): T | null {
  if (response.success) {
    console.log("Success:", response.message);
    return response.data;
  } else {
    console.error(`Error ${response.code}: ${response.error}`);
    return null;
  }
}

// Usage
const userResponse: ApiResponse<User> = {
  success: true,
  data: { id: 1, name: "John", email: "john@example.com" },
};

const user = handleApiResponse(userResponse);
```

Form Validation

```typescript
// Validation result types
type ValidationSuccess = {
  valid: true;
  value: any;
};

type ValidationError = {
  valid: false;
  errors: string[];
};

type ValidationResult = ValidationSuccess | ValidationError;

// Validator functions
type Validator<T> = (value: T) => ValidationResult;

const emailValidator: Validator<string> = (email) => {
  const errors: string[] = [];

  if (!email.includes("@")) {
    errors.push("Email must contain @");
  }

  if (email.length < 5) {
    errors.push("Email must be at least 5 characters");
  }

  return errors.length === 0
    ? { valid: true, value: email }
    : { valid: false, errors };
};

// Combine validators
function combineValidators<T>(...validators: Validator<T>[]): Validator<T> {
  return (value: T) => {
    const allErrors: string[] = [];

    for (const validator of validators) {
      const result = validator(value);
      if (!result.valid) {
        allErrors.push(...result.errors);
      }
    }

    return allErrors.length === 0
      ? { valid: true, value }
      : { valid: false, errors: allErrors };
  };
}
```

Event System

```
// Event types
type ClickEvent = {
  type: "click";
  x: number;
  y: number;
  button: "left" | "right" | "middle";
};

type KeyEvent = {
  type: "keypress";
  key: string;
  ctrlKey: boolean;
  shiftKey: boolean;
};

type ResizeEvent = {
  type: "resize";
  width: number;
  height: number;
};

type AppEvent = ClickEvent | KeyEvent | ResizeEvent;

// Event handler types
type EventHandler<T extends AppEvent> = (event: T) => void;

// Type-safe event dispatcher
class EventDispatcher {
  private handlers: {
    click: EventHandler<ClickEvent>[];
    keypress: EventHandler<KeyEvent>[];
    resize: EventHandler<ResizeEvent>[];
  } = {
    click: [],
    keypress: [],
    resize: [],
  };

  on<T extends AppEvent["type"]>(
    eventType: T,
    handler: EventHandler<Extract<AppEvent, { type: T }>>
  ): void {
    this.handlers[eventType].push(handler as any);
  }

  emit<T extends AppEvent>(event: T): void {
    const handlers = this.handlers[event.type] as EventHandler<T>[];
    handlers.forEach((handler) => handler(event));
  }
}
```

Best Practices

✅ Good Practices

```typescript
// Use discriminated unions for complex state
type RequestState =
  | { status: "idle" }
  | { status: "loading" }
  | { status: "success"; data: any }
  | { status: "error"; error: string };

// Use intersection for composing types
type TimestampedUser = User & {
  createdAt: Date;
  updatedAt: Date;
};

// Use literal types in unions for better type safety
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";

// Use type guards for runtime type checking
function isErrorResponse(response: ApiResponse<any>): response is ApiError {
  return !response.success;
}
```

❌ Avoid

```typescript
// Don't use overly complex unions
type BadUnion = string | number | boolean | object | Function | symbol; // Too broad

// Don't forget exhaustiveness checking
function badHandler(state: RequestState): void {
  switch (state.status) {
    case "loading":
      // Handle loading
      break;
    case "success":
      // Handle success
      break;
    // Missing "idle" and "error" cases!
  }
}

// Don't use intersection with conflicting types
type Conflicting = { id: string } & { id: number }; // id becomes never
```

Summary Checklist

- □ Use union types for values that can be one of several types
- □ Use discriminated unions for complex state management
- □ Implement exhaustiveness checking with `never`
- □ Use intersection types to combine multiple type contracts
- □ Implement proper type narrowing with type guards
- □ Use literal types in unions for better type safety
- □ Avoid overly complex union types
- □ Consider using branded types for better type safety

Next Steps

Now that you understand union and intersection types, let's explore type inference and type narrowing techniques in more detail.

Continue to: Type Inference and Narrowing

Type Inference and Narrowing

> Master TypeScript's type inference capabilities and learn advanced type narrowing techniques

Type Inference Fundamentals

TypeScript can automatically infer types in many situations, reducing the need for explicit type annotations while maintaining type safety.

Basic Type Inference

```
// Variable type inference
let message = "Hello World"; // Inferred as string
let count = 42; // Inferred as number
let isActive = true; // Inferred as boolean
let items = [1, 2, 3]; // Inferred as number[]
let mixed = ["hello", 42, true]; // Inferred as (string | number | boolean)[]

// Object type inference
let user = {
  id: 1,
  name: "John",
  email: "john@example.com",
}; // Inferred as { id: number; name: string; email: string; }

// Function return type inference
function add(a: number, b: number) {
  return a + b; // Return type inferred as number
}
```

```
function getUser() {
  return {
    id: 1,
    name: "John",
    isActive: true,
  }; // Return type inferred as { id: number; name: string; isActive: boolean; }
}
```

Contextual Type Inference

```
// Array method callbacks
const numbers = [1, 2, 3, 4, 5];

// TypeScript infers parameter types from context
const doubled = numbers.map((n) => n * 2); // n is inferred as number
const filtered = numbers.filter((n) => n > 2); // n is inferred as number
const sum = numbers.reduce((acc, n) => acc + n, 0); // acc and n inferred as number

// Event handlers
const button = document.querySelector("button");
button?.addEventListener("click", (event) => {
  // event is inferred as MouseEvent
  console.log(event.clientX, event.clientY);
});

// Promise chains
fetch("/api/users")
  .then((response) => response.json()) // response inferred as Response
  .then((data) => console.log(data)); // data inferred as any (from json())
```

Best Common Type Inference

```
// TypeScript finds the best common type
let mixed = [1, "hello", true]; // (string | number | boolean)[]
let numbers = [1, 2, 3.14]; // number[]

// With objects
let animals = [
  { name: "Fluffy", type: "cat" },
  { name: "Buddy", type: "dog" },
  { name: "Tweety", type: "bird" },
]; // { name: string; type: string; }[]

// When no common type exists
class Cat {
  meow() {}
}
class Dog {
  bark() {}
}

let pets = [new Cat(), new Dog()]; // (Cat | Dog)[]
```

Generic Type Inference

```
// Generic function with inference
function identity<T>(arg: T): T {
  return arg;
}

// Type parameter inferred from argument
const stringResult = identity("hello"); // T inferred as string
const numberResult = identity(42); // T inferred as number

// Multiple type parameters
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}

const stringNumberPair = pair("hello", 42); // [string, number]

// Generic constraints with inference
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const user = { id: 1, name: "John", email: "john@example.com" };
const userName = getProperty(user, "name"); // string
const userId = getProperty(user, "id"); // number
```

Advanced Type Inference

Conditional Type Inference

```
// Infer keyword in conditional types
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

type StringFunction = () => string;
type NumberFunction = () => number;

type StringReturn = ReturnType<StringFunction>; // string
type NumberReturn = ReturnType<NumberFunction>; // number

// Infer array element type
type ElementType<T> = T extends (infer U)[] ? U : never;

type StringArrayElement = ElementType<string[]>; // string
type NumberArrayElement = ElementType<number[]>; // number

// Infer promise value type
type PromiseValue<T> = T extends Promise<infer U> ? U : never;

type StringPromiseValue = PromiseValue<Promise<string>>; // string
type UserPromiseValue = PromiseValue<Promise<User>>; // User
```

Template Literal Type Inference

```
// Template literal type inference
type EventName<T extends string> = `on${Capitalize<T>}`;

type ClickEvent = EventName<"click">; // "onClick"
type HoverEvent = EventName<"hover">; // "onHover"

// Extract parts from template literals
type ExtractEventType<T> = T extends `on${infer U}` ? Lowercase<U> : never;

type ClickType = ExtractEventType<"onClick">; // "click"
type HoverType = ExtractEventType<"onHover">; // "hover"

// Complex template literal inference
type ParseRoute<T extends string> = T extends `/${infer Segment}/${infer Rest}`
  ? [Segment, ...ParseRoute<`/${Rest}`>]
  : T extends `/${infer Segment}`
  ? [Segment]
  : [];

type UserRoute = ParseRoute<"/users/123/profile">; // ["users", "123", "profile"]
```

Mapped Type Inference

```
// Infer from mapped types
type GetValueType<T> = T extends { [K in keyof T]: infer U } ? U : never;

type StringRecord = { a: string; b: string; c: string };
type StringType = GetValueType<StringRecord>; // string

type MixedRecord = { a: string; b: number; c: boolean };
type MixedType = GetValueType<MixedRecord>; // string | number | boolean

// Infer function parameter types
type Parameters<T extends (...args: any) => any> = T extends (
  ...args: infer P
) => any
  ? P
  : never;
```

```
type AddFunction = (a: number, b: number) => number;
type AddParams = Parameters<AddFunction>; // [number, number]
```

Type Narrowing Techniques

Type narrowing helps TypeScript understand more specific types within conditional blocks.

typeof Type Guards

```
function processValue(value: string | number | boolean): string {
  if (typeof value === "string") {
    // TypeScript knows value is string here
    return value.toUpperCase();
  } else if (typeof value === "number") {
    // TypeScript knows value is number here
    return value.toFixed(2);
  } else {
    // TypeScript knows value is boolean here
    return value ? "true" : "false";
  }
}

// Typeof with objects
function handleInput(input: string | string[] | null): string {
  if (typeof input === "string") {
    return input;
  } else if (typeof input === "object" && input !== null) {
    // TypeScript knows input is string[] here
    return input.join(", ");
  } else {
    // TypeScript knows input is null here
    return "No input";
  }
}
```

instanceof Type Guards

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

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

```
function handlePet(pet: Dog | Cat): void {
  if (pet instanceof Dog) {
    pet.bark(); // TypeScript knows pet is Dog
  } else {
    pet.meow(); // TypeScript knows pet is Cat
  }
}

// instanceof with built-in types
function processError(error: Error | string): string {
  if (error instanceof Error) {
    return error.message; // TypeScript knows error is Error
  } else {
    return error; // TypeScript knows error is string
  }
}
```

in Operator Type Guards

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

function move(creature: Fish | Bird | Human): void {
  if ("swim" in creature) {
    creature.swim(); // TypeScript knows creature is Fish
  } else if ("fly" in creature) {
    creature.fly(); // TypeScript knows creature is Bird
  } else {
    creature.walk(); // TypeScript knows creature is Human
  }
}

// in operator with optional properties
interface User {
  id: number;
  name: string;
  email?: string;
}

function processUser(user: User): void {
  if ("email" in user && user.email) {
    // TypeScript knows user.email exists and is not undefined
    console.log(`Email: ${user.email}`);
  }
}
```

Custom Type Guards

```typescript
// Basic type guard
function isString(value: unknown): value is string {
  return typeof value === "string";
}

function isNumber(value: unknown): value is number {
  return typeof value === "number";
}

// Object type guard
interface User {
  id: number;
  name: string;
  email: string;
}

function isUser(obj: any): obj is User {
  return (
    obj &&
    typeof obj.id === "number" &&
    typeof obj.name === "string" &&
    typeof obj.email === "string"
  );
}

// Generic type guard
function isArrayOf<T>(
  value: unknown,
  guard: (item: unknown) => item is T
): value is T[] {
  return Array.isArray(value) && value.every(guard);
}

// Usage
function processUnknownValue(value: unknown): void {
  if (isString(value)) {
    console.log(value.toUpperCase()); // TypeScript knows value is string
  } else if (isNumber(value)) {
    console.log(value.toFixed(2)); // TypeScript knows value is number
  } else if (isUser(value)) {
    console.log(`User: ${value.name}`); // TypeScript knows value is User
  }
}

const maybeUsers: unknown = [
  { id: 1, name: "John", email: "john@example.com" },
  { id: 2, name: "Jane", email: "jane@example.com" },
];

if (isArrayOf(maybeUsers, isUser)) {
  // TypeScript knows maybeUsers is User[]
  maybeUsers.forEach((user) => console.log(user.name));
}
```

Discriminated Union Narrowing

```typescript
// Discriminated unions with literal types
type LoadingState = { status: "loading" };
type SuccessState = { status: "success"; data: any };
type ErrorState = { status: "error"; error: string };

type AsyncState = LoadingState | SuccessState | ErrorState;

function handleState(state: AsyncState): void {
  switch (state.status) {
    case "loading":
      console.log("Loading...");
      break;
    case "success":
      console.log("Data:", state.data); // TypeScript knows state is SuccessState
      break;
    case "error":
      console.error("Error:", state.error); // TypeScript knows state is ErrorState
      break;
    default:
      // Exhaustiveness check
      const exhaustiveCheck: never = state;
      throw new Error(`Unhandled state: ${exhaustiveCheck}`);
  }
}

// Complex discriminated unions
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "rectangle"; width: number; height: number }
  | { kind: "triangle"; base: number; height: number };

function calculateArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2; // TypeScript knows shape has radius
    case "rectangle":
      return shape.width * shape.height; // TypeScript knows shape has width and height
    case "triangle":
      return (shape.base * shape.height) / 2; // TypeScript knows shape has base and height
    default:
      const exhaustiveCheck: never = shape;
      throw new Error(`Unknown shape: ${exhaustiveCheck}`);
  }
}
```

Truthiness Narrowing

```
// Truthiness narrowing
function processOptionalString(str: string | null | undefined): string {
  if (str) {
    // TypeScript knows str is string (not null or undefined)
    return str.toUpperCase();
  } else {
    return "No string provided";
  }
}

// Array length narrowing
function processArray(arr: string[]): string {
  if (arr.length > 0) {
    // TypeScript knows arr is not empty
    return arr[0].toUpperCase(); // Safe to access first element
  } else {
    return "Empty array";
  }
}

// Object property narrowing
interface Config {
  apiUrl?: string;
  timeout?: number;
}

function makeRequest(config: Config): void {
  if (config.apiUrl) {
    // TypeScript knows config.apiUrl is string (not undefined)
    fetch(config.apiUrl);
  }
}
```

Equality Narrowing

```
// Equality narrowing with literals
function handleStatus(
  status: "pending" | "approved" | "rejected" | null
): void {
  if (status === "approved") {
    // TypeScript knows status is "approved"
    console.log("Request approved");
  } else if (status === null) {
    // TypeScript knows status is null
    console.log("No status");
  } else {
    // TypeScript knows status is "pending" | "rejected"
    console.log(`Status: ${status}`);
  }
}

// Equality narrowing with discriminated unions
type ApiResponse =
  | { success: true; data: any }
  | { success: false; error: string };

function handleResponse(response: ApiResponse): void {
  if (response.success === true) {
    console.log("Data:", response.data); // TypeScript knows response has data
  } else {
    console.error("Error:", response.error); // TypeScript knows response has error
  }
}
```

Advanced Narrowing Patterns

Control Flow Analysis

```
// TypeScript tracks control flow
function processValue(value: string | number | null): string {
  if (value === null) {
    return "null value";
  }

  // TypeScript knows value is string | number here
  if (typeof value === "string") {
    return value.toUpperCase();
  }

  // TypeScript knows value is number here
  return value.toString();
}

// Early returns
function validateUser(user: any): User {
  if (!user) {
    throw new Error("User is required");
  }

  if (typeof user.id !== "number") {
    throw new Error("User ID must be a number");
  }

  if (typeof user.name !== "string") {
    throw new Error("User name must be a string");
  }

  // TypeScript knows user has the right shape here
  return user as User;
}
```

Assertion Functions

```
// Assertion functions
function assert(condition: any, message?: string): asserts condition {
  if (!condition) {
    throw new Error(message || "Assertion failed");
  }
}

function assertIsString(value: unknown): asserts value is string {
  if (typeof value !== "string") {
    throw new Error("Expected string");
  }
}

function assertIsUser(obj: any): asserts obj is User {
  assert(obj && typeof obj === "object", "Expected object");
  assert(typeof obj.id === "number", "Expected numeric id");
  assert(typeof obj.name === "string", "Expected string name");
  assert(typeof obj.email === "string", "Expected string email");
}

// Usage
function processUnknown(value: unknown): void {
  assertIsString(value);
  // TypeScript knows value is string after assertion
  console.log(value.toUpperCase());
}

function processUserData(data: any): void {
  assertIsUser(data);
  // TypeScript knows data is User after assertion
  console.log(`User: ${data.name} (${data.email})`);
}
```

Never Type for Exhaustiveness

```
// Exhaustiveness checking with never
type Action = { type: "increment" } | { type: "decrement" } | { type: "reset" };

function reducer(state: number, action: Action): number {
  switch (action.type) {
    case "increment":
      return state + 1;
    case "decrement":
      return state - 1;
    case "reset":
      return 0;
    default:
      // This ensures all cases are handled
      const exhaustiveCheck: never = action;
      throw new Error(`Unhandled action: ${exhaustiveCheck}`);
  }
}

// If you add a new action type, TypeScript will error
// type Action =
//   | { type: "increment" }
//   | { type: "decrement" }
//   | { type: "reset" }
//   | { type: "multiply"; factor: number }; // New action

// The default case will now error because action is not never
```

Best Practices

✅ Good Practices

```
// Let TypeScript infer when obvious
const users = [
  { id: 1, name: "John" },
  { id: 2, name: "Jane" },
]; // Let TypeScript infer the array type

// Use type guards for runtime safety
function isValidEmail(email: unknown): email is string {
  return typeof email === "string" && email.includes("@");
}

// Use assertion functions for validation
function assertIsPositive(value: number): asserts value is number {
  if (value <= 0) {
    throw new Error("Value must be positive");
  }
}

// Use discriminated unions for state management
type RequestState =
  | { status: "idle" }
  | { status: "loading" }
  | { status: "success"; data: any }
  | { status: "error"; error: string };
```

❌ Avoid

```
// Don't over-annotate when inference works
const message: string = "Hello"; // Unnecessary annotation
const message = "Hello"; // Better

// Don't use any when you can narrow
function badProcess(value: any): any {
  return value.whatever; // No type safety
}

// Don't ignore exhaustiveness checking
function badReducer(action: Action): number {
  switch (action.type) {
    case "increment":
      return 1;
    // Missing other cases - no compile-time error
  }
  return 0; // Fallback hides missing cases
}
```

Summary Checklist

- □ Leverage TypeScript's type inference when types are obvious
- □ Use type guards for runtime type checking
- □ Implement custom type guards for complex objects
- □ Use discriminated unions for state management
- □ Implement exhaustiveness checking with `never`
- □ Use assertion functions for validation
- □ Understand control flow analysis
- □ Use the `in` operator for property checking
- □ Implement proper error handling with type narrowing

Next Steps

Now that you understand type inference and narrowing, let's move on to intermediate topics starting with optional and readonly properties.

Continue to: Optional and Readonly Properties

Optional and Readonly Properties

> Learn how to work with optional properties, readonly modifiers, and default parameters in TypeScript

Optional Properties

Optional properties allow you to define object types where some properties may or may not be present.

Basic Optional Properties

```
// Interface with optional properties
interface User {
  id: number;
  name: string;
  email: string;
  avatar?: string; // Optional property
  bio?: string;
  website?: string;
  preferences?: {
    theme: "light" | "dark";
    notifications: boolean;
  };
}

// Valid objects - optional properties can be omitted
const user1: User = {
  id: 1,
  name: "John Doe",
  email: "john@example.com",
};

// Also valid - optional properties included
const user2: User = {
  id: 2,
  name: "Jane Smith",
  email: "jane@example.com",
  avatar: "avatar.jpg",
  bio: "Software developer",
  preferences: {
    theme: "dark",
    notifications: true,
  },
};
```

Optional Properties in Type Aliases

```
// Type alias with optional properties
type Product = {
  id: number;
  name: string;
  price: number;
  description?: string;
  category?: string;
  tags?: string[];
  inStock?: boolean;
};

// Configuration object with optional properties
type ApiConfig = {
  baseUrl: string;
  timeout?: number;
  retries?: number;
  headers?: Record<string, string>;
  debug?: boolean;
};

function createApiClient(config: ApiConfig) {
  const defaultConfig = {
    timeout: 5000,
    retries: 3,
    debug: false,
    ...config,
  };

  return new ApiClient(defaultConfig);
}
```

Working with Optional Properties

```
interface UserProfile {
  id: number;
  username: string;
  email: string;
  firstName?: string;
  lastName?: string;
  avatar?: string;
  socialLinks?: {
    twitter?: string;
    github?: string;
    linkedin?: string;
  };
}

// Safe access to optional properties
function displayUserInfo(user: UserProfile): string {
  let info = `${user.username} (${user.email})`;

  // Check if optional property exists
  if (user.firstName && user.lastName) {
    info += ` - ${user.firstName} ${user.lastName}`;
  }

  // Optional chaining (TypeScript 3.7+)
  const twitterHandle = user.socialLinks?.twitter;
  if (twitterHandle) {
    info += ` | Twitter: @${twitterHandle}`;
  }

  return info;
}
```

```
// Nullish coalescing with optional properties
function getUserDisplayName(user: UserProfile): string {
  return user.firstName ?? user.username ?? "Anonymous";
}

function getAvatarUrl(user: UserProfile): string {
  return user.avatar ?? "/default-avatar.png";
}
```

Optional Properties vs Union with Undefined

```
// Optional property
interface WithOptional {
  name: string;
  age?: number; // Can be omitted or undefined
}

// Union with undefined
interface WithUnion {
  name: string;
  age: number | undefined; // Must be present, but can be undefined
}

// Usage differences
const optional1: WithOptional = { name: "John" }; // Valid
const optional2: WithOptional = { name: "John", age: undefined }; // Valid

const union1: WithUnion = { name: "John", age: undefined }; // Valid
// const union2: WithUnion = { name: "John" }; // Error: Property 'age' is missing

// When to use each:
// Optional (?): When the property might not be relevant
// Union with undefined: When the property is always relevant but might not have a
value
```

Readonly Properties

Readonly properties cannot be modified after object creation.

Basic Readonly Properties

```
interface ImmutableUser {
  readonly id: number;
  readonly createdAt: Date;
  name: string; // Can be modified
  email: string;
}

const user: ImmutableUser = {
  id: 1,
  createdAt: new Date(),
  name: "John Doe",
  email: "john@example.com",
};

// user.id = 2; // Error: Cannot assign to 'id' because it is a read-only property
// user.createdAt = new Date(); // Error: Cannot assign to 'createdAt'
user.name = "Jane Doe"; // OK: name is not readonly
user.email = "jane@example.com"; // OK: email is not readonly
```

Readonly Arrays and Tuples

```
// Readonly array
interface Config {
  readonly supportedLanguages: readonly string[];
  readonly coordinates: readonly [number, number];
}

const appConfig: Config = {
  supportedLanguages: ["en", "es", "fr"],
  coordinates: [40.7128, -74.006],
};

// appConfig.supportedLanguages.push("de"); // Error: Property 'push' does not
exist
// appConfig.coordinates[0] = 50; // Error: Index signature only permits reading

// ReadonlyArray<T> type
function processItems(items: ReadonlyArray<string>): string {
  return items.join(", "); // OK: reading is allowed
  // items.push("new"); // Error: Property 'push' does not exist
}

// Readonly tuple
type Point = readonly [number, number];
const origin: Point = [0, 0];
// origin[0] = 1; // Error: Index signature only permits reading
```

Readonly Utility Type

```
// Readonly<T> utility type makes all properties readonly
interface MutableUser {
  id: number;
  name: string;
  email: string;
  preferences: {
    theme: string;
    notifications: boolean;
  };
}

type ImmutableUser = Readonly<MutableUser>;
// Equivalent to:
// {
//   readonly id: number;
//   readonly name: string;
//   readonly email: string;
//   readonly preferences: {
//     theme: string;
//     notifications: boolean;
//   };
// }

// Note: Readonly is shallow - nested objects are not made readonly
const user: ImmutableUser = {
  id: 1,
  name: "John",
  email: "john@example.com",
  preferences: {
    theme: "dark",
    notifications: true,
  },
};

// user.name = "Jane"; // Error: readonly
user.preferences.theme = "light"; // OK: nested object is not readonly
```

Deep Readonly

```
// Deep readonly utility type
type DeepReadonly<T> = {
  readonly [P in keyof T]: T[P] extends object ? DeepReadonly<T[P]> : T[P];
};

type DeepImmutableUser = DeepReadonly<MutableUser>;

const deepUser: DeepImmutableUser = {
  id: 1,
  name: "John",
  email: "john@example.com",
  preferences: {
    theme: "dark",
    notifications: true,
  },
};

// deepUser.name = "Jane"; // Error: readonly
// deepUser.preferences.theme = "light"; // Error: readonly (deep)
// Alternative using const assertions
const constUser = {
  id: 1,
  name: "John",
  email: "john@example.com",
  preferences: {
    theme: "dark",
    notifications: true,
  },
} as const;
// All properties become readonly and literal types
```

Optional Function Parameters

Basic Optional Parameters

```
// Optional parameters must come after required ones
function greet(name: string, greeting?: string): string {
  return `${greeting || "Hello"}, ${name}!`;
}

greet("John"); // "Hello, John!"
greet("John", "Hi"); // "Hi, John!"

// Multiple optional parameters
function createUser(
  name: string,
  age?: number,
  email?: string,
  isActive?: boolean
): User {
  return {
    id: Math.random(),
    name,
    age: age ?? 0,
    email: email ?? "",
    isActive: isActive ?? true,
  };
}
```

Default Parameters

```
// Default parameter values
function createConnection(
  host: string = "localhost",
  port: number = 3000,
  ssl: boolean = false
): Connection {
  return new Connection(host, port, ssl);
}

// Can call with any number of arguments
createConnection(); // Uses all defaults
createConnection("api.example.com"); // Custom host, default port and ssl
createConnection("api.example.com", 443, true); // All custom

// Default parameters can reference earlier parameters
function buildUrl(
  protocol: string = "https",
  host: string,
  path: string = "/",
  port?: number
): string {
  const portSuffix = port ? `:${port}` : "";
  return `${protocol}://${host}${portSuffix}${path}`;
}
```

Optional vs Default Parameters

```
// Optional parameter
function optionalParam(name: string, age?: number): void {
  console.log(`Name: ${name}, Age: ${age ?? "unknown"}`);
}

// Default parameter
function defaultParam(name: string, age: number = 0): void {
  console.log(`Name: ${name}, Age: ${age}`);
}

// Usage
optionalParam("John"); // age is undefined
defaultParam("John"); // age is 0

// Explicit undefined
optionalParam("John", undefined); // OK
defaultParam("John", undefined); // age becomes 0 (default applied)
```

Object Destructuring with Optional Properties

Destructuring Optional Properties

```
interface ApiOptions {
  baseUrl: string;
  timeout?: number;
  retries?: number;
  headers?: Record<string, string>;
}
```

```
// Destructuring with defaults
function makeRequest({
  baseUrl,
  timeout = 5000,
  retries = 3,
  headers = {},
}: ApiOptions): Promise<Response> {
  // Implementation
  return fetch(baseUrl, {
    headers,
    signal: AbortSignal.timeout(timeout),
  });
}

// Destructuring with optional object parameter
function createApiClient({
  baseUrl = "https://api.example.com",
  timeout = 5000,
  retries = 3,
}: ApiOptions = {}): ApiClient {
  return new ApiClient(baseUrl, timeout, retries);
}

// Can be called without arguments
const client1 = createApiClient(); // Uses all defaults
const client2 = createApiClient({ baseUrl: "https://custom.api.com" });
```

Nested Destructuring

```
interface UserSettings {
  profile: {
    name: string;
    avatar?: string;
  };
  preferences?: {
    theme?: "light" | "dark";
    notifications?: boolean;
    language?: string;
  };
}

function updateUserSettings({
  profile: { name, avatar = "/default-avatar.png" },
  preferences: { theme = "light", notifications = true, language = "en" } = {},
}: UserSettings): void {
  console.log({
    name,
    avatar,
    theme,
    notifications,
    language,
  });
}
```

Practical Examples

Configuration Objects

```
interface DatabaseConfig {
  readonly host: string;
  readonly port: number;
  readonly database: string;
  username?: string;
  password?: string;
  ssl?: boolean;
  connectionTimeout?: number;
  maxConnections?: number;
  readonly createdAt: Date;
}

function createDatabaseConnection(config: DatabaseConfig): DatabaseConnection {
  // Validate required readonly properties
  if (!config.host || !config.port || !config.database) {
    throw new Error("Host, port, and database are required");
  }

  const connectionConfig = {
    host: config.host,
    port: config.port,
    database: config.database,
    username: config.username ?? "guest",
    password: config.password ?? "",
    ssl: config.ssl ?? false,
    connectionTimeout: config.connectionTimeout ?? 30000,
    maxConnections: config.maxConnections ?? 10,
  };

  return new DatabaseConnection(connectionConfig);
}
```

API Response Types

```
interface ApiResponse<T> {
  readonly success: boolean;
  readonly timestamp: Date;
  readonly requestId: string;
  data?: T;
  error?: {
    code: string;
    message: string;
    details?: Record<string, any>;
  };
  pagination?: {
    page: number;
    limit: number;
    total: number;
    hasNext: boolean;
  };
}

function handleApiResponse<T>(response: ApiResponse<T>): T | null {
  if (response.success && response.data) {
    return response.data;
  }

  if (response.error) {
    console.error(
      `API Error ${response.error.code}: ${response.error.message}`
    );
    if (response.error.details) {
      console.error("Details:", response.error.details);
    }
  }

  return null;
}
```

Form Validation

```
interface FormData {
  email: string;
  password: string;
  confirmPassword?: string;
  firstName?: string;
  lastName?: string;
  agreeToTerms: boolean;
  newsletter?: boolean;
}

interface ValidationResult {
  readonly isValid: boolean;
  readonly errors: readonly string[];
  readonly warnings?: readonly string[];
}

function validateForm(data: FormData): ValidationResult {
  const errors: string[] = [];
  const warnings: string[] = [];

  // Required field validation
  if (!data.email.includes("@")) {
    errors.push("Invalid email format");
  }

  if (data.password.length < 8) {
    errors.push("Password must be at least 8 characters");
  }

  // Optional field validation
  if (data.confirmPassword && data.confirmPassword !== data.password) {
    errors.push("Passwords do not match");
  }

  if (!data.firstName && !data.lastName) {
    warnings.push("Consider providing your name for better experience");
  }

  return {
    isValid: errors.length === 0,
    errors,
    warnings: warnings.length > 0 ? warnings : undefined,
  };
}
```

Best Practices

✅ Good Practices

```
// Use optional properties for truly optional data
interface User {
  id: number;
  name: string;
  email: string;
  avatar?: string; // Truly optional
  lastLoginAt?: Date; // May not exist for new users
}

// Use readonly for immutable data
interface Event {
  readonly id: string;
  readonly timestamp: Date;
  readonly type: string;
  data: any; // Can be modified
}

// Provide sensible defaults
function createApiClient({
  baseUrl,
  timeout = 5000, // Reasonable default
  retries = 3,
  debug = false,
}: {
  baseUrl: string;
  timeout?: number;
  retries?: number;
  debug?: boolean;
}): ApiClient {
  return new ApiClient(baseUrl, timeout, retries, debug);
}

// Use const assertions for immutable data
const config = {
  apiUrl: "https://api.example.com",
  version: "1.0.0",
  features: ["auth", "analytics"],
} as const;
```

❌ Avoid

```
// Don't make everything optional
interface BadUser {
  id?: number; // ID should be required
  name?: string; // Name should be required
  email?: string; // Email should be required
}

// Don't use readonly for data that needs to change
interface BadCounter {
  readonly count: number; // Should be mutable
  readonly increment: () => void; // Methods don't need readonly
}

// Don't use optional when undefined union is clearer
interface ConfusingApi {
  data?: any; // Is this missing data or undefined data?
}

// Better:
interface ClearApi {
  data: any | null; // Explicitly nullable
}
```

Summary Checklist

- □ Use optional properties (`?`) for truly optional data
- □ Use readonly for immutable properties
- □ Understand the difference between optional and undefined union
- □ Use default parameters for function arguments
- □ Use `Readonly<T>` utility type when needed
- □ Consider deep readonly for nested immutability
- □ Use const assertions for compile-time immutability

- □ Provide sensible defaults in destructuring
- □ Use optional chaining (`?.`) for safe property access
- □ Use nullish coalescing (`??`) for default values

Next Steps

Now that you understand optional and readonly properties, let's explore classes, access modifiers, and inheritance in TypeScript.

Continue to: Classes and Object-Oriented Programming

Classes and Object-Oriented Programming

> Master TypeScript classes, access modifiers, inheritance, and object-oriented programming patterns

Basic Classes

Class Declaration and Constructor

```typescript
// Basic class with constructor
class User {
  // Properties
  id: number;
  name: string;
  email: string;

  // Constructor
  constructor(id: number, name: string, email: string) {
    this.id = id;
    this.name = name;
    this.email = email;
  }

  // Methods
  getDisplayName(): string {
    return `${this.name} (${this.email})`;
  }

  updateEmail(newEmail: string): void {
    this.email = newEmail;
  }
}

// Creating instances
const user1 = new User(1, "John Doe", "john@example.com");
const user2 = new User(2, "Jane Smith", "jane@example.com");

console.log(user1.getDisplayName()); // "John Doe (john@example.com)"
user1.updateEmail("john.doe@example.com");
```

Property Initialization

```
// Property initialization in declaration
class Counter {
  count: number = 0; // Default value
  step: number = 1;

  constructor(initialCount?: number, step?: number) {
    if (initialCount !== undefined) {
      this.count = initialCount;
    }
    if (step !== undefined) {
      this.step = step;
    }
  }

  increment(): void {
    this.count += this.step;
  }

  decrement(): void {
    this.count -= this.step;
  }

  reset(): void {
    this.count = 0;
  }
}

// Shorthand constructor parameter properties
class Product {
  constructor(
    public id: number,
    public name: string,
    public price: number,
    public category: string = "general"
  ) {
    // Properties are automatically created and assigned
  }

  getFormattedPrice(): string {
    return `$${this.price.toFixed(2)}`;
  }
}

const product = new Product(1, "Laptop", 999.99, "electronics");
console.log(product.name); // "Laptop"
console.log(product.getFormattedPrice()); // "$999.99"
```

Access Modifiers

Public, Private, and Protected

```typescript
class BankAccount {
  public accountNumber: string; // Accessible everywhere
  private balance: number; // Only accessible within this class
  protected accountType: string; // Accessible in this class and subclasses

  constructor(
    accountNumber: string,
    initialBalance: number,
    accountType: string
  ) {
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
    this.accountType = accountType;
  }

  // Public method
  public getBalance(): number {
    return this.balance;
  }

  // Public method
  public deposit(amount: number): void {
    if (amount > 0) {
      this.balance += amount;
    }
  }

  // Public method
  public withdraw(amount: number): boolean {
    if (this.canWithdraw(amount)) {
      this.balance -= amount;
      return true;
    }
    return false;
  }

  // Private method - only accessible within this class
  private canWithdraw(amount: number): boolean {
    return amount > 0 && amount <= this.balance;
  }

  // Protected method - accessible in subclasses
  protected getAccountInfo(): string {
    return `${this.accountType} Account: ${this.accountNumber}`;
  }
}

const account = new BankAccount("123456789", 1000, "Checking");
console.log(account.accountNumber); // OK: public
console.log(account.getBalance()); // OK: public method
```

```
// console.log(account.balance); // Error: private
// account.canWithdraw(100); // Error: private method
```

Private Fields (ES2022)

```
// Using # for truly private fields (runtime private)
class SecureUser {
  #password: string; // Truly private
  #salt: string;
  public username: string;

  constructor(username: string, password: string) {
    this.username = username;
    this.#salt = this.generateSalt();
    this.#password = this.hashPassword(password);
  }

  #generateSalt(): string {
    return Math.random().toString(36).substring(2);
  }

  #hashPassword(password: string): string {
    return `${password}_${this.#salt}_hashed`;
  }

  public verifyPassword(password: string): boolean {
    return this.#hashPassword(password) === this.#password;
  }

  public changePassword(oldPassword: string, newPassword: string): boolean {
    if (this.verifyPassword(oldPassword)) {
      this.#password = this.hashPassword(newPassword);
      return true;
    }
    return false;
  }
}

const user = new SecureUser("john", "secret123");
console.log(user.username); // OK
// console.log(user.#password); // SyntaxError: Private field '#password' must be declared in an enclosing class
```

Getters and Setters

Property Accessors

```
class Temperature {
  private _celsius: number = 0;
  constructor(celsius: number) {
    this.celsius = celsius; // Uses setter for validation
  }

  // Getter
  get celsius(): number {
    return this._celsius;
  }

  // Setter with validation
  set celsius(value: number) {
    if (value < -273.15) {
      throw new Error("Temperature cannot be below absolute zero");
    }
    this._celsius = value;
  }

  // Computed property
  get fahrenheit(): number {
    return (this._celsius * 9) / 5 + 32;
  }

  set fahrenheit(value: number) {
    this.celsius = ((value - 32) * 5) / 9;
  }

  get kelvin(): number {
    return this._celsius + 273.15;
  }

  set kelvin(value: number) {
    this.celsius = value - 273.15;
  }
}

const temp = new Temperature(25);
console.log(temp.celsius); // 25
console.log(temp.fahrenheit); // 77
console.log(temp.kelvin); // 298.15

temp.fahrenheit = 100;
console.log(temp.celsius); // 37.77777777777778
```

Read-only Properties

```
class ImmutablePoint {
  private _x: number;
  private _y: number;

  constructor(x: number, y: number) {
    this._x = x;
    this._y = y;
  }

  // Read-only getters
  get x(): number {
    return this._x;
  }

  get y(): number {
    return this._y;
  }

  // Computed properties
  get magnitude(): number {
    return Math.sqrt(this._x * this._x + this._y * this._y);
  }

  get angle(): number {
    return Math.atan2(this._y, this._x);
  }

  // Methods that return new instances (immutable pattern)
  translate(dx: number, dy: number): ImmutablePoint {
    return new ImmutablePoint(this._x + dx, this._y + dy);
  }

  scale(factor: number): ImmutablePoint {
    return new ImmutablePoint(this._x * factor, this._y * factor);
  }
}

const point = new ImmutablePoint(3, 4);
console.log(point.x, point.y); // 3, 4
console.log(point.magnitude); // 5
// point.x = 5; // Error: Cannot assign to 'x' because it is a read-only property

const newPoint = point.translate(1, 1);
console.log(newPoint.x, newPoint.y); // 4, 5
```

Inheritance

Basic Inheritance with extends

```
// Base class
class Animal {
  protected name: string;
  protected age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public makeSound(): string {
    return "Some generic animal sound";
  }

  public getInfo(): string {
    return `${this.name} is ${this.age} years old`;
  }

  protected sleep(): string {
    return `${this.name} is sleeping`;
  }
}

// Derived class
class Dog extends Animal {
  private breed: string;

  constructor(name: string, age: number, breed: string) {
    super(name, age); // Call parent constructor
    this.breed = breed;
  }

  // Override parent method
  public makeSound(): string {
    return "Woof! Woof!";
  }

  // Add new method
  public fetch(): string {
    return `${this.name} is fetching the ball`;
  }

  // Override and extend parent method
  public getInfo(): string {
    return `${super.getInfo()} and is a ${this.breed}`;
  }

  // Access protected method from parent
  public rest(): string {
    return this.sleep(); // Can access protected method
  }
}

class Cat extends Animal {
  private indoor: boolean;

  constructor(name: string, age: number, indoor: boolean = true) {
    super(name, age);
    this.indoor = indoor;
  }
  public makeSound(): string {
    return "Meow!";
  }

  public climb(): string {
    return `${this.name} is climbing`;
  }

  public getInfo(): string {
    const location = this.indoor ? "indoor" : "outdoor";
    return `${super.getInfo()} and is an ${location} cat`;
  }
}

// Usage
const dog = new Dog("Buddy", 3, "Golden Retriever");
const cat = new Cat("Whiskers", 2, true);

console.log(dog.makeSound()); // "Woof! Woof!"
console.log(cat.makeSound()); // "Meow!"
console.log(dog.getInfo()); // "Buddy is 3 years old and is a Golden Retriever"
console.log(cat.getInfo()); // "Whiskers is 2 years old and is an indoor cat"
```

Method Overriding and super

```
class Vehicle {
  protected brand: string;
  protected model: string;
  protected year: number;

  constructor(brand: string, model: string, year: number) {
    this.brand = brand;
    this.model = model;
    this.year = year;
  }

  public start(): string {
    return `Starting the ${this.brand} ${this.model}`;
  }

  public getDescription(): string {
    return `${this.year} ${this.brand} ${this.model}`;
  }

  protected performMaintenance(): string {
    return "Performing basic maintenance";
  }
}

class ElectricCar extends Vehicle {
  private batteryCapacity: number;
  private currentCharge: number;

  constructor(
    brand: string,
    model: string,
    year: number,
    batteryCapacity: number
  ) {
    super(brand, model, year);
    this.batteryCapacity = batteryCapacity;
    this.currentCharge = batteryCapacity; // Start fully charged
  }

  // Override start method
  public start(): string {
    if (this.currentCharge > 0) {
      return `${super.start()} - Electric motor engaged`;
    }
    return "Cannot start - battery depleted";
  }

  // Override and extend getDescription
  public getDescription(): string {
    return `${super.getDescription()} (Electric - ${this.batteryCapacity}kWh)`;
  }

  // New methods specific to electric cars
  public charge(amount: number): void {
    this.currentCharge = Math.min(
      this.currentCharge + amount,
      this.batteryCapacity
    );
  }

  public getBatteryLevel(): number {
    return (this.currentCharge / this.batteryCapacity) * 100;
  }

  // Override protected method
  protected performMaintenance(): string {
    return `${super.performMaintenance()} + Battery health check`;
  }

  public scheduleMaintenance(): string {
    return this.performMaintenance(); // Can access overridden protected method
  }
}

const tesla = new ElectricCar("Tesla", "Model 3", 2023, 75);
console.log(tesla.start()); // "Starting the Tesla Model 3 - Electric motor engaged"
console.log(tesla.getDescription()); // "2023 Tesla Model 3 (Electric - 75kWh)"
console.log(tesla.getBatteryLevel()); // 100
```

Static Members

Static Properties and Methods

```
class MathUtils {
  // Static properties
  static readonly PI = 3.14159;
  static readonly E = 2.71828;
  private static instanceCount = 0;

  // Instance property
  private id: number;

  constructor() {
    MathUtils.instanceCount++;
    this.id = MathUtils.instanceCount;
  }

  // Static methods
  static add(a: number, b: number): number {
    return a + b;
  }

  static multiply(a: number, b: number): number {
    return a * b;
  }

  static circleArea(radius: number): number {
    return MathUtils.PI * radius * radius;
  }

  static getInstanceCount(): number {
    return MathUtils.instanceCount;
  }

  // Instance method
  getId(): number {
    return this.id;
  }

  // Instance method accessing static member
  getCircleArea(radius: number): number {
    return MathUtils.circleArea(radius); // Access static method
  }
}

// Using static members without creating instances
console.log(MathUtils.PI); // 3.14159
console.log(MathUtils.add(5, 3)); // 8
console.log(MathUtils.circleArea(5)); // 78.5395
```

```
// Creating instances
const math1 = new MathUtils();
const math2 = new MathUtils();

console.log(math1.getId()); // 1
console.log(math2.getId()); // 2
console.log(MathUtils.getInstanceCount()); // 2
```

Static Factory Methods

```
class User {
  private constructor(
    public readonly id: string,
    public readonly name: string,
    public readonly email: string,
    public readonly role: "admin" | "user" | "guest"
  ) {}

  // Static factory methods
  static createAdmin(name: string, email: string): User {
    return new User(`admin_${Date.now()}`, name, email, "admin");
  }

  static createUser(name: string, email: string): User {
    return new User(`user_${Date.now()}`, name, email, "user");
  }

  static createGuest(): User {
    return new User(
      `guest_${Date.now()}`,
      "Guest",
      "guest@example.com",
      "guest"
    );
  }

  // Static validation method
  static isValidEmail(email: string): boolean {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
  }

  // Static factory with validation
  static fromData(data: {
    name: string;
    email: string;
    role?: "admin" | "user";
  }): User | null {
    if (!User.isValidEmail(data.email)) {
      return null;
    }
    const role = data.role || "user";
    return role === "admin"
      ? User.createAdmin(data.name, data.email)
      : User.createUser(data.name, data.email);
  }
}

// Usage
const admin = User.createAdmin("John Doe", "john@admin.com");
const user = User.createUser("Jane Smith", "jane@user.com");
const guest = User.createGuest();

const userFromData = User.fromData({
  name: "Bob Wilson",
  email: "bob@example.com",
  role: "user",
});
```

Abstract Classes

Abstract Classes and Methods

```
// Abstract base class
abstract class Shape {
  protected color: string;

  constructor(color: string) {
    this.color = color;
  }

  // Abstract method - must be implemented by subclasses
  abstract calculateArea(): number;
  abstract calculatePerimeter(): number;

  // Concrete method - can be used by subclasses
  getColor(): string {
    return this.color;
  }

  // Concrete method using abstract methods
  getDescription(): string {
    return `A ${this.color} shape with area ${this.calculateArea().toFixed(
      2
    )} and perimeter ${this.calculatePerimeter().toFixed(2)}`;
  }

  // Abstract method with default implementation
  display(): void {
    console.log(this.getDescription());
  }
}
```

```
// Concrete implementation
class Circle extends Shape {
  private radius: number;

  constructor(color: string, radius: number) {
    super(color);
    this.radius = radius;
  }

  // Must implement abstract methods
  calculateArea(): number {
    return Math.PI * this.radius * this.radius;
  }

  calculatePerimeter(): number {
    return 2 * Math.PI * this.radius;
  }

  // Additional method specific to Circle
  getDiameter(): number {
    return this.radius * 2;
  }
}

class Rectangle extends Shape {
  private width: number;
  private height: number;

  constructor(color: string, width: number, height: number) {
    super(color);
    this.width = width;
    this.height = height;
  }

  calculateArea(): number {
    return this.width * this.height;
  }

  calculatePerimeter(): number {
    return 2 * (this.width + this.height);
  }

  // Override display method
  display(): void {
    console.log(`Rectangle: ${this.getDescription()}`);
  }
}

// Usage
const circle = new Circle("red", 5);
const rectangle = new Rectangle("blue", 4, 6);

circle.display(); // "A red shape with area 78.54 and perimeter 31.42"
```

```
rectangle.display(); // "Rectangle: A blue shape with area 24.00 and perimeter 20.00"

// Cannot instantiate abstract class
// const shape = new Shape("green"); // Error: Cannot create an instance of an abstract class

// Array of shapes
const shapes: Shape[] = [circle, rectangle];
shapes.forEach((shape) => shape.display());
```

Implementing Interfaces

Classes Implementing Interfaces

```
// Interface definitions
interface Flyable {
  fly(): string;
  altitude: number;
}

interface Swimmable {
  swim(): string;
  depth: number;
}

interface Walkable {
  walk(): string;
  speed: number;
}

// Class implementing single interface
class Bird implements Flyable {
  altitude: number = 0;

  constructor(private species: string) {}

  fly(): string {
    this.altitude = 100;
    return `${this.species} is flying at ${this.altitude} feet`;
  }

  land(): void {
    this.altitude = 0;
  }
}

// Class implementing multiple interfaces
class Duck implements Flyable, Swimmable, Walkable {
  altitude: number = 0;
  depth: number = 0;
  speed: number = 2;

  constructor(private name: string) {}

  fly(): string {
    this.altitude = 50;
    return `${this.name} is flying at ${this.altitude} feet`;
  }

  swim(): string {
    this.depth = 3;
    return `${this.name} is swimming at ${this.depth} feet deep`;
  }

  walk(): string {
    return `${this.name} is walking at ${this.speed} mph`;
  }

  // Additional methods
  quack(): string {
    return `${this.name} says quack!`;
  }
}

// Interface for class structure
interface Drawable {
  draw(): void;
  getArea(): number;
}

class Square implements Drawable {
  constructor(private sideLength: number) {}

  draw(): void {
    console.log(`Drawing a square with side length ${this.sideLength}`);
  }

  getArea(): number {
    return this.sideLength * this.sideLength;
  }
}

// Usage
const bird = new Bird("Eagle");
const duck = new Duck("Donald");
const square = new Square(5);

console.log(bird.fly()); // "Eagle is flying at 100 feet"
console.log(duck.swim()); // "Donald is swimming at 3 feet deep"
console.log(duck.quack()); // "Donald says quack!"

square.draw(); // "Drawing a square with side length 5"
console.log(square.getArea()); // 25
```

Practical Examples

Event System

```
// Event system using classes
abstract class EventEmitter {
  private listeners: Map<string, Function[]> = new Map();

  on(event: string, listener: Function): void {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, []);
    }
    this.listeners.get(event)!.push(listener);
  }

  off(event: string, listener: Function): void {
    const eventListeners = this.listeners.get(event);
    if (eventListeners) {
      const index = eventListeners.indexOf(listener);
      if (index > -1) {
        eventListeners.splice(index, 1);
      }
    }
  }

  protected emit(event: string, ...args: any[]): void {
    const eventListeners = this.listeners.get(event);
    if (eventListeners) {
      eventListeners.forEach((listener) => listener(...args));
    }
  }
}

class HttpClient extends EventEmitter {
  private baseUrl: string;

  constructor(baseUrl: string) {
    super();
    this.baseUrl = baseUrl;
  }

  async get(endpoint: string): Promise<any> {
    this.emit("request:start", { method: "GET", endpoint });

    try {
      const response = await fetch(`${this.baseUrl}${endpoint}`);
      const data = await response.json();

      this.emit("request:success", { method: "GET", endpoint, data });
      return data;
    } catch (error) {
      this.emit("request:error", { method: "GET", endpoint, error });
      throw error;
    }
  }
}

// Usage
const client = new HttpClient("https://api.example.com");

client.on("request:start", (details) => {
  console.log("Request started:", details);
});

client.on("request:success", (details) => {
  console.log("Request successful:", details);
});

client.on("request:error", (details) => {
  console.error("Request failed:", details);
});
```

Repository Pattern

```
// Repository pattern with classes
interface Repository<T> {
  findById(id: string): Promise<T | null>;
  findAll(): Promise<T[]>;
  create(entity: Omit<T, "id">): Promise<T>;
  update(id: string, updates: Partial<T>): Promise<T | null>;
  delete(id: string): Promise<boolean>;
}

abstract class BaseRepository<T extends { id: string }>
  implements Repository<T>
{
  protected items: Map<string, T> = new Map();

  async findById(id: string): Promise<T | null> {
    return this.items.get(id) || null;
  }

  async findAll(): Promise<T[]> {
    return Array.from(this.items.values());
  }

  async create(entity: Omit<T, "id">): Promise<T> {
    const id = this.generateId();
    const newEntity = { ...entity, id } as T;
    this.items.set(id, newEntity);
    return newEntity;
  }
  async update(id: string, updates: Partial<T>): Promise<T | null> {
    const existing = this.items.get(id);
    if (!existing) return null;

    const updated = { ...existing, ...updates };
    this.items.set(id, updated);
    return updated;
  }

  async delete(id: string): Promise<boolean> {
    return this.items.delete(id);
  }

  protected abstract generateId(): string;
}

interface User {
  id: string;
  name: string;
  email: string;
  createdAt: Date;
}

class UserRepository extends BaseRepository<User> {
  protected generateId(): string {
    return `user_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
  }

  // Additional user-specific methods
  async findByEmail(email: string): Promise<User | null> {
    const users = await this.findAll();
    return users.find((user) => user.email === email) || null;
  }

  async findActiveUsers(): Promise<User[]> {
    const users = await this.findAll();
    const thirtyDaysAgo = new Date(Date.now() - 30 * 24 * 60 * 60 * 1000);
    return users.filter((user) => user.createdAt > thirtyDaysAgo);
  }
}

// Usage
const userRepo = new UserRepository();

async function example() {
  const user = await userRepo.create({
    name: "John Doe",
    email: "john@example.com",
    createdAt: new Date(),
  });

  console.log("Created user:", user);
  const foundUser = await userRepo.findByEmail("john@example.com");
  console.log("Found user:", foundUser);
}
```

Best Practices

✅ Good Practices

```
// Use access modifiers appropriately
class GoodExample {
  private _data: string[]; // Private implementation detail
  protected config: object; // For subclasses
  public readonly id: string; // Public immutable property

  constructor(id: string) {
    this.id = id;
    this._data = [];
    this.config = {};
  }

  // Public interface
  public addItem(item: string): void {
    this.validateItem(item);
    this._data.push(item);
  }

  // Private helper
  private validateItem(item: string): void {
    if (!item.trim()) {
      throw new Error("Item cannot be empty");
    }
  }
}

// Use composition over inheritance when appropriate
class Logger {
  log(message: string): void {
    console.log(`[${new Date().toISOString()}] ${message}`);
  }
}

class UserService {
  private logger = new Logger(); // Composition

  createUser(userData: any): User {
    this.logger.log("Creating user");
    // Implementation
    return new User(userData);
  }
}
```

```
// Use interfaces for contracts
interface PaymentProcessor {
  processPayment(amount: number): Promise<boolean>;
}

class StripeProcessor implements PaymentProcessor {
  async processPayment(amount: number): Promise<boolean> {
    // Stripe implementation
    return true;
  }
}

class PayPalProcessor implements PaymentProcessor {
  async processPayment(amount: number): Promise<boolean> {
    // PayPal implementation
    return true;
  }
}
```

❌ Avoid

```
// Don't make everything public
class BadExample {
  public internalData: any; // Should be private
  public helperMethod(): void {} // Should be private
}

// Don't use inheritance for code reuse only
class BadInheritance extends Array {
  // This is confusing - is it an array or something else?
}

// Don't ignore access modifiers
class IgnoredModifiers {
  private secret: string;

  constructor() {
    this.secret = "secret";
  }
}

// Accessing private members (bad practice)
const bad = new IgnoredModifiers();
// (bad as any).secret; // Don't do this!
```

Summary Checklist

- ☐ Use appropriate access modifiers (public, private, protected)
- ☐ Understand the difference between TypeScript private and # private fields
- ☐ Use getters and setters for controlled property access

- □ Implement inheritance with extends and super
- □ Use abstract classes for shared behavior with required implementations
- □ Implement interfaces to define contracts
- □ Use static members for class-level functionality
- □ Prefer composition over inheritance when appropriate
- □ Use readonly for immutable properties
- □ Follow the principle of least privilege for access modifiers

Next Steps

Now that you understand classes and OOP in TypeScript, let's explore generics for creating reusable and type-safe code.

Continue to: Generics and Reusable Code

Generics and Reusable Code

> Master TypeScript generics to create flexible, reusable, and type-safe code components

Introduction to Generics

Generics allow you to create reusable components that work with multiple types while maintaining type safety.

Basic Generic Functions

```
// Without generics - limited to specific types
function identityString(arg: string): string {
  return arg;
}

function identityNumber(arg: number): number {
  return arg;
}

// With generics - works with any type
function identity<T>(arg: T): T {
  return arg;
}

// Usage
const stringResult = identity<string>("hello"); // Type: string
const numberResult = identity<number>(42); // Type: number
const booleanResult = identity<boolean>(true); // Type: boolean

// Type inference - TypeScript can infer the type
const inferredString = identity("hello"); // Type: string (inferred)
const inferredNumber = identity(42); // Type: number (inferred)
```

```
// Generic function with multiple type parameters
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}

const stringNumberPair = pair("hello", 42); // Type: [string, number]
const booleanStringPair = pair(true, "world"); // Type: [boolean, string]
```

Generic Array Functions

```
// Generic function working with arrays
function getFirstElement<T>(array: T[]): T | undefined {
  return array.length > 0 ? array[0] : undefined;
}

const numbers = [1, 2, 3, 4, 5];
const strings = ["a", "b", "c"];
const booleans = [true, false, true];

const firstNumber = getFirstElement(numbers); // Type: number | undefined
const firstString = getFirstElement(strings); // Type: string | undefined
const firstBoolean = getFirstElement(booleans); // Type: boolean | undefined

// Generic function for array manipulation
function map<T, U>(array: T[], transform: (item: T) => U): U[] {
  const result: U[] = [];
  for (const item of array) {
    result.push(transform(item));
  }
  return result;
}

// Usage
const doubled = map([1, 2, 3], (x) => x * 2); // Type: number[]
const lengths = map(["hello", "world"], (s) => s.length); // Type: number[]
const uppercased = map(["a", "b", "c"], (s) => s.toUpperCase()); // Type: string[]

// Generic filter function
function filter<T>(array: T[], predicate: (item: T) => boolean): T[] {
  const result: T[] = [];
  for (const item of array) {
    if (predicate(item)) {
      result.push(item);
    }
  }
  return result;
}

const evenNumbers = filter([1, 2, 3, 4, 5], (n) => n % 2 === 0); // [2, 4]
const longStrings = filter(["a", "hello", "hi", "world"], (s) => s.length > 2); // ["hello", "world"]
```

Generic Interfaces

Basic Generic Interfaces

```
// Generic interface for a container
interface Container<T> {
  value: T;
  getValue(): T;
  setValue(value: T): void;
}

// Implementation for different types
class StringContainer implements Container<string> {
  constructor(public value: string) {}

  getValue(): string {
    return this.value;
  }

  setValue(value: string): void {
    this.value = value;
  }
}

class NumberContainer implements Container<number> {
  constructor(public value: number) {}

  getValue(): number {
    return this.value;
  }

  setValue(value: number): void {
    this.value = value;
  }
}

// Generic implementation
class GenericContainer<T> implements Container<T> {
  constructor(public value: T) {}

  getValue(): T {
    return this.value;
  }

  setValue(value: T): void {
    this.value = value;
  }
}

// Usage
const stringContainer = new GenericContainer<string>("hello");
const numberContainer = new GenericContainer<number>(42);
const booleanContainer = new GenericContainer<boolean>(true);
```

Generic Interfaces for Data Structures

```
// Generic interface for a key-value store
interface KeyValueStore<K, V> {
  get(key: K): V | undefined;
  set(key: K, value: V): void;
  has(key: K): boolean;
  delete(key: K): boolean;
  keys(): K[];
  values(): V[];
  entries(): [K, V][];
}

// Implementation using Map
class MapStore<K, V> implements KeyValueStore<K, V> {
  private store = new Map<K, V>();

  get(key: K): V | undefined {
    return this.store.get(key);
  }

  set(key: K, value: V): void {
    this.store.set(key, value);
  }

  has(key: K): boolean {
    return this.store.has(key);
  }

  delete(key: K): boolean {
    return this.store.delete(key);
  }

  keys(): K[] {
    return Array.from(this.store.keys());
  }

  values(): V[] {
    return Array.from(this.store.values());
  }

  entries(): [K, V][] {
    return Array.from(this.store.entries());
  }
}

// Usage
const userStore = new MapStore<string, { name: string; age: number }>();
```

```
userStore.set("user1", { name: "John", age: 30 });
userStore.set("user2", { name: "Jane", age: 25 });

const user = userStore.get("user1"); // Type: { name: string; age: number } |
undefined

// Generic interface for API responses
interface ApiResponse<T> {
  data: T;
  status: number;
  message: string;
  timestamp: Date;
}

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

interface Product {
  id: number;
  name: string;
  price: number;
}

// Usage with different data types
const userResponse: ApiResponse<User> = {
  data: { id: 1, name: "John", email: "john@example.com" },
  status: 200,
  message: "Success",
  timestamp: new Date(),
};

const productsResponse: ApiResponse<Product[]> = {
  data: [
    { id: 1, name: "Laptop", price: 999 },
    { id: 2, name: "Mouse", price: 25 },
  ],
  status: 200,
  message: "Success",
  timestamp: new Date(),
};
```

Generic Classes

Basic Generic Classes

```
// Generic class for a simple stack
class Stack<T> {
  private items: T[] = [];
  push(item: T): void {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }

  peek(): T | undefined {
    return this.items[this.items.length - 1];
  }

  isEmpty(): boolean {
    return this.items.length === 0;
  }

  size(): number {
    return this.items.length;
  }

  toArray(): T[] {
    return [...this.items];
  }
}

// Usage
const numberStack = new Stack<number>();
numberStack.push(1);
numberStack.push(2);
numberStack.push(3);

console.log(numberStack.pop()); // 3
console.log(numberStack.peek()); // 2

const stringStack = new Stack<string>();
stringStack.push("hello");
stringStack.push("world");

console.log(stringStack.toArray()); // ["hello", "world"]
```

Generic Classes with Multiple Type Parameters

```
// Generic class for a result type (similar to Rust's Result or Haskell's Either)
class Result<T, E> {
  private constructor(
    private readonly _value: T | null,
    private readonly _error: E | null,
    private readonly _isSuccess: boolean
  ) {}
  static success<T, E>(value: T): Result<T, E> {
    return new Result<T, E>(value, null, true);
  }

  static failure<T, E>(error: E): Result<T, E> {
    return new Result<T, E>(null, error, false);
  }

  isSuccess(): boolean {
    return this._isSuccess;
  }

  isFailure(): boolean {
    return !this._isSuccess;
  }

  getValue(): T {
    if (!this._isSuccess) {
      throw new Error("Cannot get value from failed result");
    }
    return this._value!;
  }

  getError(): E {
    if (this._isSuccess) {
      throw new Error("Cannot get error from successful result");
    }
    return this._error!;
  }

  map<U>(fn: (value: T) => U): Result<U, E> {
    if (this._isSuccess) {
      return Result.success<U, E>(fn(this._value!));
    }
    return Result.failure<U, E>(this._error!);
  }

  flatMap<U>(fn: (value: T) => Result<U, E>): Result<U, E> {
    if (this._isSuccess) {
      return fn(this._value!);
    }
    return Result.failure<U, E>(this._error!);
  }
}

// Usage
function divide(a: number, b: number): Result<number, string> {
  if (b === 0) {
    return Result.failure("Division by zero");
  }
  return Result.success(a / b);
}

function sqrt(x: number): Result<number, string> {
  if (x < 0) {
    return Result.failure("Cannot take square root of negative number");
  }
  return Result.success(Math.sqrt(x));
}

// Chaining operations
const result = divide(10, 2)
  .flatMap((x) => sqrt(x))
  .map((x) => x.toFixed(2));

if (result.isSuccess()) {
  console.log("Result:", result.getValue()); // "2.24"
} else {
  console.log("Error:", result.getError());
}
```

Generic Constraints

Basic Constraints with extends

```
// Constraint: T must have a length property
interface Lengthwise {
  length: number;
}

function logLength<T extends Lengthwise>(arg: T): T {
  console.log(`Length: ${arg.length}`);
  return arg;
}

// Valid calls
logLength("hello"); // string has length
logLength([1, 2, 3]); // array has length
logLength({ length: 10, value: "test" }); // object with length property

// Invalid call
// logLength(123); // Error: number doesn't have length property

// Constraint: T must extend a specific type
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const person = { name: "John", age: 30, email: "john@example.com" };

const name = getProperty(person, "name"); // Type: string
const age = getProperty(person, "age"); // Type: number
// const invalid = getProperty(person, "invalid"); // Error: "invalid" is not a key of person
```

Multiple Constraints

```
// Multiple constraints
interface Serializable {
  serialize(): string;
}

interface Timestamped {
  timestamp: Date;
}

// T must implement both interfaces
function processData<T extends Serializable & Timestamped>(data: T): string {
  const serialized = data.serialize();
  const time = data.timestamp.toISOString();
  return `${time}: ${serialized}`;
}

class LogEntry implements Serializable, Timestamped {
  constructor(public message: string, public timestamp: Date = new Date()) {}

  serialize(): string {
    return JSON.stringify({ message: this.message, timestamp: this.timestamp });
  }
}

const entry = new LogEntry("User logged in");
const processed = processData(entry);

// Constraint with conditional types
type NonNullable<T> = T extends null | undefined ? never : T;

function ensureNonNull<T>(value: T): NonNullable<T> {
  if (value === null || value === undefined) {
    throw new Error("Value cannot be null or undefined");
  }
  return value as NonNullable<T>;
}

const maybeString: string | null = "hello";
const definiteString = ensureNonNull(maybeString); // Type: string
```

Constraints with Class Types

```
// Generic constraint with constructor
interface Constructable {
  new (...args: any[]): any;
}

function createInstance<T extends Constructable>(
  ctor: T,
  ...args: any[]
): InstanceType<T> {
  return new ctor(...args);
}

class User {
  constructor(public name: string, public age: number) {}
}

class Product {
  constructor(public name: string, public price: number) {}
}

const user = createInstance(User, "John", 30); // Type: User
const product = createInstance(Product, "Laptop", 999); // Type: Product

// Generic factory with constraints
abstract class Animal {
  abstract makeSound(): string;
}

class Dog extends Animal {
  makeSound(): string {
    return "Woof!";
  }
}

class Cat extends Animal {
  makeSound(): string {
    return "Meow!";
  }
}

function createAnimal<T extends Animal>(AnimalClass: new () => T): T {
  return new AnimalClass();
}

const dog = createAnimal(Dog); // Type: Dog
const cat = createAnimal(Cat); // Type: Cat
```

Advanced Generic Patterns

Conditional Types

```
// Basic conditional type
type IsString<T> = T extends string ? true : false;

type Test1 = IsString<string>; // true
type Test2 = IsString<number>; // false
```

```
// Conditional type for extracting array element type
type ArrayElement<T> = T extends (infer U)[] ? U : never;

type StringArrayElement = ArrayElement<string[]>; // string
type NumberArrayElement = ArrayElement<number[]>; // number
type NotArrayElement = ArrayElement<string>; // never

// Conditional type for function return type
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

type FunctionReturn = ReturnType<() => string>; // string
type MethodReturn = ReturnType<(x: number) => boolean>; // boolean

// Practical conditional type for API responses
type ApiResult<T> = T extends { error: any }
  ? { success: false; error: T["error"] }
  : { success: true; data: T };

type SuccessResult = ApiResult<{ name: string }>; // { success: true; data: { name: string } }
type ErrorResult = ApiResult<{ error: string }>; // { success: false; error: string }
```

Mapped Types with Generics

```
// Generic mapped type for making properties optional
type Partial<T> = {
  [P in keyof T]?: T[P];
};

// Generic mapped type for making properties required
type Required<T> = {
  [P in keyof T]-?: T[P];
};

// Generic mapped type for making properties readonly
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

// Custom mapped type for nullable properties
type Nullable<T> = {
  [P in keyof T]: T[P] | null;
};

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

```
type PartialUser = Partial<User>; // All properties optional
type RequiredUser = Required<User>; // All properties required (including age)
type ReadonlyUser = Readonly<User>; // All properties readonly
type NullableUser = Nullable<User>; // All properties can be null

// Generic mapped type with transformation
type Stringify<T> = {
  [P in keyof T]: string;
};

type StringifiedUser = Stringify<User>; // All properties are strings

// Generic mapped type with filtering
type PickByType<T, U> = {
  [P in keyof T as T[P] extends U ? P : never]: T[P];
};

type StringProperties = PickByType<User, string>; // { name: string; email: string
}
type NumberProperties = PickByType<User, number>; // { id: number; age?: number }
```

Generic Utility Functions

```
// Generic deep clone function
function deepClone<T>(obj: T): T {
  if (obj === null || typeof obj !== "object") {
    return obj;
  }

  if (obj instanceof Date) {
    return new Date(obj.getTime()) as T;
  }

  if (obj instanceof Array) {
    return obj.map((item) => deepClone(item)) as T;
  }

  if (typeof obj === "object") {
    const cloned = {} as T;
    for (const key in obj) {
      if (obj.hasOwnProperty(key)) {
        cloned[key] = deepClone(obj[key]);
      }
    }
    return cloned;
  }

  return obj;
}
```

```typescript
// Generic memoization function
function memoize<TArgs extends any[], TReturn>(
  fn: (...args: TArgs) => TReturn
): (...args: TArgs) => TReturn {
  const cache = new Map<string, TReturn>();

  return (...args: TArgs): TReturn => {
    const key = JSON.stringify(args);

    if (cache.has(key)) {
      return cache.get(key)!;
    }

    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}

// Usage
const expensiveFunction = (x: number, y: number): number => {
  console.log(`Computing ${x} + ${y}`);
  return x + y;
};

const memoizedFunction = memoize(expensiveFunction);

console.log(memoizedFunction(1, 2)); // "Computing 1 + 2", returns 3
console.log(memoizedFunction(1, 2)); // Returns 3 (from cache, no console.log)

// Generic retry function
async function retry<T>(
  fn: () => Promise<T>,
  maxAttempts: number = 3,
  delay: number = 1000
): Promise<T> {
  let lastError: Error;

  for (let attempt = 1; attempt <= maxAttempts; attempt++) {
    try {
      return await fn();
    } catch (error) {
      lastError = error as Error;

      if (attempt === maxAttempts) {
        throw lastError;
      }

      await new Promise((resolve) => setTimeout(resolve, delay));
    }
  }

  throw lastError!;
}
```

```typescript
// Usage
const fetchData = async (): Promise<string> => {
  const response = await fetch("https://api.example.com/data");
  if (!response.ok) {
    throw new Error("Failed to fetch");
  }
  return response.text();
};

const dataWithRetry = await retry(fetchData, 3, 2000);
```

Practical Examples

Generic Repository Pattern

```typescript
// Generic repository interface
interface Repository<T, ID> {
  findById(id: ID): Promise<T | null>;
  findAll(): Promise<T[]>;
  create(entity: Omit<T, "id">): Promise<T>;
  update(id: ID, updates: Partial<T>): Promise<T | null>;
  delete(id: ID): Promise<boolean>;
}

// Generic base repository implementation
abstract class BaseRepository<T extends { id: ID }, ID>
  implements Repository<T, ID>
{
  protected items: Map<ID, T> = new Map();

  async findById(id: ID): Promise<T | null> {
    return this.items.get(id) || null;
  }

  async findAll(): Promise<T[]> {
    return Array.from(this.items.values());
  }

  async create(entity: Omit<T, "id">): Promise<T> {
    const id = this.generateId();
    const newEntity = { ...entity, id } as T;
    this.items.set(id, newEntity);
    return newEntity;
  }

  async update(id: ID, updates: Partial<T>): Promise<T | null> {
    const existing = this.items.get(id);
    if (!existing) return null;

    const updated = { ...existing, ...updates };
    this.items.set(id, updated);
    return updated;
  }

  async delete(id: ID): Promise<boolean> {
    return this.items.delete(id);
  }

  protected abstract generateId(): ID;
}

// Specific repository implementations
interface User {
  id: string;
  name: string;
  email: string;
  createdAt: Date;
}

class UserRepository extends BaseRepository<User, string> {
  protected generateId(): string {
    return `user_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
  }

  async findByEmail(email: string): Promise<User | null> {
    const users = await this.findAll();
    return users.find((user) => user.email === email) || null;
  }
}

interface Product {
  id: number;
  name: string;
  price: number;
  category: string;
}

class ProductRepository extends BaseRepository<Product, number> {
  private nextId = 1;

  protected generateId(): number {
    return this.nextId++;
  }

  async findByCategory(category: string): Promise<Product[]> {
    const products = await this.findAll();
    return products.filter((product) => product.category === category);
  }
}
```

Generic Event System

```
// Generic event system
type EventMap = Record<string, any>;

class TypedEventEmitter<TEvents extends EventMap> {
  private listeners: {
    [K in keyof TEvents]?: Array<(data: TEvents[K]) => void>;
  } = {};

  on<K extends keyof TEvents>(
    event: K,
    listener: (data: TEvents[K]) => void
  ): void {
    if (!this.listeners[event]) {
      this.listeners[event] = [];
    }
    this.listeners[event]!.push(listener);
  }

  off<K extends keyof TEvents>(
    event: K,
    listener: (data: TEvents[K]) => void
  ): void {
    const eventListeners = this.listeners[event];
    if (eventListeners) {
      const index = eventListeners.indexOf(listener);
      if (index > -1) {
        eventListeners.splice(index, 1);
      }
    }
  }

  emit<K extends keyof TEvents>(event: K, data: TEvents[K]): void {
    const eventListeners = this.listeners[event];
    if (eventListeners) {
      eventListeners.forEach((listener) => listener(data));
    }
  }
}

// Define event types
interface AppEvents {
  "user:login": { userId: string; timestamp: Date };
  "user:logout": { userId: string; timestamp: Date };
  "product:created": { productId: number; name: string };
  "product:updated": { productId: number; changes: string[] };
}

// Usage with type safety
const eventEmitter = new TypedEventEmitter<AppEvents>();

// Type-safe event listeners
eventEmitter.on("user:login", (data) => {
  console.log(`User ${data.userId} logged in at ${data.timestamp}`);
});

eventEmitter.on("product:created", (data) => {
  console.log(`Product ${data.name} created with ID ${data.productId}`);
});

// Type-safe event emission
eventEmitter.emit("user:login", {
  userId: "user123",
  timestamp: new Date(),
});

eventEmitter.emit("product:created", {
  productId: 1,
  name: "Laptop",
});

// TypeScript will catch errors
// eventEmitter.emit('user:login', { userId: 123 }); // Error: userId should be string
// eventEmitter.emit('invalid:event', {}); // Error: event doesn't exist
```

Best Practices

✅ Good Practices

```
// Use meaningful generic parameter names
interface Repository<TEntity, TKey> {
  // Clear what T represents
  findById(id: TKey): Promise<TEntity | null>;
}

// Use constraints to make generics more specific
function processItems<T extends { id: string }>(items: T[]): T[] {
  return items.filter((item) => item.id.length > 0);
}

// Provide default type parameters when appropriate
interface ApiResponse<TData = any, TError = string> {
  data?: TData;
  error?: TError;
  success: boolean;
}

// Use generic constraints for better type safety
function updateEntity<T extends { id: string }>(
  entity: T,
  updates: Partial<Omit<T, "id">>
): T {
  return { ...entity, ...updates };
}
```

```typescript
// Use conditional types for complex type transformations
type NonNullable<T> = T extends null | undefined ? never : T;

function assertNonNull<T>(value: T): asserts value is NonNullable<T> {
  if (value === null || value === undefined) {
    throw new Error("Value is null or undefined");
  }
}
```

❌ Avoid

```typescript
// Don't use single-letter names without context
function bad<T, U, V>(a: T, b: U): V {
  // What do these represent?
  // Implementation
}

// Don't make everything generic unnecessarily
function unnecessarilyGeneric<T>(value: T): T {
  return value; // This doesn't add value
}

// Better: Only make it generic if it needs to be
function identity(value: string): string {
  return value;
}

// Don't use any in generic constraints
function badConstraint<T extends any>(value: T): T {
  return value; // any defeats the purpose
}

// Don't ignore type safety with generics
function unsafe<T>(value: any): T {
  return value as T; // Dangerous casting
}
```

Summary Checklist

- ☐ Understand basic generic syntax with `<T>`
- ☐ Use generics for functions, interfaces, and classes
- ☐ Apply generic constraints with `extends`
- ☐ Use multiple type parameters when needed
- ☐ Understand conditional types and mapped types
- ☐ Use meaningful names for generic parameters
- ☐ Apply constraints to make generics more specific
- ☐ Leverage type inference when possible
- ☐ Use utility types like `Partial<T>`, `Required<T>`, etc.

- □ Create reusable generic patterns for common scenarios

Next Steps

Now that you understand generics, let's explore type guards and advanced type checking techniques in TypeScript.

Continue to: Type Guards and Advanced Type Checking

Type Guards and Advanced Type Checking

> Master TypeScript's type guards, type predicates, and advanced type checking techniques for runtime type safety

Introduction to Type Guards

Type guards are runtime checks that help TypeScript narrow down types within conditional blocks, providing both runtime safety and compile-time type information.

Built-in Type Guards

typeof Type Guard

```
// Basic typeof type guard
function processValue(value: string | number | boolean): string {
  if (typeof value === "string") {
    // TypeScript knows value is string here
    return value.toUpperCase();
  }

  if (typeof value === "number") {
    // TypeScript knows value is number here
    return value.toFixed(2);
  }

  if (typeof value === "boolean") {
    // TypeScript knows value is boolean here
    return value ? "true" : "false";
  }

  // TypeScript knows this is unreachable
  return "unknown";
}

// More complex typeof usage
function handleInput(input: unknown): string {
  if (typeof input === "string") {
    return `String: ${input}`;
  }
  if (typeof input === "number") {
    return `Number: ${input}`;
  }

  if (typeof input === "object" && input !== null) {
    if (Array.isArray(input)) {
      return `Array with ${input.length} items`;
    }
    return `Object: ${JSON.stringify(input)}`;
  }

  return `Unknown type: ${typeof input}`;
}

// Function type checking
function executeIfFunction(value: unknown): any {
  if (typeof value === "function") {
    // TypeScript knows value is a function
    return value();
  }

  throw new Error("Value is not a function");
}
```

instanceof Type Guard

```
// Classes for instanceof examples
class Dog {
  constructor(public name: string) {}

  bark(): string {
    return `${this.name} says woof!`;
  }
}

class Cat {
  constructor(public name: string) {}

  meow(): string {
    return `${this.name} says meow!`;
  }
}

class Bird {
  constructor(public name: string) {}

  fly(): string {
    return `${this.name} is flying!`;
  }
}
```

```
// instanceof type guard
function handleAnimal(animal: Dog | Cat | Bird): string {
  if (animal instanceof Dog) {
    // TypeScript knows animal is Dog
    return animal.bark();
  }

  if (animal instanceof Cat) {
    // TypeScript knows animal is Cat
    return animal.meow();
  }

  if (animal instanceof Bird) {
    // TypeScript knows animal is Bird
    return animal.fly();
  }

  // TypeScript knows this is unreachable
  throw new Error("Unknown animal type");
}

// instanceof with built-in types
function processError(error: unknown): string {
  if (error instanceof Error) {
    return `Error: ${error.message}`;
  }

  if (error instanceof TypeError) {
    return `Type Error: ${error.message}`;
  }

  if (error instanceof RangeError) {
    return `Range Error: ${error.message}`;
  }

  return `Unknown error: ${String(error)}`;
}

// instanceof with Date, Array, etc.
function handleValue(value: unknown): string {
  if (value instanceof Date) {
    return `Date: ${value.toISOString()}`;
  }

  if (value instanceof Array) {
    return `Array with ${value.length} items`;
  }

  if (value instanceof RegExp) {
    return `RegExp: ${value.source}`;
  }
  return "Unknown value type";
}
```

in Operator Type Guard

```
// Interfaces for 'in' operator examples
interface Car {
  brand: string;
  model: string;
  drive(): void;
}

interface Boat {
  name: string;
  length: number;
  sail(): void;
}

interface Plane {
  model: string;
  capacity: number;
  fly(): void;
}

// 'in' operator type guard
function operateVehicle(vehicle: Car | Boat | Plane): string {
  if ("drive" in vehicle) {
    // TypeScript knows vehicle is Car
    vehicle.drive();
    return `Driving ${vehicle.brand} ${vehicle.model}`;
  }

  if ("sail" in vehicle) {
    // TypeScript knows vehicle is Boat
    vehicle.sail();
    return `Sailing ${vehicle.name} (${vehicle.length}ft)`;
  }

  if ("fly" in vehicle) {
    // TypeScript knows vehicle is Plane
    vehicle.fly();
    return `Flying ${vehicle.model} (capacity: ${vehicle.capacity})`;
  }

  throw new Error("Unknown vehicle type");
}

// 'in' operator with optional properties
interface User {
  id: number;
  name: string;
  email?: string;
  phone?: string;
}

function getContactInfo(user: User): string {
  const contacts: string[] = [];

  if ("email" in user && user.email) {
    contacts.push(`Email: ${user.email}`);
  }

  if ("phone" in user && user.phone) {
    contacts.push(`Phone: ${user.phone}`);
  }

  return contacts.length > 0
    ? contacts.join(", ")
    : "No contact information available";
}

// Complex 'in' operator usage
interface ApiSuccessResponse {
  success: true;
  data: any;
}

interface ApiErrorResponse {
  success: false;
  error: string;
  code: number;
}

type ApiResponse = ApiSuccessResponse | ApiErrorResponse;

function handleApiResponse(response: ApiResponse): string {
  if ("data" in response) {
    // TypeScript knows response is ApiSuccessResponse
    return `Success: ${JSON.stringify(response.data)}`;
  }

  if ("error" in response) {
    // TypeScript knows response is ApiErrorResponse
    return `Error ${response.code}: ${response.error}`;
  }

  throw new Error("Invalid response format");
}
```

Custom Type Guards

Type Predicate Functions

```typescript
// Basic type predicate
function isString(value: unknown): value is string {
  return typeof value === "string";
}

function isNumber(value: unknown): value is number {
  return typeof value === "number" && !isNaN(value);
}

function isBoolean(value: unknown): value is boolean {
  return typeof value === "boolean";
}

// Usage of type predicates
function processUnknownValue(value: unknown): string {
  if (isString(value)) {
    // TypeScript knows value is string
    return value.toUpperCase();
  }

  if (isNumber(value)) {
    // TypeScript knows value is number
    return value.toFixed(2);
  }

  if (isBoolean(value)) {
    // TypeScript knows value is boolean
    return value ? "YES" : "NO";
  }

  return "Unknown type";
}

// Complex type predicate for objects
interface Person {
  name: string;
  age: number;
  email?: string;
}

function isPerson(value: unknown): value is Person {
  return (
    typeof value === "object" &&
    value !== null &&
    "name" in value &&
    "age" in value &&
    typeof (value as any).name === "string" &&
    typeof (value as any).age === "number" &&
    ((value as any).email === undefined ||
      typeof (value as any).email === "string")
  );
}
```

```
// Array type predicate
function isStringArray(value: unknown): value is string[] {
  return (
    Array.isArray(value) && value.every((item) => typeof item === "string")
  );
}

function isNumberArray(value: unknown): value is number[] {
  return (
    Array.isArray(value) && value.every((item) => typeof item === "number")
  );
}

// Generic type predicate
function isArrayOf<T>(
  value: unknown,
  itemGuard: (item: unknown) => item is T
): value is T[] {
  return Array.isArray(value) && value.every(itemGuard);
}

// Usage of generic type predicate
const data: unknown = ["hello", "world", "typescript"];

if (isArrayOf(data, isString)) {
  // TypeScript knows data is string[]
  console.log(data.map((s) => s.toUpperCase()));
}
```

Advanced Type Predicates

```
// Type predicate for discriminated unions
interface Circle {
  kind: "circle";
  radius: number;
}

interface Rectangle {
  kind: "rectangle";
  width: number;
  height: number;
}

interface Triangle {
  kind: "triangle";
  base: number;
  height: number;
}

type Shape = Circle | Rectangle | Triangle;
```

```
// Type predicates for each shape
function isCircle(shape: Shape): shape is Circle {
  return shape.kind === "circle";
}

function isRectangle(shape: Shape): shape is Rectangle {
  return shape.kind === "rectangle";
}

function isTriangle(shape: Shape): shape is Triangle {
  return shape.kind === "triangle";
}

// Calculate area using type predicates
function calculateArea(shape: Shape): number {
  if (isCircle(shape)) {
    return Math.PI * shape.radius * shape.radius;
  }

  if (isRectangle(shape)) {
    return shape.width * shape.height;
  }

  if (isTriangle(shape)) {
    return (shape.base * shape.height) / 2;
  }

  // TypeScript knows this is unreachable
  throw new Error("Unknown shape");
}

// Type predicate for nullable values
function isNotNull<T>(value: T | null): value is T {
  return value !== null;
}

function isNotUndefined<T>(value: T | undefined): value is T {
  return value !== undefined;
}

function isNotNullOrUndefined<T>(value: T | null | undefined): value is T {
  return value !== null && value !== undefined;
}

// Usage with array filtering
const mixedArray: (string | null | undefined)[] = [
  "hello",
  null,
  "world",
  undefined,
  "typescript",
];

const validStrings = mixedArray.filter(isNotNullOrUndefined);
```

```
// TypeScript knows validStrings is string[]

console.log(validStrings.map((s) => s.toUpperCase()));
```

Type Guards for API Responses

```
// API response type guards
interface User {
  id: number;
  name: string;
  email: string;
  createdAt: string;
}

interface Product {
  id: number;
  name: string;
  price: number;
  category: string;
}

// Type guard for User
function isUser(value: unknown): value is User {
  return (
    typeof value === "object" &&
    value !== null &&
    typeof (value as any).id === "number" &&
    typeof (value as any).name === "string" &&
    typeof (value as any).email === "string" &&
    typeof (value as any).createdAt === "string"
  );
}

// Type guard for Product
function isProduct(value: unknown): value is Product {
  return (
    typeof value === "object" &&
    value !== null &&
    typeof (value as any).id === "number" &&
    typeof (value as any).name === "string" &&
    typeof (value as any).price === "number" &&
    typeof (value as any).category === "string"
  );
}

// Generic API response type guard
interface ApiResponse<T> {
  success: boolean;
  data?: T;
  error?: string;
}
```

```
function isApiResponse<T>(
  value: unknown,
  dataGuard: (data: unknown) => data is T
): value is ApiResponse<T> {
  if (typeof value !== "object" || value === null) {
    return false;
  }

  const response = value as any;

  if (typeof response.success !== "boolean") {
    return false;
  }

  if (response.data !== undefined && !dataGuard(response.data)) {
    return false;
  }

  if (response.error !== undefined && typeof response.error !== "string") {
    return false;
  }

  return true;
}

// Usage
async function fetchUser(id: number): Promise<User> {
  const response = await fetch(`/api/users/${id}`);
  const data = await response.json();

  if (isApiResponse(data, isUser)) {
    if (data.success && data.data) {
      return data.data; // TypeScript knows this is User
    }
    throw new Error(data.error || "Unknown error");
  }

  throw new Error("Invalid API response format");
}
```

Discriminated Unions and Exhaustive Checking

Basic Discriminated Unions

```
// Discriminated union with literal types
interface LoadingState {
  status: "loading";
}

interface SuccessState {
  status: "success";
  data: any;
}

interface ErrorState {
  status: "error";
  error: string;
}

type AsyncState = LoadingState | SuccessState | ErrorState;

// Type guard using discriminant property
function handleAsyncState(state: AsyncState): string {
  switch (state.status) {
    case "loading":
      return "Loading...";

    case "success":
      // TypeScript knows state is SuccessState
      return `Success: ${JSON.stringify(state.data)}`;

    case "error":
      // TypeScript knows state is ErrorState
      return `Error: ${state.error}`;

    default:
      // Exhaustive check - TypeScript ensures all cases are handled
      const exhaustiveCheck: never = state;
      throw new Error(`Unhandled state: ${exhaustiveCheck}`);
  }
}

// Type predicate for discriminated union
function isSuccessState(state: AsyncState): state is SuccessState {
  return state.status === "success";
}

function isErrorState(state: AsyncState): state is ErrorState {
  return state.status === "error";
}

function isLoadingState(state: AsyncState): state is LoadingState {
  return state.status === "loading";
}
```

Complex Discriminated Unions

```
// Complex discriminated union for form validation
interface ValidField {
  type: "valid";
  value: string;
```

```
}

interface InvalidField {
  type: "invalid";
  value: string;
  errors: string[];
}

interface PendingField {
  type: "pending";
  value: string;
  validationPromise: Promise<boolean>;
}

type FieldState = ValidField | InvalidField | PendingField;

// Type guards for field states
function isValidField(field: FieldState): field is ValidField {
  return field.type === "valid";
}

function isInvalidField(field: FieldState): field is InvalidField {
  return field.type === "invalid";
}

function isPendingField(field: FieldState): field is PendingField {
  return field.type === "pending";
}

// Form validation handler
function renderField(field: FieldState): string {
  if (isValidField(field)) {
    return `✓ ${field.value}`;
  }

  if (isInvalidField(field)) {
    const errorList = field.errors.join(", ");
    return `✗ ${field.value} (${errorList})`;
  }

  if (isPendingField(field)) {
    return `⏳ ${field.value} (validating...)`;
  }

  // Exhaustive check
  const exhaustiveCheck: never = field;
  throw new Error(`Unhandled field type: ${exhaustiveCheck}`);
}

// Event system with discriminated unions
interface UserLoginEvent {
  type: "user:login";
  userId: string;
  timestamp: Date;
```

```
}

interface UserLogoutEvent {
  type: "user:logout";
  userId: string;
  timestamp: Date;
}

interface ProductCreatedEvent {
  type: "product:created";
  productId: number;
  name: string;
  timestamp: Date;
}

interface ProductUpdatedEvent {
  type: "product:updated";
  productId: number;
  changes: Record<string, any>;
  timestamp: Date;
}

type AppEvent =
  | UserLoginEvent
  | UserLogoutEvent
  | ProductCreatedEvent
  | ProductUpdatedEvent;

// Event handler with type guards
function handleEvent(event: AppEvent): void {
  switch (event.type) {
    case "user:login":
      console.log(`User ${event.userId} logged in at ${event.timestamp}`);
      break;

    case "user:logout":
      console.log(`User ${event.userId} logged out at ${event.timestamp}`);
      break;

    case "product:created":
      console.log(`Product "${event.name}" created with ID ${event.productId}`);
      break;

    case "product:updated":
      console.log(`Product ${event.productId} updated:`, event.changes);
      break;

    default:
      // Exhaustive check ensures all event types are handled
      const exhaustiveCheck: never = event;
      throw new Error(`Unhandled event type: ${exhaustiveCheck}`);
  }
}
```

Assertion Functions

Basic Assertion Functions

```
// Basic assertion function
function assert(condition: any, message?: string): asserts condition {
  if (!condition) {
    throw new Error(message || "Assertion failed");
  }
}

// Type assertion function
function assertIsString(value: unknown): asserts value is string {
  if (typeof value !== "string") {
    throw new Error(`Expected string, got ${typeof value}`);
  }
}

function assertIsNumber(value: unknown): asserts value is number {
  if (typeof value !== "number" || isNaN(value)) {
    throw new Error(`Expected number, got ${typeof value}`);
  }
}

// Usage of assertion functions
function processUserInput(input: unknown): string {
  assertIsString(input);
  // TypeScript now knows input is string
  return input.toUpperCase();
}

function calculateSquare(input: unknown): number {
  assertIsNumber(input);
  // TypeScript now knows input is number
  return input * input;
}

// Complex assertion function
function assertIsUser(value: unknown): asserts value is User {
  if (!isPerson(value)) {
    throw new Error("Value is not a valid User object");
  }
}

// Assertion function for non-null values
function assertNotNull<T>(value: T | null): asserts value is T {
  if (value === null) {
    throw new Error("Value is null");
  }
}
```

```
function assertNotUndefined<T>(value: T | undefined): asserts value is T {
  if (value === undefined) {
    throw new Error("Value is undefined");
  }
}

function assertNotNullOrUndefined<T>(
  value: T | null | undefined
): asserts value is T {
  if (value === null || value === undefined) {
    throw new Error("Value is null or undefined");
  }
}
```

Advanced Assertion Functions

```
// Assertion function for array validation
function assertIsArrayOf<T>(
  value: unknown,
  itemAssertion: (item: unknown) => asserts item is T
): asserts value is T[] {
  if (!Array.isArray(value)) {
    throw new Error("Value is not an array");
  }

  value.forEach((item, index) => {
    try {
      itemAssertion(item);
    } catch (error) {
      throw new Error(`Item at index ${index} is invalid: ${error.message}`);
    }
  });
}

// Assertion function for object properties
function assertHasProperty<T, K extends string>(
  obj: T,
  key: K
): asserts obj is T & Record<K, unknown> {
  if (typeof obj !== "object" || obj === null || !(key in obj)) {
    throw new Error(`Object does not have property '${key}'`);
  }
}

// Usage examples
function processApiData(data: unknown): void {
  assertIsArrayOf(data, assertIsUser);
  // TypeScript now knows data is User[]

  data.forEach((user) => {
    console.log(`Processing user: ${user.name}`);
  });
}

function processConfig(config: unknown): void {
  assertHasProperty(config, "apiUrl");
  assertHasProperty(config, "timeout");

  // TypeScript knows config has apiUrl and timeout properties
  assertIsString(config.apiUrl);
  assertIsNumber(config.timeout);

  console.log(`API URL: ${config.apiUrl}, Timeout: ${config.timeout}`);
}
```

Practical Examples

Form Validation with Type Guards

```
// Form field types
interface FormField {
  name: string;
  value: string;
  required: boolean;
}

interface EmailField extends FormField {
  type: "email";
}

interface PasswordField extends FormField {
  type: "password";
  minLength: number;
}

interface NumberField extends FormField {
  type: "number";
  min?: number;
  max?: number;
}

type Field = EmailField | PasswordField | NumberField;

// Type guards for field types
function isEmailField(field: Field): field is EmailField {
  return field.type === "email";
}

function isPasswordField(field: Field): field is PasswordField {
  return field.type === "password";
}
```

```
function isNumberField(field: Field): field is NumberField {
  return field.type === "number";
}

// Validation functions
function validateEmail(email: string): boolean {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return emailRegex.test(email);
}

function validatePassword(password: string, minLength: number): boolean {
  return password.length >= minLength;
}

function validateNumber(value: string, min?: number, max?: number): boolean {
  const num = parseFloat(value);
  if (isNaN(num)) return false;
  if (min !== undefined && num < min) return false;
  if (max !== undefined && num > max) return false;
  return true;
}

// Main validation function using type guards
function validateField(field: Field): { isValid: boolean; error?: string } {
  if (field.required && !field.value.trim()) {
    return { isValid: false, error: `${field.name} is required` };
  }

  if (isEmailField(field)) {
    if (field.value && !validateEmail(field.value)) {
      return { isValid: false, error: "Invalid email format" };
    }
  } else if (isPasswordField(field)) {
    if (field.value && !validatePassword(field.value, field.minLength)) {
      return {
        isValid: false,
        error: `Password must be at least ${field.minLength} characters`,
      };
    }
  } else if (isNumberField(field)) {
    if (field.value && !validateNumber(field.value, field.min, field.max)) {
      let error = "Invalid number";
      if (field.min !== undefined && field.max !== undefined) {
        error += ` (must be between ${field.min} and ${field.max})`;
      } else if (field.min !== undefined) {
        error += ` (must be at least ${field.min})`;
      } else if (field.max !== undefined) {
        error += ` (must be at most ${field.max})`;
      }
      return { isValid: false, error };
    }
  }
  return { isValid: true };
}
```

API Client with Type Guards

```
// API response types
interface ApiSuccessResponse<T> {
  success: true;
  data: T;
  timestamp: string;
}

interface ApiErrorResponse {
  success: false;
  error: {
    code: string;
    message: string;
    details?: any;
  };
  timestamp: string;
}

type ApiResponse<T> = ApiSuccessResponse<T> | ApiErrorResponse;

// Type guards for API responses
function isApiSuccessResponse<T>(
  response: ApiResponse<T>
): response is ApiSuccessResponse<T> {
  return response.success === true;
}

function isApiErrorResponse<T>(
  response: ApiResponse<T>
): response is ApiErrorResponse {
  return response.success === false;
}

// Generic API client
class ApiClient {
  constructor(private baseUrl: string) {}

  async request<T>(endpoint: string, options: RequestInit = {}): Promise<T> {
    const response = await fetch(`${this.baseUrl}${endpoint}`, {
      headers: {
        "Content-Type": "application/json",
        ...options.headers,
      },
      ...options,
    });

    const data: ApiResponse<T> = await response.json();
    if (isApiSuccessResponse(data)) {
      return data.data;
    }

    if (isApiErrorResponse(data)) {
      throw new Error(`API Error ${data.error.code}: ${data.error.message}`);
    }

    throw new Error("Invalid API response format");
  }

  async get<T>(endpoint: string): Promise<T> {
    return this.request<T>(endpoint, { method: "GET" });
  }

  async post<T>(endpoint: string, body: any): Promise<T> {
    return this.request<T>(endpoint, {
      method: "POST",
      body: JSON.stringify(body),
    });
  }
}

// Usage with type safety
const apiClient = new ApiClient("https://api.example.com");

async function fetchUser(id: number): Promise<User> {
  try {
    const user = await apiClient.get<User>(`/users/${id}`);
    return user;
  } catch (error) {
    console.error("Failed to fetch user:", error);
    throw error;
  }
}
```

Best Practices

✅ Good Practices

```
// Use type predicates for reusable type checking
function isValidEmail(value: string): boolean {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value);
}

function isUser(value: unknown): value is User {
  return (
    typeof value === "object" &&
    value !== null &&
    "id" in value &&
    "name" in value &&
    "email" in value &&
    typeof (value as any).id === "number" &&
    typeof (value as any).name === "string" &&
    isValidEmail((value as any).email)
  );
}

// Use discriminated unions for state management
type RequestState =
  | { status: "idle" }
  | { status: "loading" }
  | { status: "success"; data: any }
  | { status: "error"; error: string };

// Use assertion functions for runtime validation
function assertIsPositiveNumber(value: unknown): asserts value is number {
  if (typeof value !== "number" || value <= 0) {
    throw new Error("Expected positive number");
  }
}

// Use exhaustive checking with never
function handleState(state: RequestState): string {
  switch (state.status) {
    case "idle":
      return "Ready";
    case "loading":
      return "Loading...";
    case "success":
      return `Success: ${state.data}`;
    case "error":
      return `Error: ${state.error}`;
    default:
      const exhaustiveCheck: never = state;
      throw new Error(`Unhandled state: ${exhaustiveCheck}`);
  }
}
```

❌ Avoid

```
// Don't use type assertions without validation
function badTypeAssertion(value: unknown): User {
  return value as User; // Dangerous - no runtime check
}

// Don't ignore exhaustive checking
function incompleteHandler(state: RequestState): string {
  switch (state.status) {
    case "idle":
      return "Ready";
    case "loading":
      return "Loading...";
    // Missing success and error cases
  }
  return "Unknown"; // This hides missing cases
}

// Don't make type guards too complex
function overlyComplexGuard(value: unknown): value is ComplexType {
  // 50 lines of validation logic...
  // Better to break into smaller, focused guards
}

// Don't use any in type guards
function badGuard(value: any): value is User {
  return value.id && value.name; // Loses type safety
}
```

Summary Checklist

- □ Use built-in type guards (`typeof`, `instanceof`, `in`)
- □ Create custom type predicates with `value is Type`
- □ Use discriminated unions for complex state management
- □ Implement exhaustive checking with `never`
- □ Use assertion functions for runtime validation
- □ Combine type guards with control flow analysis
- □ Validate API responses with type guards
- □ Use type guards in array filtering and mapping
- □ Avoid unsafe type assertions
- □ Keep type guards focused and reusable

Next Steps

Now that you understand type guards and advanced type checking, let's explore advanced TypeScript features like keyof, typeof, and conditional types.

Continue to: Advanced TypeScript Features

Advanced TypeScript Features

> Explore advanced TypeScript features including keyof, typeof, conditional types, mapped types, and template literal types

keyof Operator

The `keyof` operator creates a union type of all property names of a given type.

Basic keyof Usage

```
// Basic keyof example
interface User {
  id: number;
  name: string;
  email: string;
  age: number;
}

// keyof creates a union of property names
type UserKeys = keyof User; // "id" | "name" | "email" | "age"

// Using keyof in function parameters
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const user: User = {
  id: 1,
  name: "John Doe",
  email: "john@example.com",
  age: 30,
};

const name = getProperty(user, "name"); // Type: string
const age = getProperty(user, "age"); // Type: number
// const invalid = getProperty(user, "invalid"); // Error: Argument of type '"invalid"' is not assignable

// keyof with arrays
const fruits = ["apple", "banana", "orange"] as const;
type FruitKeys = keyof typeof fruits; // "0" | "1" | "2" | "length" | "toString" | ...

// keyof with string literal types
type Colors = {
  red: string;
  green: string;
  blue: string;
};

type ColorKeys = keyof Colors; // "red" | "green" | "blue"
```

Advanced keyof Patterns

```
// keyof with generic constraints
function updateProperty<T, K extends keyof T>(obj: T, key: K, value: T[K]): T {
  return {
    ...obj,
    [key]: value,
  };
}

const updatedUser = updateProperty(user, "name", "Jane Doe");
const updatedAge = updateProperty(user, "age", 31);
// const invalid = updateProperty(user, "name", 123); // Error: Type 'number' is not assignable to type 'string'

// keyof with multiple objects
function copyProperty<T, U, K extends keyof T & keyof U>(
  source: T,
  target: U,
  key: K
): U {
  return {
    ...target,
    [key]: source[key],
  };
}

interface Employee {
  id: number;
  name: string;
  department: string;
}

const employee: Employee = {
  id: 1,
  name: "John",
  department: "Engineering",
};

// Can copy properties that exist in both types
const result = copyProperty(user, employee, "name"); // OK: both have 'name'
// const invalid = copyProperty(user, employee, "email"); // Error: 'email' doesn't exist in Employee

// keyof with conditional types
type StringPropertyNames<T> = {
  [K in keyof T]: T[K] extends string ? K : never;
}[keyof T];

type UserStringProps = StringPropertyNames<User>; // "name" | "email"

// keyof with filtering
type PickByType<T, U> = {
  [P in keyof T as T[P] extends U ? P : never]: T[P];
};

type UserStringProperties = PickByType<User, string>; // { name: string; email: string }
type UserNumberProperties = PickByType<User, number>; // { id: number; age: number }
```

typeof Operator

The `typeof` operator captures the type of a value or variable.

Basic typeof Usage

```
// typeof with variables
const message = "Hello, TypeScript!";
type MessageType = typeof message; // string

const count = 42;
type CountType = typeof count; // number

const isActive = true;
type IsActiveType = typeof isActive; // boolean

// typeof with objects
const config = {
  apiUrl: "https://api.example.com",
  timeout: 5000,
  retries: 3,
  debug: false,
};

type Config = typeof config;
// {
//   apiUrl: string;
//   timeout: number;
//   retries: number;
//   debug: boolean;
// }

// typeof with functions
function calculateArea(width: number, height: number): number {
  return width * height;
}

type CalculateAreaType = typeof calculateArea;
// (width: number, height: number) => number

// typeof with classes
class DatabaseConnection {
  constructor(public connectionString: string) {}

  connect(): void {
    console.log("Connecting to database...");
  }
}

type DatabaseConnectionType = typeof DatabaseConnection;
// typeof DatabaseConnection (constructor type)
```

```
type DatabaseInstanceType = InstanceType<typeof DatabaseConnection>;
// DatabaseConnection (instance type)
```

Advanced typeof Patterns

```
// typeof with const assertions
const themes = {
  light: {
    background: "#ffffff",
    text: "#000000",
  },
  dark: {
    background: "#000000",
    text: "#ffffff",
  },
} as const;

type Themes = typeof themes;
// {
//   readonly light: {
//     readonly background: "#ffffff";
//     readonly text: "#000000";
//   };
//   readonly dark: {
//     readonly background: "#000000";
//     readonly text: "#ffffff";
//   };
// }

type ThemeNames = keyof typeof themes; // "light" | "dark"
type ThemeColors = (typeof themes)["light"]; // { readonly background: "#ffffff"; readonly text: "#000000"; }

// typeof with arrays
const statusCodes = [200, 404, 500] as const;
type StatusCodes = typeof statusCodes; // readonly [200, 404, 500]
type StatusCode = (typeof statusCodes)[number]; // 200 | 404 | 500

// typeof with enum-like objects
const UserRole = {
  ADMIN: "admin",
  USER: "user",
  GUEST: "guest",
} as const;

type UserRoleType = typeof UserRole;
type UserRoleValue = (typeof UserRole)[keyof typeof UserRole]; // "admin" | "user" | "guest"

// typeof with complex nested structures
const apiEndpoints = {
  users: {
    list: "/api/users",
    create: "/api/users",
    update: (id: number) => `/api/users/${id}`,
    delete: (id: number) => `/api/users/${id}`,
  },
  products: {
    list: "/api/products",
    create: "/api/products",
    update: (id: number) => `/api/products/${id}`,
  },
} as const;

type ApiEndpoints = typeof apiEndpoints;
type UserEndpoints = typeof apiEndpoints.users;
type UpdateUserEndpoint = typeof apiEndpoints.users.update; // (id: number) => string
```

Conditional Types

Conditional types allow you to create types that depend on a condition.

Basic Conditional Types

```
// Basic conditional type syntax: T extends U ? X : Y
type IsString<T> = T extends string ? true : false;

type Test1 = IsString<string>; // true
type Test2 = IsString<number>; // false
type Test3 = IsString<"hello">; // true

// Conditional type with generic constraints
type ArrayElement<T> = T extends (infer U)[] ? U : never;

type StringArrayElement = ArrayElement<string[]>; // string
type NumberArrayElement = ArrayElement<number[]>; // number
type NotArrayElement = ArrayElement<string>; // never

// Conditional type for function return types
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

type FunctionReturn = ReturnType<() => string>; // string
type MethodReturn = ReturnType<(x: number) => boolean>; // boolean
type NotFunctionReturn = ReturnType<string>; // never

// Conditional type for promise unwrapping
type Awaited<T> = T extends Promise<infer U> ? U : T;

type PromiseString = Awaited<Promise<string>>; // string
type PromiseNumber = Awaited<Promise<number>>; // number
type NotPromise = Awaited<string>; // string
```

Advanced Conditional Types

```typescript
// Nested conditional types
type DeepAwaited<T> = T extends Promise<infer U> ? DeepAwaited<U> : T;

type NestedPromise = DeepAwaited<Promise<Promise<string>>>; // string

// Conditional types with union distribution
type ToArray<T> = T extends any ? T[] : never;

type UnionArrays = ToArray<string | number>; // string[] | number[]

// Non-distributive conditional types
type ToArrayNonDistributive<T> = [T] extends [any] ? T[] : never;

type NonDistributiveResult = ToArrayNonDistributive<string | number>; // (string | number)[]

// Conditional types for filtering
type NonNullable<T> = T extends null | undefined ? never : T;

type FilteredType = NonNullable<string | null | undefined>; // string

// Conditional types with multiple conditions
type TypeName<T> = T extends string
  ? "string"
  : T extends number
  ? "number"
  : T extends boolean
  ? "boolean"
  : T extends undefined
  ? "undefined"
  : T extends Function
  ? "function"
  : "object";

type StringTypeName = TypeName<string>; // "string"
type NumberTypeName = TypeName<42>; // "number"
type FunctionTypeName = TypeName<() => void>; // "function"

// Conditional types for object property extraction
type GetProperty<T, K> = K extends keyof T ? T[K] : never;

type UserName = GetProperty<User, "name">; // string
type UserInvalid = GetProperty<User, "invalid">; // never
```

Practical Conditional Types

```
// API response type transformation
type ApiResponse<T> = {
  data: T;
  status: number;
  message: string;
};

type UnwrapApiResponse<T> = T extends ApiResponse<infer U> ? U : T;

type UserData = UnwrapApiResponse<ApiResponse<User>>; // User
type DirectData = UnwrapApiResponse<string>; // string

// Function parameter extraction
type Parameters<T extends (...args: any) => any> = T extends (
  ...args: infer P
) => any
  ? P
  : never;

type CalcParams = Parameters<typeof calculateArea>; // [number, number]

// Object value types
type ValueOf<T> = T[keyof T];

type UserValues = ValueOf<User>; // string | number

// Required vs Optional property detection
type RequiredKeys<T> = {
  [K in keyof T]-?: {} extends Pick<T, K> ? never : K;
}[keyof T];

type OptionalKeys<T> = {
  [K in keyof T]-?: {} extends Pick<T, K> ? K : never;
}[keyof T];

interface PartialUser {
  id: number;
  name?: string;
  email?: string;
}

type RequiredUserKeys = RequiredKeys<PartialUser>; // "id"
type OptionalUserKeys = OptionalKeys<PartialUser>; // "name" | "email"
```

Mapped Types

Mapped types create new types by transforming properties of existing types.

Basic Mapped Types

```
// Basic mapped type syntax
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

type Partial<T> = {
  [P in keyof T]?: T[P];
};

type Required<T> = {
  [P in keyof T]-?: T[P];
};

// Usage
type ReadonlyUser = Readonly<User>;
type PartialUser = Partial<User>;
type RequiredUser = Required<PartialUser>;

// Custom mapped types
type Nullable<T> = {
  [P in keyof T]: T[P] | null;
};

type Stringify<T> = {
  [P in keyof T]: string;
};

type NullableUser = Nullable<User>; // All properties can be null
type StringifiedUser = Stringify<User>; // All properties are strings
```

Advanced Mapped Types

```
// Mapped types with key remapping
type Getters<T> = {
  [P in keyof T as `get${Capitalize<string & P>}`]: () => T[P];
};

type UserGetters = Getters<User>;
// {
//   getId: () => number;
//   getName: () => string;
//   getEmail: () => string;
//   getAge: () => number;
// }

// Mapped types with filtering
type PickByType<T, U> = {
  [P in keyof T as T[P] extends U ? P : never]: T[P];
};
```

```
type UserStringProps = PickByType<User, string>; // { name: string; email: string
}

// Mapped types with transformation
type EventHandlers<T> = {
  [P in keyof T as `on${Capitalize<string & P>}Change`]: (value: T[P]) => void;
};

type UserEventHandlers = EventHandlers<User>;
// {
//   onIdChange: (value: number) => void;
//   onNameChange: (value: string) => void;
//   onEmailChange: (value: string) => void;
//   onAgeChange: (value: number) => void;
// }

// Deep mapped types
type DeepReadonly<T> = {
  readonly [P in keyof T]: T[P] extends object ? DeepReadonly<T[P]> : T[P];
};

type DeepPartial<T> = {
  [P in keyof T]?: T[P] extends object ? DeepPartial<T[P]> : T[P];
};

interface NestedUser {
  id: number;
  profile: {
    name: string;
    settings: {
      theme: string;
      notifications: boolean;
    };
  };
}

type DeepReadonlyUser = DeepReadonly<NestedUser>;
type DeepPartialUser = DeepPartial<NestedUser>;
```

Template Literal Types

Template literal types allow you to create types using template literal syntax.

Basic Template Literal Types

```
// Basic template literal types
type Greeting = `Hello, ${string}!`;

type PersonalGreeting = `Hello, ${"John" | "Jane"}!`; // "Hello, John!" | "Hello,
Jane!"
```

```
// Template literals with unions
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";
type ApiEndpoint = `/api/${string}`;
type HttpRequest = `${HttpMethod} ${ApiEndpoint}`;

// Examples of HttpRequest:
// "GET /api/users" | "POST /api/users" | "PUT /api/users" | "DELETE /api/users" | ...

// Template literals with specific strings
type Color = "red" | "green" | "blue";
type Shade = "light" | "dark";
type ColorVariant = `${Shade}-${Color}`;
// "light-red" | "light-green" | "light-blue" | "dark-red" | "dark-green" | "dark-blue"
```

Advanced Template Literal Types

```
// Template literals with utility types
type EventName<T> = {
  [K in keyof T]: `${string & K}Changed`;
}[keyof T];

type UserEventNames = EventName<User>; // "idChanged" | "nameChanged" | "emailChanged" | "ageChanged"

// Template literals for CSS properties
type CSSUnit = "px" | "em" | "rem" | "%" | "vh" | "vw";
type CSSValue = `${number}${CSSUnit}`;

type Margin = CSSValue; // "10px" | "1em" | "100%" | etc.

// Template literals for database operations
type TableName = "users" | "products" | "orders";
type Operation = "select" | "insert" | "update" | "delete";
type SqlOperation = `${Operation}_${TableName}`;
// "select_users" | "insert_users" | "update_users" | "delete_users" | ...

// Template literals with conditional types
type AddPrefix<T, P extends string> = {
  [K in keyof T as K extends string ? `${P}${K}` : never]: T[K];
};

type PrefixedUser = AddPrefix<User, "user_">;
// {
//   user_id: number;
//   user_name: string;
//   user_email: string;
//   user_age: number;
// }
// Template literals for path building
type PathSegment = string | number;
type BuildPath<T extends readonly PathSegment[]> = T extends readonly [
  infer First,
  ...infer Rest
]
  ? First extends PathSegment
    ? Rest extends readonly PathSegment[]
      ? Rest["length"] extends 0
        ? `${First}`
        : `${First}/${BuildPath<Rest>}`
      : never
    : never
  : "";

type ApiPath = BuildPath<["api", "v1", "users", number]>; // "api/v1/users/number"
```

Practical Template Literal Examples

```
// Type-safe event system
type EventMap = {
  user: { id: number; name: string };
  product: { id: number; price: number };
  order: { id: number; total: number };
};

type EventType = keyof EventMap;
type EventAction = "created" | "updated" | "deleted";
type EventName = `${EventType}:${EventAction}`;

class TypedEventEmitter {
  private listeners: Map<EventName, Function[]> = new Map();

  on<T extends EventType, A extends EventAction>(
    event: `${T}:${A}`,
    listener: (data: EventMap[T]) => void
  ): void {
    const eventName = event as EventName;
    if (!this.listeners.has(eventName)) {
      this.listeners.set(eventName, []);
    }
    this.listeners.get(eventName)!.push(listener);
  }

  emit<T extends EventType, A extends EventAction>(
    event: `${T}:${A}`,
    data: EventMap[T]
  ): void {
    const eventName = event as EventName;
    const eventListeners = this.listeners.get(eventName);
    if (eventListeners) {
      eventListeners.forEach((listener) => listener(data));
    }
  }
}

// Usage
const emitter = new TypedEventEmitter();

emitter.on("user:created", (user) => {
  console.log(`User created: ${user.name}`);
});

emitter.emit("user:created", { id: 1, name: "John" });

// Type-safe CSS-in-JS
type CSSProperty =
  | "color"
  | "background-color"
  | "font-size"
  | "margin"
  | "padding"
  | "width"
  | "height";

type CSSValue = string | number;

type CSSRule = `${CSSProperty}: ${CSSValue}`;

type StyleObject = {
  [K in CSSProperty]?: CSSValue;
};

function createStyles(styles: StyleObject): string {
  return Object.entries(styles)
    .map(([property, value]) => `${property}: ${value}`)
    .join("; ");
}

const buttonStyles = createStyles({
  "background-color": "#007bff",
  color: "white",
  padding: "10px 20px",
  border: "none",
});
```

Utility Types

TypeScript provides many built-in utility types for common type transformations.

Built-in Utility Types

```
// Pick - Select specific properties
type UserSummary = Pick<User, "id" | "name">;
// { id: number; name: string }

// Omit - Exclude specific properties
type UserWithoutId = Omit<User, "id">;
// { name: string; email: string; age: number }

// Record - Create object type with specific keys and values
type UserRoles = Record<"admin" | "user" | "guest", string[]>;
// {
//   admin: string[];
//   user: string[];
//   guest: string[];
// }

// Exclude - Remove types from union
type NonStringTypes = Exclude<string | number | boolean, string>;
// number | boolean

// Extract - Keep only specific types from union
type StringTypes = Extract<string | number | boolean, string>;
// string

// ReturnType - Get function return type
function getUser(): User {
  return { id: 1, name: "John", email: "john@example.com", age: 30 };
}

type GetUserReturn = ReturnType<typeof getUser>; // User

// Parameters - Get function parameter types
type GetUserParams = Parameters<typeof getProperty>; // [T, K]

// ConstructorParameters - Get constructor parameter types
type DbConnectionParams = ConstructorParameters<typeof DatabaseConnection>; // [string]

// InstanceType - Get instance type of constructor
type DbInstance = InstanceType<typeof DatabaseConnection>; // DatabaseConnection
```

Custom Utility Types

```
// Deep Pick - Pick nested properties
type DeepPick<T, K extends string> = K extends `${infer Key}.${infer Rest}`
  ? Key extends keyof T
    ? { [P in Key]: DeepPick<T[Key], Rest> }
    : never
  : K extends keyof T
  ? { [P in K]: T[K] }
  : never;

interface NestedData {
  user: {
    profile: {
      name: string;
      age: number;
    };
    settings: {
      theme: string;
    };
  };
}

type UserName = DeepPick<NestedData, "user.profile.name">;
// { user: { profile: { name: string } } }

// Mutable - Remove readonly modifiers
type Mutable<T> = {
  -readonly [P in keyof T]: T[P];
};

type MutableReadonlyUser = Mutable<Readonly<User>>;
// { id: number; name: string; email: string; age: number }

// Optional to Required - Make specific properties required
type MakeRequired<T, K extends keyof T> = T & Required<Pick<T, K>>;

interface PartialProfile {
  name?: string;
  email?: string;
  age?: number;
}

type ProfileWithRequiredName = MakeRequired<PartialProfile, "name">;
// { name: string; email?: string; age?: number }

// Function property names
type FunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? K : never;
}[keyof T];

type NonFunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? never : K;
}[keyof T];

class Example {
  name: string = "";
  age: number = 0;
  getName(): string {
    return this.name;
  }
  setAge(age: number): void {
    this.age = age;
  }
}

type ExampleFunctions = FunctionPropertyNames<Example>; // "getName" | "setAge"
type ExampleProperties = NonFunctionPropertyNames<Example>; // "name" | "age"
```

Practical Examples

Type-Safe Configuration System

```
// Configuration schema
interface ConfigSchema {
  database: {
    host: string;
    port: number;
    username: string;
    password: string;
  };
  api: {
    baseUrl: string;
    timeout: number;
    retries: number;
  };
  features: {
    authentication: boolean;
    logging: boolean;
    analytics: boolean;
  };
}

// Type-safe configuration access
type ConfigPath<T, K extends string = ""> = {
  [P in keyof T]: T[P] extends object
    ? ConfigPath<T[P], K extends "" ? `${string & P}` : `${K}.${string & P}`>
    : K extends ""
    ? P
    : `${K}.${string & P}`;
}[keyof T];

type ConfigPaths = ConfigPath<ConfigSchema>;
// "database.host" | "database.port" | "database.username" | "database.password" |
// "api.baseUrl" | "api.timeout" | "api.retries" |
// "features.authentication" | "features.logging" | "features.analytics"

// Get nested value type
type GetConfigValue<
  T,
  P extends string
> = P extends `${infer Key}.${infer Rest}`
  ? Key extends keyof T
    ? GetConfigValue<T[Key], Rest>
    : never
  : P extends keyof T
  ? T[P]
  : never;

class ConfigManager {
  constructor(private config: ConfigSchema) {}

  get<P extends ConfigPaths>(path: P): GetConfigValue<ConfigSchema, P> {
    const keys = path.split(".");
    let value: any = this.config;

    for (const key of keys) {
      value = value[key];
    }

    return value;
  }
}

// Usage
const config = new ConfigManager({
  database: {
    host: "localhost",
    port: 5432,
    username: "admin",
    password: "secret",
  },
  api: {
    baseUrl: "https://api.example.com",
    timeout: 5000,
    retries: 3,
  },
  features: {
    authentication: true,
    logging: false,
    analytics: true,
  },
});

const dbHost = config.get("database.host"); // Type: string
const apiTimeout = config.get("api.timeout"); // Type: number
const authEnabled = config.get("features.authentication"); // Type: boolean
```

Advanced Form Validation

```
// Form field types
type FieldType =
  | "text"
  | "email"
  | "password"
```

```
    | "number"
    | "select"
    | "checkbox";

  interface BaseField {
    type: FieldType;
    label: string;
    required?: boolean;
  }

  interface TextField extends BaseField {
    type: "text" | "email" | "password";
    minLength?: number;
    maxLength?: number;
    pattern?: RegExp;
  }

  interface NumberField extends BaseField {
    type: "number";
    min?: number;
    max?: number;
    step?: number;
  }

  interface SelectField extends BaseField {
    type: "select";
    options: { value: string; label: string }[];
  }

  interface CheckboxField extends BaseField {
    type: "checkbox";
  }

  type Field = TextField | NumberField | SelectField | CheckboxField;

  // Form schema type
  type FormSchema = Record<string, Field>;

  // Extract form data type from schema
  type FormData<T extends FormSchema> = {
    [K in keyof T]: T[K] extends { type: "checkbox" }
      ? boolean
      : T[K] extends { type: "number" }
      ? number
      : string;
  };

  // Validation result type
  type ValidationResult<T extends FormSchema> = {
    [K in keyof T]?: string[];
  };

  // Form validator
  class FormValidator<T extends FormSchema> {
    constructor(private schema: T) {}

    validate(data: Partial<FormData<T>>): ValidationResult<T> {
      const errors: ValidationResult<T> = {};

      for (const [fieldName, field] of Object.entries(this.schema)) {
        const value = data[fieldName as keyof T];
        const fieldErrors: string[] = [];

        // Required validation
        if (
          field.required &&
          (value === undefined || value === null || value === "")
        ) {
          fieldErrors.push(`${field.label} is required`);
          continue;
        }

        if (value !== undefined && value !== null && value !== "") {
          // Type-specific validation
          if (field.type === "email" && typeof value === "string") {
            const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
            if (!emailRegex.test(value)) {
              fieldErrors.push("Invalid email format");
            }
          }

          if (
            field.type === "text" ||
            field.type === "email" ||
            field.type === "password"
          ) {
            const textField = field as TextField;
            const stringValue = value as string;

            if (textField.minLength && stringValue.length < textField.minLength) {
              fieldErrors.push(`Minimum length is ${textField.minLength}`);
            }

            if (textField.maxLength && stringValue.length > textField.maxLength) {
              fieldErrors.push(`Maximum length is ${textField.maxLength}`);
            }

            if (textField.pattern && !textField.pattern.test(stringValue)) {
              fieldErrors.push("Invalid format");
            }
          }

          if (field.type === "number") {
            const numberField = field as NumberField;
            const numberValue = value as number;

            if (numberField.min !== undefined && numberValue < numberField.min) {
              fieldErrors.push(`Minimum value is ${numberField.min}`);
            }

            if (numberField.max !== undefined && numberValue > numberField.max) {
              fieldErrors.push(`Maximum value is ${numberField.max}`);
            }
          }
        }

        if (fieldErrors.length > 0) {
          errors[fieldName as keyof T] = fieldErrors;
        }
      }

      return errors;
    }
  }

  // Usage
  const userFormSchema = {
    name: {
      type: "text" as const,
      label: "Full Name",
      required: true,
      minLength: 2,
      maxLength: 50,
    },
    email: {
      type: "email" as const,
      label: "Email Address",
      required: true,
    },
    age: {
      type: "number" as const,
      label: "Age",
      min: 18,
      max: 120,
    },
    newsletter: {
      type: "checkbox" as const,
      label: "Subscribe to newsletter",
    },
  };

  type UserFormData = FormData<typeof userFormSchema>;
  // {
  //   name: string;
  //   email: string;
  //   age: number;
  //   newsletter: boolean;
  // }

  const validator = new FormValidator(userFormSchema);

  const formData: Partial<UserFormData> = {
    name: "John Doe",
    email: "invalid-email",
    age: 15,
    newsletter: true,
  };

  const validationErrors = validator.validate(formData);
  console.log(validationErrors);
  // {
  //   email: ['Invalid email format'],
  //   age: ['Minimum value is 18']
  // }
```

Best Practices

✅ Good Practices

```
// Use meaningful names for type parameters
type ApiResponse<TData, TError = string> = {
  data?: TData;
  error?: TError;
  success: boolean;
};

// Use conditional types for complex type logic
type NonNullable<T> = T extends null | undefined ? never : T;

// Use mapped types for systematic transformations
type EventHandlers<T> = {
  [K in keyof T as `on${Capitalize<string & K>}Change`]: (value: T[K]) => void;
};

// Use template literal types for type-safe string manipulation
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";
type ApiEndpoint = `/api/${string}`;
type ApiCall = `${HttpMethod} ${ApiEndpoint}`;

// Use utility types to avoid repetition
type CreateUserRequest = Omit<User, "id">;
type UpdateUserRequest = Partial<Omit<User, "id">>;
```

❌ Avoid

```
// Don't overuse complex type manipulations
type OverlyComplex<T> = {
  [K in keyof T as T[K] extends Function
    ? never
    : K extends `${infer Prefix}_${infer Suffix}`
    ? `${Prefix}${Capitalize<Suffix>}`
    : K]: T[K] extends object ? OverlyComplex<T[K]> : T[K];
}; // Too complex, hard to understand

// Don't use any in advanced types
type BadConditional<T> = T extends any ? any : never; // Defeats the purpose

// Don't create overly nested conditional types
type TooNested<T> = T extends A
  ? T extends B
    ? T extends C
      ? T extends D
        ? E
        : F
      : G
    : H
  : I; // Hard to read and maintain

// Don't ignore type safety with assertions
function unsafeTypeManipulation<T>(value: unknown): T {
  return value as T; // Dangerous
}
```

Summary Checklist

- □ Use `keyof` to create unions of property names
- □ Use `typeof` to capture types from values
- □ Create conditional types with `T extends U ? X : Y`
- □ Use `infer` to extract types in conditional types
- □ Create mapped types for systematic transformations
- □ Use template literal types for string manipulation
- □ Leverage built-in utility types (`Pick`, `Omit`, `Record`, etc.)
- □ Create custom utility types for common patterns
- □ Use distributive conditional types appropriately
- □ Combine advanced features for complex type logic

Next Steps

Now that you understand advanced TypeScript features, let's explore modules, namespaces, and code organization patterns.

Continue to: Modules and Namespaces

Modules and Namespaces

> Learn how to organize and structure TypeScript code using modules and namespaces for better maintainability and scalability

ES6 Modules in TypeScript

TypeScript fully supports ES6 modules, which are the standard way to organize code in modern JavaScript and TypeScript applications.

Basic Module Exports and Imports

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

export function subtract(a: number, b: number): number {
  return a - b;
}

export const PI = 3.14159;

export interface Calculator {
  add(a: number, b: number): number;
  subtract(a: number, b: number): number;
}

// logger.ts - Default export
export default class Logger {
  private prefix: string;

  constructor(prefix: string = "LOG") {
    this.prefix = prefix;
  }

  log(message: string): void {
    console.log(`[${this.prefix}] ${message}`);
  }

  error(message: string): void {
    console.error(`[${this.prefix}] ERROR: ${message}`);
  }
}

// utils.ts - Mixed exports
export function formatDate(date: Date): string {
  return date.toISOString().split("T")[0];
}

export function generateId(): string {
  return Math.random().toString(36).substr(2, 9);
}

const DEFAULT_CONFIG = {
  timeout: 5000,
  retries: 3,
};
```

```
export { DEFAULT_CONFIG };
export { DEFAULT_CONFIG as Config }; // Export with alias

// Re-export from another module
export { add, subtract } from "./math";
```

Importing Modules

```
// app.ts - Various import patterns

// Named imports
import { add, subtract, PI } from "./math";
import { formatDate, generateId } from "./utils";

// Default import
import Logger from "./logger";

// Import with alias
import { Calculator as MathCalculator } from "./math";
import { DEFAULT_CONFIG as AppConfig } from "./utils";

// Import entire module as namespace
import * as MathUtils from "./math";
import * as Utils from "./utils";

// Side-effect import (runs module code without importing anything)
import "./polyfills";

// Dynamic imports (returns Promise)
async function loadMath() {
  const mathModule = await import("./math");
  return mathModule.add(5, 3);
}

// Usage examples
const logger = new Logger("APP");
logger.log("Application started");

const result = add(10, 5);
const today = formatDate(new Date());
const id = generateId();

// Using namespace imports
const sum = MathUtils.add(1, 2);
const difference = MathUtils.subtract(5, 3);
const formattedDate = Utils.formatDate(new Date());

// Using aliased imports
class BasicCalculator implements MathCalculator {
  add(a: number, b: number): number {
    return a + b;
  }

  subtract(a: number, b: number): number {
    return a - b;
  }
}
```

Module Resolution

```
// Relative imports
import { UserService } from "./services/user-service"; // Same directory
import { ApiClient } from "../api/client"; // Parent directory
import { Config } from "../../config/app-config"; // Multiple levels up

// Non-relative imports (node_modules or paths mapping)
import express from "express"; // npm package
import { Observable } from "rxjs"; // npm package
import { Component } from "@angular/core"; // scoped package

// Import with file extensions (sometimes required)
import { helper } from "./helper.js";
import data from "./data.json";

// Import types only (TypeScript 3.8+)
import type { User } from "./types/user";
import type { ApiResponse } from "./api/types";

// Import both value and type
import { type User, createUser } from "./user";
```

Advanced Module Patterns

Barrel Exports

```
// types/index.ts - Barrel file for types
export type { User } from "./user";
export type { Product } from "./product";
export type { Order } from "./order";
export type { ApiResponse, ApiError } from "./api";

// services/index.ts - Barrel file for services
export { UserService } from "./user-service";
export { ProductService } from "./product-service";
export { OrderService } from "./order-service";
export { ApiService } from "./api-service";

// utils/index.ts - Barrel file for utilities
export * from "./date-utils";
export * from "./string-utils";
export * from "./validation-utils";
export { default as Logger } from "./logger";

// Now you can import from the barrel
import { User, Product, ApiResponse } from "./types";
import { UserService, ProductService } from "./services";
import { formatDate, validateEmail, Logger } from "./utils";
```

Module Augmentation

```
// Extending existing modules

// global.d.ts - Augmenting global scope
declare global {
  interface Window {
    myApp: {
      version: string;
      config: Record<string, any>;
    };
  }

  namespace NodeJS {
    interface ProcessEnv {
      NODE_ENV: "development" | "production" | "test";
      DATABASE_URL: string;
      API_KEY: string;
    }
  }
}

// express.d.ts - Augmenting Express Request
import { User } from "./types/user";

declare module "express-serve-static-core" {
  interface Request {
    user?: User;
    requestId: string;
  }
}

// lodash-extensions.ts - Adding methods to existing library
import _ from "lodash";

declare module "lodash" {
  interface LoDashStatic {
    customMethod(value: any): boolean;
  }
}

_.customMethod = function (value: any): boolean {
  return typeof value === "string" && value.length > 0;
};

// Usage
const isValid = _.customMethod("hello"); // TypeScript knows about this method
```

Conditional Module Loading

```
// feature-loader.ts
interface FeatureModule {
  initialize(): void;
  cleanup(): void;
}

class FeatureLoader {
  private loadedFeatures = new Map<string, FeatureModule>();

  async loadFeature(featureName: string): Promise<FeatureModule | null> {
    if (this.loadedFeatures.has(featureName)) {
      return this.loadedFeatures.get(featureName)!;
    }

    try {
      let module: FeatureModule;

      switch (featureName) {
        case "analytics":
          module = await import("./features/analytics");
          break;
        case "chat":
          module = await import("./features/chat");
          break;
        case "notifications":
          module = await import("./features/notifications");
          break;
        default:
          console.warn(`Unknown feature: ${featureName}`);
          return null;
      }

      module.initialize();
      this.loadedFeatures.set(featureName, module);
      return module;
    } catch (error) {
      console.error(`Failed to load feature ${featureName}:`, error);
      return null;
    }
  }

  unloadFeature(featureName: string): void {
    const feature = this.loadedFeatures.get(featureName);
    if (feature) {
      feature.cleanup();
      this.loadedFeatures.delete(featureName);
    }
  }
}

// features/analytics.ts
export function initialize(): void {
  console.log("Analytics feature initialized");
  // Setup analytics tracking
}

export function cleanup(): void {
  console.log("Analytics feature cleaned up");
  // Cleanup analytics resources
}

// Usage
const featureLoader = new FeatureLoader();

// Load features conditionally
if (process.env.NODE_ENV === "production") {
  await featureLoader.loadFeature("analytics");
}

if (userHasPremium) {
  await featureLoader.loadFeature("chat");
}
```

Namespaces

While modules are preferred in modern TypeScript, namespaces can still be useful for organizing code within a single file or for library definitions.

Basic Namespace Usage

```
// geometry.ts
namespace Geometry {
  export interface Point {
    x: number;
    y: number;
  }

  export interface Rectangle {
    topLeft: Point;
    bottomRight: Point;
  }

  export function distance(p1: Point, p2: Point): number {
    const dx = p2.x - p1.x;
    const dy = p2.y - p1.y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  export function area(rect: Rectangle): number {
    const width = rect.bottomRight.x - rect.topLeft.x;
    const height = rect.bottomRight.y - rect.topLeft.y;
    return width * height;
  }

  export namespace Circle {
    export interface CircleShape {
      center: Point;
      radius: number;
    }

    export function area(circle: CircleShape): number {
      return Math.PI * circle.radius * circle.radius;
    }

    export function circumference(circle: CircleShape): number {
      return 2 * Math.PI * circle.radius;
    }
  }
}

// Usage
const point1: Geometry.Point = { x: 0, y: 0 };
const point2: Geometry.Point = { x: 3, y: 4 };
const dist = Geometry.distance(point1, point2);

const rectangle: Geometry.Rectangle = {
  topLeft: { x: 0, y: 0 },
  bottomRight: { x: 10, y: 5 },
};
const rectArea = Geometry.area(rectangle);

const circle: Geometry.Circle.CircleShape = {
  center: { x: 0, y: 0 },
  radius: 5,
};
const circleArea = Geometry.Circle.area(circle);
```

Namespace Merging

```
// Multiple namespace declarations with the same name are merged
namespace Utils {
  export function formatDate(date: Date): string {
    return date.toISOString().split("T")[0];
  }
}
```

```
namespace Utils {
  export function formatTime(date: Date): string {
    return date.toTimeString().split(" ")[0];
  }
}

namespace Utils {
  export interface Config {
    dateFormat: string;
    timeFormat: string;
  }

  export const defaultConfig: Config = {
    dateFormat: "YYYY-MM-DD",
    timeFormat: "HH:mm:ss",
  };
}

// All declarations are merged into one namespace
const formattedDate = Utils.formatDate(new Date());
const formattedTime = Utils.formatTime(new Date());
const config = Utils.defaultConfig;
```

Namespace Aliases

```
// Long namespace names can be aliased
namespace VeryLongCompanyName {
  export namespace ProductManagement {
    export namespace InventorySystem {
      export interface Product {
        id: string;
        name: string;
        price: number;
      }

      export function createProduct(name: string, price: number): Product {
        return {
          id: Math.random().toString(36),
          name,
          price,
        };
      }
    }
  }
}

// Create alias for easier access
import Inventory = VeryLongCompanyName.ProductManagement.InventorySystem;

// Now use the shorter alias
const product: Inventory.Product = Inventory.createProduct("Widget", 29.99);
```

Module vs Namespace Guidelines

When to Use Modules

```
// ✅ Use modules for:

// 1. Separate files with related functionality
// user-service.ts
export class UserService {
  async getUser(id: string): Promise<User> {
    // Implementation
  }
}

// 2. Third-party library integration
// api-client.ts
import axios from "axios";

export class ApiClient {
  constructor(private baseURL: string) {}

  async get<T>(endpoint: string): Promise<T> {
    const response = await axios.get(`${this.baseURL}${endpoint}`);
    return response.data;
  }
}

// 3. Code that needs to be tree-shaken
// utils.ts
export function debounce<T extends (...args: any[]) => any>(
  func: T,
  wait: number
): T {
  // Implementation
}

export function throttle<T extends (...args: any[]) => any>(
  func: T,
  limit: number
): T {
  // Implementation
}

// Only import what you need
import { debounce } from "./utils"; // throttle is not included in bundle
```

When to Use Namespaces

```
// ✅ Use namespaces for:

// 1. Organizing related types and functions in a single file
namespace ValidationRules {
  export interface Rule<T> {
    validate(value: T): boolean;
    message: string;
  }

  export const required: Rule<any> = {
    validate: (value) => value != null && value !== "",
    message: "This field is required",
  };

  export const email: Rule<string> = {
    validate: (value) => /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value),
    message: "Invalid email format",
  };

  export function minLength(min: number): Rule<string> {
    return {
      validate: (value) => value.length >= min,
      message: `Minimum length is ${min}`,
    };
  }
}

// 2. Library type definitions
declare namespace MyLibrary {
  interface Config {
    apiKey: string;
    baseUrl: string;
  }

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

  function initialize(config: Config): void;
  function getUser(id: string): Promise<User>;
}

// 3. Global augmentations
declare global {
  namespace Express {
    interface Request {
      user?: User;
    }
  }
}
```

Practical Examples

Plugin System with Modules

```typescript
// plugin-system.ts
export interface Plugin {
  name: string;
  version: string;
  initialize(app: Application): void;
  destroy(): void;
}

export interface Application {
  registerRoute(path: string, handler: Function): void;
  registerMiddleware(middleware: Function): void;
  getConfig(key: string): any;
}

export class PluginManager {
  private plugins = new Map<string, Plugin>();
  private app: Application;

  constructor(app: Application) {
    this.app = app;
  }

  async loadPlugin(pluginPath: string): Promise<void> {
    try {
      const pluginModule = await import(pluginPath);
      const plugin: Plugin = pluginModule.default || pluginModule;

      if (this.plugins.has(plugin.name)) {
        throw new Error(`Plugin ${plugin.name} is already loaded`);
      }

      plugin.initialize(this.app);
      this.plugins.set(plugin.name, plugin);

      console.log(`Plugin ${plugin.name} v${plugin.version} loaded`);
    } catch (error) {
      console.error(`Failed to load plugin from ${pluginPath}:`, error);
    }
  }

  unloadPlugin(name: string): void {
    const plugin = this.plugins.get(name);
    if (plugin) {
      plugin.destroy();
      this.plugins.delete(name);
      console.log(`Plugin ${name} unloaded`);
    }
  }
  getLoadedPlugins(): string[] {
    return Array.from(this.plugins.keys());
  }
}

// plugins/auth-plugin.ts
import { Plugin, Application } from "../plugin-system";

class AuthPlugin implements Plugin {
  name = "auth";
  version = "1.0.0";

  initialize(app: Application): void {
    // Register authentication middleware
    app.registerMiddleware((req: any, res: any, next: any) => {
      // Authentication logic
      next();
    });

    // Register auth routes
    app.registerRoute("/login", this.handleLogin);
    app.registerRoute("/logout", this.handleLogout);
  }

  destroy(): void {
    // Cleanup resources
    console.log("Auth plugin destroyed");
  }

  private handleLogin(req: any, res: any): void {
    // Login logic
  }

  private handleLogout(req: any, res: any): void {
    // Logout logic
  }
}

export default AuthPlugin;

// plugins/analytics-plugin.ts
import { Plugin, Application } from "../plugin-system";

class AnalyticsPlugin implements Plugin {
  name = "analytics";
  version = "2.1.0";
  private trackingId: string;

  initialize(app: Application): void {
    this.trackingId = app.getConfig("analytics.trackingId");

    // Register analytics middleware
    app.registerMiddleware((req: any, res: any, next: any) => {
      this.trackPageView(req.path);
      next();
    });
  }

  destroy(): void {
    // Send final analytics data
    this.flush();
  }

  private trackPageView(path: string): void {
    // Track page view
    console.log(`Tracking page view: ${path}`);
  }

  private flush(): void {
    // Send remaining analytics data
    console.log("Analytics data flushed");
  }
}

export default AnalyticsPlugin;

// app.ts
import { PluginManager, Application } from "./plugin-system";

class MyApplication implements Application {
  private routes = new Map<string, Function>();
  private middlewares: Function[] = [];
  private config = new Map<string, any>();

  constructor() {
    this.config.set("analytics.trackingId", "GA-123456789");
  }

  registerRoute(path: string, handler: Function): void {
    this.routes.set(path, handler);
  }

  registerMiddleware(middleware: Function): void {
    this.middlewares.push(middleware);
  }

  getConfig(key: string): any {
    return this.config.get(key);
  }
}

// Usage
const app = new MyApplication();
const pluginManager = new PluginManager(app);

// Load plugins dynamically
await pluginManager.loadPlugin("./plugins/auth-plugin");
await pluginManager.loadPlugin("./plugins/analytics-plugin");

console.log("Loaded plugins:", pluginManager.getLoadedPlugins());
```

Configuration Management System

```
// config/types.ts
export interface DatabaseConfig {
  host: string;
  port: number;
  username: string;
  password: string;
  database: string;
  ssl: boolean;
}

export interface ApiConfig {
  baseUrl: string;
  timeout: number;
  retries: number;
  apiKey: string;
}

export interface LoggingConfig {
  level: "debug" | "info" | "warn" | "error";
  format: "json" | "text";
  outputs: ("console" | "file" | "remote")[];
}

export interface AppConfig {
  environment: "development" | "staging" | "production";
  port: number;
  database: DatabaseConfig;
  api: ApiConfig;
  logging: LoggingConfig;
  features: {
    authentication: boolean;
    analytics: boolean;
    caching: boolean;
  };
}

// config/loaders.ts
import { AppConfig } from "./types";

export interface ConfigLoader {
  load(): Promise<Partial<AppConfig>>;
}

export class EnvironmentConfigLoader implements ConfigLoader {
  async load(): Promise<Partial<AppConfig>> {
    return {
      environment: (process.env.NODE_ENV as any) || "development",
      port: parseInt(process.env.PORT || "3000"),
      database: {
        host: process.env.DB_HOST || "localhost",
        port: parseInt(process.env.DB_PORT || "5432"),
        username: process.env.DB_USERNAME || "user",
        password: process.env.DB_PASSWORD || "password",
        database: process.env.DB_NAME || "myapp",
        ssl: process.env.DB_SSL === "true",
      },
      api: {
        baseUrl: process.env.API_BASE_URL || "http://localhost:3000",
        timeout: parseInt(process.env.API_TIMEOUT || "5000"),
        retries: parseInt(process.env.API_RETRIES || "3"),
        apiKey: process.env.API_KEY || "",
      },
    };
  }
}

export class FileConfigLoader implements ConfigLoader {
  constructor(private filePath: string) {}

  async load(): Promise<Partial<AppConfig>> {
    try {
      const fs = await import("fs/promises");
      const content = await fs.readFile(this.filePath, "utf-8");
      return JSON.parse(content);
    } catch (error) {
      console.warn(`Failed to load config from ${this.filePath}:`, error);
      return {};
    }
  }
}

export class RemoteConfigLoader implements ConfigLoader {
  constructor(private url: string, private apiKey: string) {}

  async load(): Promise<Partial<AppConfig>> {
    try {
      const response = await fetch(this.url, {
        headers: {
          Authorization: `Bearer ${this.apiKey}`,
          "Content-Type": "application/json",
        },
      });

      if (!response.ok) {
        throw new Error(`HTTP ${response.status}: ${response.statusText}`);
      }

      return await response.json();
    } catch (error) {
      console.warn(`Failed to load remote config from ${this.url}:`, error);
      return {};
    }
  }
}

// config/manager.ts
import { AppConfig } from "./types";
import { ConfigLoader } from "./loaders";

export class ConfigManager {
  private config: AppConfig;
  private loaders: ConfigLoader[] = [];

  constructor(private defaultConfig: AppConfig) {
    this.config = { ...defaultConfig };
  }

  addLoader(loader: ConfigLoader): void {
    this.loaders.push(loader);
  }

  async load(): Promise<void> {
    for (const loader of this.loaders) {
      try {
        const partialConfig = await loader.load();
        this.config = this.mergeConfig(this.config, partialConfig);
      } catch (error) {
        console.error("Config loader failed:", error);
      }
    }
  }

  get<K extends keyof AppConfig>(key: K): AppConfig[K] {
    return this.config[key];
  }

  getAll(): Readonly<AppConfig> {
    return Object.freeze({ ...this.config });
  }

  private mergeConfig(base: AppConfig, partial: Partial<AppConfig>): AppConfig {
    const result = { ...base };

    for (const [key, value] of Object.entries(partial)) {
      if (value !== undefined) {
        if (
          typeof value === "object" &&
          !Array.isArray(value) &&
          value !== null
        ) {
          (result as any)[key] = { ...(result as any)[key], ...value };
        } else {
          (result as any)[key] = value;
        }
      }
    }

    return result;
  }
}

// config/index.ts - Barrel export
export * from "./types";
export * from "./loaders";
export * from "./manager";

// Default configuration
export const defaultConfig: AppConfig = {
  environment: "development",
  port: 3000,
  database: {
    host: "localhost",
    port: 5432,
    username: "user",
    password: "password",
    database: "myapp",
    ssl: false,
  },
  api: {
    baseUrl: "http://localhost:3000",
    timeout: 5000,
    retries: 3,
    apiKey: "",
  },
  logging: {
    level: "info",
    format: "text",
    outputs: ["console"],
  },
  features: {
    authentication: true,
    analytics: false,
    caching: true,
  },
};

// app.ts - Usage
import {
  ConfigManager,
  EnvironmentConfigLoader,
  FileConfigLoader,
  RemoteConfigLoader,
  defaultConfig,
} from "./config";

const configManager = new ConfigManager(defaultConfig);
```

```
// Add multiple config sources
configManager.addLoader(new EnvironmentConfigLoader());
configManager.addLoader(new FileConfigLoader("./config.json"));

if (process.env.REMOTE_CONFIG_URL) {
  configManager.addLoader(
    new RemoteConfigLoader(
      process.env.REMOTE_CONFIG_URL,
      process.env.REMOTE_CONFIG_API_KEY || ""
    )
  );
}

// Load configuration
await configManager.load();

// Use configuration
const dbConfig = configManager.get("database");
const apiConfig = configManager.get("api");
const isAnalyticsEnabled = configManager.get("features").analytics;

console.log("Database host:", dbConfig.host);
console.log("API base URL:", apiConfig.baseUrl);
console.log("Analytics enabled:", isAnalyticsEnabled);
```

Best Practices

✅ Good Practices

```
// Use barrel exports for clean imports
// index.ts
export * from "./user-service";
export * from "./product-service";
export { default as Logger } from "./logger";

// Prefer named exports over default exports
export class UserService {
  /* ... */
}
export interface User {
  /* ... */
}
export const API_VERSION = "v1";

// Use type-only imports when possible
import type { User } from "./types";
import { createUser } from "./user-factory";

// Organize related functionality in modules
// user/
//   ├── types.ts
//   ├── service.ts
//   ├── repository.ts
//   ├── validation.ts
//   └── index.ts

// Use consistent naming conventions
export class UserService {} // PascalCase for classes
export interface UserConfig {} // PascalCase for interfaces
export const USER_ROLES = {} as const; // UPPER_SNAKE_CASE for constants
export function createUser() {} // camelCase for functions
```

❌ Avoid

```
// Don't use namespaces for code that could be modules
namespace UserManagement {
  // ❌ Should be separate modules
  export class UserService {}
  export class UserRepository {}
  export class UserValidator {}
}

// Don't mix default and named exports in the same module
export default class UserService {} // ❌
export class UserRepository {} // ❌ Inconsistent

// Don't create deep namespace hierarchies
namespace Company.Department.Team.Project.Module {
  // ❌ Too deep
  export function doSomething() {}
}

// Don't use relative imports for distant files
import { Utils } from "../../../../../../../utils"; // ❌ Hard to maintain

// Don't export everything
export * from "./internal-helpers"; // ❌ Exposes internal implementation
```

Summary Checklist

- □ Use ES6 modules for organizing code across files
- □ Prefer named exports over default exports
- □ Use barrel exports for clean import statements
- □ Implement proper module resolution strategies
- □ Use type-only imports when importing only types
- □ Organize related functionality into cohesive modules
- □ Use namespaces sparingly, mainly for type definitions
- □ Implement dynamic imports for code splitting
- □ Follow consistent naming conventions

- ☐ Avoid deep namespace hierarchies

Next Steps

Now that you understand modules and namespaces, let's explore error handling patterns and best practices in TypeScript.

Continue to: Error Handling in TypeScript

Error Handling in TypeScript

> Learn comprehensive error handling strategies, custom error types, and best practices for building robust TypeScript applications

Basic Error Handling

Try-Catch-Finally

```
// Basic try-catch structure
function parseJSON(jsonString: string): any {
  try {
    return JSON.parse(jsonString);
  } catch (error) {
    console.error("Failed to parse JSON:", error);
    return null;
  }
}

// Try-catch with finally
function processFile(filename: string): string | null {
  let fileHandle: any = null;

  try {
    fileHandle = openFile(filename);
    const content = readFile(fileHandle);
    return processContent(content);
  } catch (error) {
    console.error(`Error processing file ${filename}:`, error);
    return null;
  } finally {
    // Always executed, regardless of success or failure
    if (fileHandle) {
      closeFile(fileHandle);
    }
  }
}

// Type-safe error handling
function safeDivide(a: number, b: number): number {
  try {
    if (b === 0) {
      throw new Error("Division by zero is not allowed");
    }
    return a / b;
  } catch (error) {
    if (error instanceof Error) {
      console.error("Division error:", error.message);
    } else {
      console.error("Unknown error:", error);
    }
    throw error; // Re-throw if you want calling code to handle it
  }
}

// Multiple catch scenarios (TypeScript doesn't have multiple catch blocks)
function handleMultipleErrors(operation: string): void {
  try {
    performOperation(operation);
  } catch (error) {
    if (error instanceof TypeError) {
      console.error("Type error:", error.message);
    } else if (error instanceof RangeError) {
      console.error("Range error:", error.message);
    } else if (error instanceof Error) {
      console.error("General error:", error.message);
    } else {
      console.error("Unknown error:", error);
    }
  }
}

function performOperation(operation: string): void {
  // Implementation that might throw different types of errors
}

function openFile(filename: string): any {
  /* Implementation */
}
function readFile(handle: any): string {
  /* Implementation */
}
function closeFile(handle: any): void {
  /* Implementation */
}
function processContent(content: string): string {
  /* Implementation */
}
```

Custom Error Types

Creating Custom Error Classes

```
// Base custom error class
abstract class AppError extends Error {
  abstract readonly statusCode: number;
  abstract readonly isOperational: boolean;

  constructor(message: string, public readonly context?: Record<string, any>) {
    super(message);
    this.name = this.constructor.name;

    // Maintains proper stack trace for where our error was thrown (only available on V8)
    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, this.constructor);
    }
  }
}

// Specific error types
class ValidationError extends AppError {
  readonly statusCode = 400;
  readonly isOperational = true;

  constructor(
    message: string,
    public readonly field: string,
    public readonly value: any,
    context?: Record<string, any>
  ) {
    super(message, context);
  }
}

class NotFoundError extends AppError {
  readonly statusCode = 404;
  readonly isOperational = true;

  constructor(
    public readonly resource: string,
    public readonly id: string | number,
    context?: Record<string, any>
  ) {
    super(`${resource} with id ${id} not found`, context);
  }
}

class DatabaseError extends AppError {
  readonly statusCode = 500;
  readonly isOperational = false;

  constructor(
    message: string,
    public readonly query?: string,
    public readonly originalError?: Error,
    context?: Record<string, any>
  ) {
    super(message, context);
  }
}

class AuthenticationError extends AppError {
  readonly statusCode = 401;
  readonly isOperational = true;

  constructor(
    message: string = "Authentication failed",
    context?: Record<string, any>
  ) {
    super(message, context);
  }
}

class AuthorizationError extends AppError {
  readonly statusCode = 403;
  readonly isOperational = true;

  constructor(
    public readonly requiredPermission: string,
    public readonly userPermissions: string[],
    context?: Record<string, any>
  ) {
    super(`Access denied. Required permission: ${requiredPermission}`, context);
  }
}

// Business logic specific errors
class InsufficientFundsError extends AppError {
  readonly statusCode = 400;
  readonly isOperational = true;

  constructor(
    public readonly requestedAmount: number,
    public readonly availableAmount: number,
    context?: Record<string, any>
  ) {
    super(
      `Insufficient funds. Requested: ${requestedAmount}, Available:
${availableAmount}`,
      context
    );
  }
}

class RateLimitError extends AppError {
  readonly statusCode = 429;
  readonly isOperational = true;

  constructor(
    public readonly limit: number,
    public readonly resetTime: Date,
    context?: Record<string, any>
  ) {
    super(
      `Rate limit exceeded. Limit: ${limit}, Reset time:
${resetTime.toISOString()}`,
      context
    );
  }
}
```

Error Factory Pattern

```
// Error factory for consistent error creation
class ErrorFactory {
  static validation(
    field: string,
    value: any,
    message: string
  ): ValidationError {
    return new ValidationError(message, field, value, {
      timestamp: new Date().toISOString(),
      type: "validation",
    });
  }

  static notFound(resource: string, id: string | number): NotFoundError {
    return new NotFoundError(resource, id, {
      timestamp: new Date().toISOString(),
      type: "not_found",
    });
  }

  static database(
    message: string,
    query?: string,
    originalError?: Error
  ): DatabaseError {
    return new DatabaseError(message, query, originalError, {
      timestamp: new Date().toISOString(),
      type: "database",
    });
  }

  static authentication(message?: string): AuthenticationError {
    return new AuthenticationError(message, {
      timestamp: new Date().toISOString(),
      type: "authentication",
    });
  }
  static authorization(
    requiredPermission: string,
    userPermissions: string[]
  ): AuthorizationError {
    return new AuthorizationError(requiredPermission, userPermissions, {
      timestamp: new Date().toISOString(),
      type: "authorization",
    });
  }

  static rateLimit(limit: number, resetTime: Date): RateLimitError {
    return new RateLimitError(limit, resetTime, {
      timestamp: new Date().toISOString(),
      type: "rate_limit",
    });
  }
}

// Usage
try {
  validateEmail("invalid-email");
} catch (error) {
  throw ErrorFactory.validation(
    "email",
    "invalid-email",
    "Invalid email format"
  );
}

function validateEmail(email: string): void {
  if (!email.includes("@")) {
    throw new Error("Invalid email");
  }
}
```

Result Pattern

The Result pattern is an alternative to throwing exceptions, providing explicit error handling.

Basic Result Implementation

```
// Result type definition
type Result<T, E = Error> = Success<T> | Failure<E>;

class Success<T> {
  readonly isSuccess = true;
  readonly isFailure = false;

  constructor(public readonly value: T) {}

  map<U>(fn: (value: T) => U): Result<U, never> {
    return new Success(fn(this.value));
  }

  flatMap<U, F>(fn: (value: T) => Result<U, F>): Result<U, F> {
    return fn(this.value);
  }

  mapError<F>(_fn: (error: never) => F): Result<T, F> {
    return this as any;
  }

  unwrap(): T {
    return this.value;
  }

  unwrapOr(_defaultValue: T): T {
    return this.value;
  }
}

class Failure<E> {
  readonly isSuccess = false;
  readonly isFailure = true;

  constructor(public readonly error: E) {}

  map<U>(_fn: (value: never) => U): Result<U, E> {
    return this as any;
  }

  flatMap<U, F>(_fn: (value: never) => Result<U, F>): Result<U, E | F> {
    return this as any;
  }

  mapError<F>(fn: (error: E) => F): Result<never, F> {
    return new Failure(fn(this.error));
  }

  unwrap(): never {
    throw new Error("Called unwrap on a Failure");
  }

  unwrapOr<T>(defaultValue: T): T {
    return defaultValue;
  }
}

// Helper functions
function success<T>(value: T): Success<T> {
  return new Success(value);
}

function failure<E>(error: E): Failure<E> {
  return new Failure(error);
}

// Utility function to wrap try-catch in Result
function tryCatch<T>(fn: () => T): Result<T, Error> {
  try {
    return success(fn());
  } catch (error) {
    return failure(error instanceof Error ? error : new Error(String(error)));
  }
}

// Async version
async function tryAsync<T>(fn: () => Promise<T>): Promise<Result<T, Error>> {
  try {
    const value = await fn();
    return success(value);
  } catch (error) {
    return failure(error instanceof Error ? error : new Error(String(error)));
  }
}
```

Using the Result Pattern

```
// Example: User service with Result pattern
interface User {
  id: string;
  name: string;
  email: string;
  age: number;
}

class UserService {
  private users: Map<string, User> = new Map();

  createUser(userData: Omit<User, "id">): Result<User, ValidationError> {
    // Validate input
    const validationResult = this.validateUserData(userData);
    if (validationResult.isFailure) {
      return validationResult;
    }

    // Create user
    const user: User = {
      id: this.generateId(),
      ...userData,
    };

    this.users.set(user.id, user);
    return success(user);
  }
  getUserById(id: string): Result<User, NotFoundError> {
    const user = this.users.get(id);
    if (!user) {
      return failure(ErrorFactory.notFound("User", id));
    }
    return success(user);
  }

  updateUser(
    id: string,
    updates: Partial<Omit<User, "id">>
  ): Result<User, NotFoundError | ValidationError> {
    const userResult = this.getUserById(id);
    if (userResult.isFailure) {
      return userResult;
    }

    const user = userResult.value;
    const updatedUserData = { ...user, ...updates };

    // Validate updated data
    const validationResult = this.validateUserData(updatedUserData);
    if (validationResult.isFailure) {
      return validationResult;
    }

    const updatedUser = { ...user, ...updates };
    this.users.set(id, updatedUser);
    return success(updatedUser);
  }

  deleteUser(id: string): Result<void, NotFoundError> {
    if (!this.users.has(id)) {
      return failure(ErrorFactory.notFound("User", id));
    }

    this.users.delete(id);
    return success(undefined);
  }

  private validateUserData(
    userData: Omit<User, "id">
  ): Result<void, ValidationError> {
    if (!userData.name || userData.name.trim().length === 0) {
      return failure(
        ErrorFactory.validation("name", userData.name, "Name is required")
      );
    }

    if (!userData.email || !this.isValidEmail(userData.email)) {
      return failure(
        ErrorFactory.validation(
          "email",
          userData.email,
          "Valid email is required"
        )
      );
    }

    if (userData.age < 0 || userData.age > 150) {
      return failure(
        ErrorFactory.validation(
          "age",
          userData.age,
          "Age must be between 0 and 150"
        )
      );
    }

    return success(undefined);
  }

  private isValidEmail(email: string): boolean {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
  }

  private generateId(): string {
    return Math.random().toString(36).substr(2, 9);
  }
}

// Usage with Result pattern
const userService = new UserService();

// Creating a user
const createResult = userService.createUser({
  name: "John Doe",
  email: "john@example.com",
  age: 30,
});

if (createResult.isSuccess) {
  console.log("User created:", createResult.value);
} else {
  console.error("Failed to create user:", createResult.error.message);
}

// Chaining operations with flatMap
const result = userService
  .createUser({
    name: "Jane Doe",
    email: "jane@example.com",
    age: 25,
  })
  .flatMap((user) => userService.updateUser(user.id, { age: 26 }))
  .map((user) => ({ ...user, displayName: `${user.name} (${user.age})` }));
if (result.isSuccess) {
  console.log("Final result:", result.value);
} else {
  console.error("Operation failed:", result.error.message);
}
```

Async Error Handling

Promise-based Error Handling

```
// Basic async error handling
async function fetchUserData(userId: string): Promise<User> {
  try {
    const response = await fetch(`/api/users/${userId}`);

    if (!response.ok) {
      if (response.status === 404) {
        throw ErrorFactory.notFound("User", userId);
      } else if (response.status === 401) {
        throw ErrorFactory.authentication();
      } else {
        throw new Error(`HTTP ${response.status}: ${response.statusText}`);
      }
    }

    const userData = await response.json();
    return userData;
  } catch (error) {
    if (error instanceof AppError) {
      throw error; // Re-throw our custom errors
    } else if (error instanceof TypeError) {
      // Network errors often manifest as TypeErrors
      throw new Error("Network error: Unable to fetch user data");
    } else {
      throw new Error(`Unexpected error: ${error}`);
    }
  }
}

// Async Result pattern
async function fetchUserDataSafe(
  userId: string
): Promise<Result<User, AppError>> {
  try {
    const response = await fetch(`/api/users/${userId}`);

    if (!response.ok) {
      if (response.status === 404) {
        return failure(ErrorFactory.notFound("User", userId));
      } else if (response.status === 401) {
        return failure(ErrorFactory.authentication());
      } else {
        return failure(
          new AppError(`HTTP ${response.status}: ${response.statusText}`) as any
        );
      }
    }

    const userData = await response.json();
    return success(userData);
  } catch (error) {
    if (error instanceof AppError) {
      return failure(error);
    } else {
      return failure(new AppError(`Unexpected error: ${error}`) as any);
    }
  }
}

// Multiple async operations with error handling
async function processUserWorkflow(
  userId: string
): Promise<Result<string, AppError>> {
  const userResult = await fetchUserDataSafe(userId);
  if (userResult.isFailure) {
    return userResult as any;
  }

  const user = userResult.value;

  // Validate user permissions
  const permissionsResult = await checkUserPermissions(user.id);
  if (permissionsResult.isFailure) {
    return permissionsResult as any;
  }

  // Process user data
  const processResult = await processUserData(user);
  if (processResult.isFailure) {
    return processResult as any;
  }

  return success(`User ${user.name} processed successfully`);
}

async function checkUserPermissions(
  userId: string
): Promise<Result<string[], AuthorizationError>> {
  // Implementation
  return success(["read", "write"]);
}

async function processUserData(user: User): Promise<Result<void, Error>> {
  // Implementation
  return success(undefined);
}
```

Parallel Error Handling

```
// Handle multiple async operations
async function fetchMultipleUsers(
  userIds: string[]
): Promise<Result<User[], AppError[]>> {
  const results = await Promise.allSettled(
    userIds.map((id) => fetchUserDataSafe(id))
  );

  const users: User[] = [];
  const errors: AppError[] = [];

  for (const result of results) {
    if (result.status === "fulfilled") {
      if (result.value.isSuccess) {
        users.push(result.value.value);
      } else {
        errors.push(result.value.error);
      }
    } else {
      errors.push(new AppError(`Promise rejected: ${result.reason}`) as any);
    }
  }

  if (errors.length > 0) {
    return failure(errors);
  }

  return success(users);
}

// Retry mechanism with exponential backoff
async function withRetry<T>(
  operation: () => Promise<T>,
  maxRetries: number = 3,
  baseDelay: number = 1000
): Promise<T> {
  let lastError: Error;

  for (let attempt = 0; attempt <= maxRetries; attempt++) {
    try {
      return await operation();
    } catch (error) {
      lastError = error instanceof Error ? error : new Error(String(error));

      if (attempt === maxRetries) {
        break;
      }

      // Exponential backoff
      const delay = baseDelay * Math.pow(2, attempt);
      await new Promise((resolve) => setTimeout(resolve, delay));
    }
  }

  throw lastError!;
}

// Circuit breaker pattern
class CircuitBreaker {
  private failures = 0;
  private lastFailureTime = 0;
  private state: "CLOSED" | "OPEN" | "HALF_OPEN" = "CLOSED";

  constructor(private threshold: number = 5, private timeout: number = 60000) {}

  async execute<T>(operation: () => Promise<T>): Promise<T> {
    if (this.state === "OPEN") {
      if (Date.now() - this.lastFailureTime > this.timeout) {
        this.state = "HALF_OPEN";
      } else {
        throw new Error("Circuit breaker is OPEN");
      }
    }

    try {
      const result = await operation();
      this.onSuccess();
      return result;
    } catch (error) {
      this.onFailure();
      throw error;
    }
  }

  private onSuccess(): void {
    this.failures = 0;
    this.state = "CLOSED";
  }

  private onFailure(): void {
    this.failures++;
    this.lastFailureTime = Date.now();

    if (this.failures >= this.threshold) {
      this.state = "OPEN";
    }
  }

  getState(): string {
    return this.state;
  }
}

// Usage
const circuitBreaker = new CircuitBreaker(3, 30000);

async function reliableFetchUser(userId: string): Promise<User> {
  return circuitBreaker.execute(async () => {
    return withRetry(() => fetchUserData(userId), 2, 500);
  });
}
```

Error Logging and Monitoring

Structured Error Logging

```
// Logger interface
interface Logger {
  error(message: string, error?: Error, context?: Record<string, any>): void;
  warn(message: string, context?: Record<string, any>): void;
  info(message: string, context?: Record<string, any>): void;
  debug(message: string, context?: Record<string, any>): void;
}

// Error context interface
interface ErrorContext {
  userId?: string;
  requestId?: string;
  operation?: string;
  timestamp: string;
  stackTrace?: string;
  additionalData?: Record<string, any>;
}

// Enhanced error logger
class ErrorLogger implements Logger {
  constructor(private serviceName: string, private environment: string) {}

  error(
    message: string,
    error?: Error,
    context: Record<string, any> = {}
  ): void {
    const errorContext: ErrorContext = {
      ...context,
      timestamp: new Date().toISOString(),
      stackTrace: error?.stack,
    };

    const logEntry = {
      level: "error",
      service: this.serviceName,
      environment: this.environment,
      message,
      error: error
        ? {
            name: error.name,
            message: error.message,
            stack: error.stack,
          }
        : undefined,
      context: errorContext,
    };

    // In production, send to logging service
    if (this.environment === "production") {
      this.sendToLoggingService(logEntry);
    } else {
      console.error(JSON.stringify(logEntry, null, 2));
    }
  }

  warn(message: string, context: Record<string, any> = {}): void {
    this.log("warn", message, context);
  }

  info(message: string, context: Record<string, any> = {}): void {
    this.log("info", message, context);
  }

  debug(message: string, context: Record<string, any> = {}): void {
    this.log("debug", message, context);
  }

  private log(
    level: string,
    message: string,
    context: Record<string, any>
  ): void {
    const logEntry = {
      level,
      service: this.serviceName,
      environment: this.environment,
      message,
      context: {
        ...context,
        timestamp: new Date().toISOString(),
      },
    };

    console.log(JSON.stringify(logEntry, null, 2));
  }

  private async sendToLoggingService(logEntry: any): Promise<void> {
    try {
      // Send to external logging service (e.g., CloudWatch, Datadog, etc.)
      await fetch("/api/logs", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(logEntry),
      });
    } catch (error) {
      // Fallback to console if logging service is unavailable
      console.error("Failed to send log to service:", error);
      console.error("Original log entry:", logEntry);
    }
  }
}

// Error monitoring and metrics
class ErrorMonitor {
  private errorCounts = new Map<string, number>();
  private logger: Logger;

  constructor(logger: Logger) {
    this.logger = logger;
  }

  recordError(error: Error, context: Record<string, any> = {}): void {
    const errorKey = `${error.name}:${error.message}`;
    const currentCount = this.errorCounts.get(errorKey) || 0;
    this.errorCounts.set(errorKey, currentCount + 1);

    this.logger.error("Error recorded", error, {
      ...context,
      errorCount: currentCount + 1,
      errorKey,
    });

    // Alert if error frequency is high
    if (currentCount + 1 >= 10) {
      this.sendAlert(error, currentCount + 1, context);
    }
  }

  getErrorStats(): Record<string, number> {
    return Object.fromEntries(this.errorCounts);
  }

  resetStats(): void {
    this.errorCounts.clear();
  }

  private sendAlert(
    error: Error,
    count: number,
    context: Record<string, any>
  ): void {
    this.logger.error("High error frequency detected", error, {
      ...context,
      alertType: "high_frequency",
      errorCount: count,
      threshold: 10,
    });

    // In production, send to alerting system
    // await this.sendToAlertingSystem({ error, count, context });
  }
}
```

Global Error Handling

Application-wide Error Handler

```
// Global error handler
class GlobalErrorHandler {
  private logger: Logger;
  private monitor: ErrorMonitor;

  constructor(logger: Logger, monitor: ErrorMonitor) {
    this.logger = logger;
    this.monitor = monitor;
    this.setupGlobalHandlers();
  }

  private setupGlobalHandlers(): void {
    // Handle unhandled promise rejections
    process.on("unhandledRejection", (reason: any, promise: Promise<any>) => {
      const error =
        reason instanceof Error ? reason : new Error(String(reason));
      this.handleError(error, {
        type: "unhandledRejection",
        promise: promise.toString(),
      });
    });

    // Handle uncaught exceptions
    process.on("uncaughtException", (error: Error) => {
      this.handleError(error, {
        type: "uncaughtException",
        fatal: true,
      });

      // Graceful shutdown
      process.exit(1);
    });

    // Handle warnings
    process.on("warning", (warning: Error) => {
      this.logger.warn("Process warning", {
        name: warning.name,
        message: warning.message,
        stack: warning.stack,
      });
    });
  }

  handleError(error: Error, context: Record<string, any> = {}): void {
    // Determine if error is operational or programming error
    const isOperational = error instanceof AppError && error.isOperational;

    this.monitor.recordError(error, {
      ...context,
      isOperational,
      handled: true,
    });

    if (!isOperational) {
      // Programming errors should be logged with high priority
      this.logger.error("Programming error detected", error, {
        ...context,
        severity: "critical",
      });
    }
  }

  // Express.js error middleware
  expressErrorHandler() {
    return (error: Error, req: any, res: any, next: any) => {
      const context = {
        method: req.method,
        url: req.url,
        userAgent: req.get("User-Agent"),
        ip: req.ip,
        userId: req.user?.id,
        requestId: req.requestId,
      };

      this.handleError(error, context);

      if (error instanceof AppError) {
        res.status(error.statusCode).json({
          error: {
            message: error.message,
            type: error.constructor.name,
            ...(error.isOperational
              ? {}
              : { details: "Internal server error" }),
          },
        });
      } else {
        res.status(500).json({
          error: {
            message: "Internal server error",
            type: "InternalError",
          },
        });
      }
    };
  }
}

// Application setup
const logger = new ErrorLogger(
  "user-service",
  process.env.NODE_ENV || "development"
);
const monitor = new ErrorMonitor(logger);
const globalErrorHandler = new GlobalErrorHandler(logger, monitor);

// Express app setup (example)
// app.use(globalErrorHandler.expressErrorHandler());
```

Best Practices

✅ Good Practices

```
// Create specific error types for different scenarios
class PaymentError extends AppError {
  constructor(
    message: string,
    public readonly paymentId: string,
    public readonly amount: number,
    public readonly currency: string
  ) {
    super(message);
  }
}

// Use Result pattern for operations that commonly fail
function parseConfig(configString: string): Result<Config, ValidationError> {
  try {
    const config = JSON.parse(configString);
    return validateConfig(config);
  } catch (error) {
    return failure(
      new ValidationError("Invalid JSON format", "config", configString)
    );
  }
}

// Provide context in error messages
function processOrder(orderId: string): void {
  try {
    // Process order
  } catch (error) {
    throw new Error(`Failed to process order ${orderId}: ${error}`);
  }
}

// Use type guards for error handling
function isAppError(error: unknown): error is AppError {
  return error instanceof AppError;
}

function handleApiError(error: unknown): void {
  if (isAppError(error)) {
    console.error(`App error [${error.statusCode}]: ${error.message}`);
  } else if (error instanceof Error) {
    console.error(`Unexpected error: ${error.message}`);
  } else {
    console.error("Unknown error:", error);
  }
}

// Validate inputs early
function createUser(userData: any): User {
  if (!userData || typeof userData !== "object") {
    throw new ValidationError("Invalid user data", "userData", userData);
  }

  if (!userData.email || typeof userData.email !== "string") {
    throw new ValidationError("Email is required", "email", userData.email);
  }

  // Continue with user creation
  return userData as User;
}
```

❌ Avoid

```
// Don't swallow errors silently
try {
  riskyOperation();
} catch (error) {
  // ❌ Silent failure
}

// Don't use generic error messages
throw new Error("Something went wrong"); // ❌ Not helpful

// Don't catch and re-throw without adding value
try {
  operation();
} catch (error) {
  throw error; // ❌ Pointless catch
}

// Don't use string errors
throw "Error message"; // ❌ Should be Error object

// Don't ignore error types
function handleError(error: any): void {
  // ❌ Should be more specific
  console.log(error.message); // Might not exist
}

// Don't create overly broad catch blocks
try {
  operation1();
  operation2();
  operation3();
} catch (error) {
  // ❌ Can't tell which operation failed
  console.error("One of the operations failed");
}

function riskyOperation(): void {
  /* Implementation */
}
function operation(): void {
  /* Implementation */
}
function operation1(): void {
  /* Implementation */
}
function operation2(): void {
  /* Implementation */
}
function operation3(): void {
  /* Implementation */
}
interface Config {
  /* Definition */
}
function validateConfig(config: any): Result<Config, ValidationError> {
  /* Implementation */ return success({} as Config);
}
```

Summary Checklist

- □ Create specific error types for different scenarios
- □ Use the Result pattern for operations that commonly fail
- □ Implement proper async error handling
- □ Set up structured error logging
- □ Use global error handlers for unhandled errors
- □ Provide meaningful error messages with context

- □ Distinguish between operational and programming errors
- □ Implement retry mechanisms for transient failures
- □ Use circuit breakers for external service calls
- □ Monitor and alert on error patterns

Next Steps

Now that you understand error handling in TypeScript, let's explore utility types and advanced type manipulations.

Continue to: Utility Types and Type Manipulations

Utility Types and Type Manipulations

> Master TypeScript's built-in utility types and learn to create custom type manipulations for advanced type safety and code reusability

Built-in Utility Types

TypeScript provides many built-in utility types that help transform and manipulate existing types.

Object Manipulation Utilities

```
// Base interface for examples
interface User {
  id: number;
  name: string;
  email: string;
  age: number;
  isActive: boolean;
  createdAt: Date;
  updatedAt: Date;
}

// Partial<T> - Makes all properties optional
type PartialUser = Partial<User>;
// {
//   id?: number;
//   name?: string;
//   email?: string;
//   age?: number;
//   isActive?: boolean;
//   createdAt?: Date;
//   updatedAt?: Date;
// }

// Required<T> - Makes all properties required
interface OptionalUser {
  id?: number;
  name?: string;
  email?: string;
}

type RequiredUser = Required<OptionalUser>;
// {
//   id: number;
//   name: string;
//   email: string;
// }

// Readonly<T> - Makes all properties readonly
type ReadonlyUser = Readonly<User>;
// {
//   readonly id: number;
//   readonly name: string;
//   readonly email: string;
//   // ... all properties are readonly
// }

// Pick<T, K> - Select specific properties
type UserSummary = Pick<User, "id" | "name" | "email">;
// {
//   id: number;
//   name: string;
//   email: string;
// }

// Omit<T, K> - Exclude specific properties
type CreateUserRequest = Omit<User, "id" | "createdAt" | "updatedAt">;
// {
//   name: string;
//   email: string;
//   age: number;
//   isActive: boolean;
// }

// Record<K, T> - Create object type with specific keys and values
type UserRoles = Record<"admin" | "user" | "guest", string[]>;
// {
//   admin: string[];
//   user: string[];
//   guest: string[];
// }

type StatusMessages = Record<number, string>;
// { [key: number]: string }

// Example usage
const httpStatusMessages: StatusMessages = {
  200: "OK",
  404: "Not Found",
  500: "Internal Server Error",
};
```

```
const rolePermissions: UserRoles = {
  admin: ["read", "write", "delete"],
  user: ["read", "write"],
  guest: ["read"],
};
```

Union and Intersection Utilities

```
// Exclude<T, U> - Remove types from union
type PrimaryColors = "red" | "green" | "blue";
type WarmColors = "red" | "orange" | "yellow";

type CoolColors = Exclude<PrimaryColors, "red">; // 'green' | 'blue'
type NonWarmPrimary = Exclude<PrimaryColors, WarmColors>; // 'green' | 'blue'

// Extract<T, U> - Keep only specific types from union
type WarmPrimaryColors = Extract<PrimaryColors, WarmColors>; // 'red'

// NonNullable<T> - Remove null and undefined
type MaybeString = string | null | undefined;
type DefiniteString = NonNullable<MaybeString>; // string

// Example with more complex types
type ApiResponse<T> = T | null | undefined | Error;
type ValidApiResponse<T> = NonNullable<ApiResponse<T>>; // T | Error

// Practical example: filtering union types
type EventType = "click" | "hover" | "focus" | "blur" | "keydown" | "keyup";
type MouseEvents = Extract<EventType, "click" | "hover">; // 'click' | 'hover'
type KeyboardEvents = Extract<EventType, `key${string}`>; // 'keydown' | 'keyup'
type NonMouseEvents = Exclude<EventType, MouseEvents>; // 'focus' | 'blur' | 'keydown' | 'keyup'
```

Function Utilities

```
// Function type for examples
function calculateTotal(price: number, tax: number, discount?: number): number {
  const subtotal = price + price * tax;
  return discount ? subtotal - discount : subtotal;
}

class UserService {
  async getUser(id: string): Promise<User> {
    // Implementation
    return {} as User;
  }

  updateUser(id: string, updates: Partial<User>): User {
    // Implementation
    return {} as User;
  }
}

// Parameters<T> - Extract function parameter types
type CalculateTotalParams = Parameters<typeof calculateTotal>;
// [price: number, tax: number, discount?: number]

type GetUserParams = Parameters<UserService["getUser"]>;
// [id: string]

// ReturnType<T> - Extract function return type
type CalculateTotalReturn = ReturnType<typeof calculateTotal>; // number
type GetUserReturn = ReturnType<UserService["getUser"]>; // Promise<User>

// ConstructorParameters<T> - Extract constructor parameter types
class DatabaseConnection {
  constructor(host: string, port: number, options?: { ssl: boolean }) {
    // Implementation
  }
}

type DbConnectionParams = ConstructorParameters<typeof DatabaseConnection>;
// [host: string, port: number, options?: { ssl: boolean }]

// InstanceType<T> - Extract instance type from constructor
type DbInstance = InstanceType<typeof DatabaseConnection>; // DatabaseConnection

// ThisParameterType<T> - Extract 'this' parameter type
function greetUser(this: User, message: string): string {
  return `${message}, ${this.name}!`;
}

type GreetUserThis = ThisParameterType<typeof greetUser>; // User

// OmitThisParameter<T> - Remove 'this' parameter from function type
type GreetUserWithoutThis = OmitThisParameter<typeof greetUser>;
// (message: string) => string

// Practical example: creating type-safe event handlers
interface EventHandlers {
  onClick(this: HTMLButtonElement, event: MouseEvent): void;
  onSubmit(this: HTMLFormElement, event: SubmitEvent): void;
  onChange(this: HTMLInputElement, event: Event): void;
}

type ClickHandler = EventHandlers["onClick"];
type ClickHandlerParams = Parameters<ClickHandler>; // [event: MouseEvent]
type ClickHandlerThis = ThisParameterType<ClickHandler>; // HTMLButtonElement
type ClickHandlerWithoutThis = OmitThisParameter<ClickHandler>; // (event: MouseEvent) => void
```

String Manipulation Utilities

```typescript
// Uppercase<T> - Convert string literal to uppercase
type UppercaseHello = Uppercase<"hello">; // 'HELLO'
type UppercaseColors = Uppercase<"red" | "green" | "blue">; // 'RED' | 'GREEN' | 'BLUE'

// Lowercase<T> - Convert string literal to lowercase
type LowercaseHello = Lowercase<"HELLO">; // 'hello'
type LowercaseStatus = Lowercase<"SUCCESS" | "ERROR" | "PENDING">; // 'success' | 'error' | 'pending'

// Capitalize<T> - Capitalize first letter
type CapitalizedHello = Capitalize<"hello world">; // 'Hello world'
type CapitalizedColors = Capitalize<"red" | "green" | "blue">; // 'Red' | 'Green' | 'Blue'

// Uncapitalize<T> - Uncapitalize first letter
type UncapitalizedHello = Uncapitalize<"Hello World">; // 'hello World'

// Practical example: API endpoint generation
type HttpMethod = "get" | "post" | "put" | "delete";
type Resource = "user" | "product" | "order";

type ApiEndpoint<
  M extends HttpMethod,
  R extends Resource
> = `${Uppercase<M>} /api/${R}s`;

type UserEndpoints = ApiEndpoint<HttpMethod, "user">;
// 'GET /api/users' | 'POST /api/users' | 'PUT /api/users' | 'DELETE /api/users'

// Environment variable types
type EnvPrefix = "DATABASE" | "API" | "REDIS";
type EnvSuffix = "HOST" | "PORT" | "PASSWORD";

type EnvVariable<P extends EnvPrefix, S extends EnvSuffix> = `${P}_${S}`;

type DatabaseEnvVars = EnvVariable<"DATABASE", EnvSuffix>;
// 'DATABASE_HOST' | 'DATABASE_PORT' | 'DATABASE_PASSWORD'

// CSS property generation
type CSSProperty = "margin" | "padding";
type CSSDirection = "top" | "right" | "bottom" | "left";

type CSSDirectionalProperty<
  P extends CSSProperty,
  D extends CSSDirection
> = `${P}-${D}`;

type MarginProperties = CSSDirectionalProperty<"margin", CSSDirection>;
// 'margin-top' | 'margin-right' | 'margin-bottom' | 'margin-left'
```

Custom Utility Types

Advanced Object Manipulation

```typescript
// DeepPartial - Make all properties optional recursively
type DeepPartial<T> = {
  [P in keyof T]?: T[P] extends object ? DeepPartial<T[P]> : T[P];
};

interface NestedConfig {
  database: {
    host: string;
    port: number;
    credentials: {
      username: string;
      password: string;
    };
  };
  api: {
    baseUrl: string;
    timeout: number;
  };
}

type PartialNestedConfig = DeepPartial<NestedConfig>;
// {
//   database?: {
//     host?: string;
//     port?: number;
//     credentials?: {
//       username?: string;
//       password?: string;
//     };
//   };
//   api?: {
//     baseUrl?: string;
//     timeout?: number;
//   };
// }

// DeepReadonly - Make all properties readonly recursively
type DeepReadonly<T> = {
  readonly [P in keyof T]: T[P] extends object ? DeepReadonly<T[P]> : T[P];
};

type ReadonlyNestedConfig = DeepReadonly<NestedConfig>;

// DeepRequired - Make all properties required recursively
type DeepRequired<T> = {
  [P in keyof T]-?: T[P] extends object ? DeepRequired<T[P]> : T[P];
};
```

```
// Mutable - Remove readonly modifiers
type Mutable<T> = {
  -readonly [P in keyof T]: T[P];
};

type MutableUser = Mutable<ReadonlyUser>;
// Back to regular User interface

// PickByType - Pick properties by their type
type PickByType<T, U> = {
  [P in keyof T as T[P] extends U ? P : never]: T[P];
};

type UserStringProperties = PickByType<User, string>;
// { name: string; email: string }

type UserNumberProperties = PickByType<User, number>;
// { id: number; age: number }

type UserDateProperties = PickByType<User, Date>;
// { createdAt: Date; updatedAt: Date }

// OmitByType - Omit properties by their type
type OmitByType<T, U> = {
  [P in keyof T as T[P] extends U ? never : P]: T[P];
};

type UserWithoutDates = OmitByType<User, Date>;
// { id: number; name: string; email: string; age: number; isActive: boolean }

// NonEmptyArray - Ensure array has at least one element
type NonEmptyArray<T> = [T, ...T[]];

function processItems<T>(items: NonEmptyArray<T>): T {
  return items[0]; // Safe to access first element
}

// Usage
const validItems: NonEmptyArray<string> = ["first", "second"];
const firstItem = processItems(validItems); // OK

// const emptyItems: NonEmptyArray<string> = []; // Error: Source has 0 element(s) but target requires 1
```

Conditional Type Utilities

```
// IsNever - Check if type is never
type IsNever<T> = [T] extends [never] ? true : false;

type TestNever1 = IsNever<never>; // true
type TestNever2 = IsNever<string>; // false

// IsAny - Check if type is any
type IsAny<T> = 0 extends 1 & T ? true : false;

type TestAny1 = IsAny<any>; // true
type TestAny2 = IsAny<string>; // false

// IsUnknown - Check if type is unknown
type IsUnknown<T> = IsAny<T> extends true
  ? false
  : unknown extends T
  ? true
  : false;

type TestUnknown1 = IsUnknown<unknown>; // true
type TestUnknown2 = IsUnknown<string>; // false

// Equals - Check if two types are equal
type Equals<X, Y> = (<T>() => T extends X ? 1 : 2) extends <T>() => T extends Y
  ? 1
  : 2
  ? true
  : false;

type TestEquals1 = Equals<string, string>; // true
type TestEquals2 = Equals<string, number>; // false
type TestEquals3 = Equals<string | number, number | string>; // true

// If - Conditional type helper
type If<C extends boolean, T, F> = C extends true ? T : F;

type TestIf1 = If<true, "yes", "no">; // 'yes'
type TestIf2 = If<false, "yes", "no">; // 'no'

// Not - Boolean negation
type Not<C extends boolean> = C extends true ? false : true;

type TestNot1 = Not<true>; // false
type TestNot2 = Not<false>; // true

// And - Boolean AND operation
type And<A extends boolean, B extends boolean> = A extends true ? B : false;

type TestAnd1 = And<true, true>; // true
type TestAnd2 = And<true, false>; // false
type TestAnd3 = And<false, true>; // false

// Or - Boolean OR operation
type Or<A extends boolean, B extends boolean> = A extends true ? true : B;

type TestOr1 = Or<true, false>; // true
type TestOr2 = Or<false, false>; // false
type TestOr3 = Or<false, true>; // true
```

Array and Tuple Utilities

```
// Head - Get first element of array/tuple
type Head<T extends readonly unknown[]> = T extends readonly [
  infer H,
  ...unknown[]
]
  ? H
  : never;

type FirstString = Head<["a", "b", "c"]>; // 'a'
type FirstNumber = Head<[1, 2, 3]>; // 1
type EmptyHead = Head<[]>; // never

// Tail - Get all elements except first
type Tail<T extends readonly unknown[]> = T extends readonly [
  unknown,
  ...infer Rest
]
  ? Rest
  : [];

type RestStrings = Tail<["a", "b", "c"]>; // ['b', 'c']
type RestNumbers = Tail<[1, 2, 3]>; // [2, 3]
type EmptyTail = Tail<[]>; // []

// Last - Get last element of array/tuple
type Last<T extends readonly unknown[]> = T extends readonly [
  ...unknown[],
  infer L
]
  ? L
  : never;

type LastString = Last<["a", "b", "c"]>; // 'c'
type LastNumber = Last<[1, 2, 3]>; // 3

// Length - Get length of tuple
type Length<T extends readonly unknown[]> = T["length"];

type LengthOfTuple = Length<["a", "b", "c"]>; // 3
type LengthOfEmpty = Length<[]>; // 0

// Reverse - Reverse tuple order
type Reverse<T extends readonly unknown[]> = T extends readonly [
  ...infer Rest,
  infer Last
]
  ? [Last, ...Reverse<Rest>]
  : [];
```

```
type ReversedTuple = Reverse<["a", "b", "c"]>; // ['c', 'b', 'a']
type ReversedNumbers = Reverse<[1, 2, 3, 4]>; // [4, 3, 2, 1]

// Flatten - Flatten nested arrays
type Flatten<T extends readonly unknown[]> = T extends readonly [
  infer First,
  ...infer Rest
]
  ? First extends readonly unknown[]
    ? [...Flatten<First>, ...Flatten<Rest>]
    : [First, ...Flatten<Rest>]
  : [];

type FlatArray = Flatten<[1, [2, 3], [4, [5, 6]]]>; // [1, 2, 3, 4, [5, 6]]

// Includes - Check if array includes specific type
type Includes<T extends readonly unknown[], U> = T extends readonly [
  infer First,
  ...infer Rest
]
  ? Equals<First, U> extends true
    ? true
    : Includes<Rest, U>
  : false;

type HasString = Includes<["a", "b", "c"], "b">; // true
type HasNumber = Includes<["a", "b", "c"], 1>; // false

// Unique - Remove duplicate types from tuple
type Unique<
  T extends readonly unknown[],
  Result extends readonly unknown[] = []
> = T extends readonly [infer First, ...infer Rest]
  ? Includes<Result, First> extends true
    ? Unique<Rest, Result>
    : Unique<Rest, [...Result, First]>
  : Result;

type UniqueArray = Unique<["a", "b", "a", "c", "b"]>; // ['a', 'b', 'c']
```

String Manipulation Utilities

```
// Split - Split string by delimiter
type Split<
  S extends string,
  D extends string
> = S extends `${infer T}${D}${infer U}` ? [T, ...Split<U, D>] : [S];

type SplitPath = Split<"user/profile/settings", "/">; // ['user', 'profile', 'settings']
type SplitEmail = Split<"user@example.com", "@">; // ['user', 'example.com']

// Join - Join array of strings with delimiter
type Join<T extends readonly string[], D extends string> = T extends readonly [
  infer First,
  ...infer Rest
]
  ? First extends string
    ? Rest extends readonly string[]
      ? Rest["length"] extends 0
        ? First
        : `${First}${D}${Join<Rest, D>}`
      : never
    : never
  : "";

type JoinedPath = Join<["user", "profile", "settings"], "/">; // 'user/profile/settings'
type JoinedWords = Join<["hello", "world"], " ">; // 'hello world'

// Replace - Replace substring in string
type Replace<
  S extends string,
  From extends string,
  To extends string
> = S extends `${infer Prefix}${From}${infer Suffix}`
  ? `${Prefix}${To}${Suffix}`
  : S;

type ReplacedString = Replace<"hello world", "world", "TypeScript">; // 'hello TypeScript'

// ReplaceAll - Replace all occurrences of substring
type ReplaceAll<
  S extends string,
  From extends string,
  To extends string
> = S extends `${infer Prefix}${From}${infer Suffix}`
  ? `${Prefix}${To}${ReplaceAll<Suffix, From, To>}`
  : S;

type ReplacedAllSpaces = ReplaceAll<"hello world test", " ", "-">; // 'hello-world-test'

// StartsWith - Check if string starts with prefix
type StartsWith<
  S extends string,
  Prefix extends string
> = S extends `${Prefix}${string}` ? true : false;

type StartsWithHello = StartsWith<"hello world", "hello">; // true
type StartsWithBye = StartsWith<"hello world", "bye">; // false

// EndsWith - Check if string ends with suffix
```

```
type EndsWith<
  S extends string,
  Suffix extends string
> = S extends `${string}${Suffix}` ? true : false;

type EndsWithWorld = EndsWith<"hello world", "world">; // true
type EndsWithTest = EndsWith<"hello world", "test">; // false

// TrimLeft - Remove leading whitespace
type TrimLeft<S extends string> = S extends ` ${infer Rest}`
  ? TrimLeft<Rest>
  : S;

type TrimmedLeft = TrimLeft<"   hello world">; // 'hello world'

// TrimRight - Remove trailing whitespace
type TrimRight<S extends string> = S extends `${infer Rest} `
  ? TrimRight<Rest>
  : S;

type TrimmedRight = TrimRight<"hello world   ">; // 'hello world'

// Trim - Remove leading and trailing whitespace
type Trim<S extends string> = TrimLeft<TrimRight<S>>;

type TrimmedString = Trim<"   hello world   ">; // 'hello world'
```

Path and Property Utilities

```
// Get - Get nested property type by path
type Get<T, K> = K extends `${infer Key}.${infer Rest}`
  ? Key extends keyof T
    ? Get<T[Key], Rest>
    : never
  : K extends keyof T
  ? T[K]
  : never;

interface NestedObject {
  user: {
    profile: {
      name: string;
      age: number;
    };
    settings: {
      theme: "light" | "dark";
      notifications: boolean;
    };
  };
  app: {
    version: string;
  };
}

type UserName = Get<NestedObject, "user.profile.name">; // string
type UserAge = Get<NestedObject, "user.profile.age">; // number
type Theme = Get<NestedObject, "user.settings.theme">; // 'light' | 'dark'
type AppVersion = Get<NestedObject, "app.version">; // string

// Paths - Generate all possible paths in an object
type Paths<T, D extends number = 10> = [D] extends [never]
  ? never
  : T extends object
  ? {
      [K in keyof T]-?: K extends string | number
        ? `${K}` | Join<[K, Paths<T[K], Prev[D]>], ".">
        : never;
    }[keyof T]
  : "";

type Prev = [
  never,
  0,
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16,
  17,
  18,
  19,
  20,
  ...0[]
];

type AllPaths = Paths<NestedObject>;
// 'user' | 'app' | 'user.profile' | 'user.settings' | 'user.profile.name' |
// 'user.profile.age' | 'user.settings.theme' | 'user.settings.notifications' | 'app.version'

// Leaves - Get only leaf paths (paths to primitive values)
type Leaves<T, D extends number = 10> = [D] extends [never]
  ? never
  : T extends object
  ? {
      [K in keyof T]-?: Join<[K, Leaves<T[K], Prev[D]>], ".">;
    }[keyof T]
  : "";

type LeafPaths = Leaves<NestedObject>;
// 'user.profile.name' | 'user.profile.age' | 'user.settings.theme' |
// 'user.settings.notifications' | 'app.version'

// Set - Set nested property type by path
type Set<T, K extends string, V> = K extends `${infer Key}.${infer Rest}`
  ? Key extends keyof T
    ? {
        [P in keyof T]: P extends Key ? Set<T[P], Rest, V> : T[P];
      }
    : T
  : K extends keyof T
  ? {
      [P in keyof T]: P extends K ? V : T[P];
    }
  : T;

type UpdatedObject = Set<NestedObject, "user.profile.name", number>;
// Changes user.profile.name from string to number
```

Practical Examples

Type-Safe Configuration System

```
// Configuration schema with nested structure
interface AppConfig {
  database: {
    host: string;
    port: number;
    ssl: boolean;
    pool: {
      min: number;
      max: number;
    };
  };
  api: {
    baseUrl: string;
    timeout: number;
    retries: number;
  };
  features: {
    authentication: boolean;
    logging: boolean;
    analytics: boolean;
  };
}
```

```
// Type-safe configuration getter
class ConfigManager<T extends Record<string, any>> {
  constructor(private config: T) {}

  get<P extends Paths<T>>(path: P): Get<T, P> {
    const keys = path.split(".") as string[];
    let value: any = this.config;

    for (const key of keys) {
      value = value?.[key];
    }

    return value;
  }

  set<P extends Paths<T>, V>(path: P, value: V): ConfigManager<Set<T, P, V>> {
    const keys = path.split(".") as string[];
    const newConfig = JSON.parse(JSON.stringify(this.config));
    let current = newConfig;

    for (let i = 0; i < keys.length - 1; i++) {
      current = current[keys[i]];
    }

    current[keys[keys.length - 1]] = value;
    return new ConfigManager(newConfig);
  }

  update<P extends Paths<T>>(
    path: P,
    updater: (current: Get<T, P>) => Get<T, P>
  ): ConfigManager<T> {
    const currentValue = this.get(path);
    const newValue = updater(currentValue);
    return this.set(path, newValue) as ConfigManager<T>;
  }
}

// Usage
const config = new ConfigManager<AppConfig>({
  database: {
    host: "localhost",
    port: 5432,
    ssl: false,
    pool: { min: 2, max: 10 },
  },
  api: {
    baseUrl: "https://api.example.com",
    timeout: 5000,
    retries: 3,
  },
  features: {
    authentication: true,
    logging: false,
    analytics: true,
  },
});

// Type-safe access
const dbHost = config.get("database.host"); // string
const poolMax = config.get("database.pool.max"); // number
const authEnabled = config.get("features.authentication"); // boolean

// Type-safe updates
const updatedConfig = config
  .set("database.ssl", true)
  .set("api.timeout", 10000)
  .update("database.pool.max", (current) => current * 2);
```

Form Validation System

```
// Form field types
type FieldType = "string" | "number" | "boolean" | "date" | "email";

interface FieldSchema {
  type: FieldType;
  required?: boolean;
  min?: number;
  max?: number;
  pattern?: string;
}

type FormSchema = Record<string, FieldSchema>;

// Extract form data type from schema
type FormDataFromSchema<T extends FormSchema> = {
  [K in keyof T]: T[K]["type"] extends "string" | "email"
    ? string
    : T[K]["type"] extends "number"
    ? number
    : T[K]["type"] extends "boolean"
    ? boolean
    : T[K]["type"] extends "date"
    ? Date
    : unknown;
};

// Extract required fields
type RequiredFields<T extends FormSchema> = {
  [K in keyof T]: T[K]["required"] extends true ? K : never;
}[keyof T];

// Extract optional fields
type OptionalFields<T extends FormSchema> = Exclude<keyof T, RequiredFields<T>>;
```

```
// Create form data type with proper optionality
type FormData<T extends FormSchema> = Pick<
  FormDataFromSchema<T>,
  RequiredFields<T>
> &
  Partial<Pick<FormDataFromSchema<T>, OptionalFields<T>>>;

// Validation result type
type ValidationResult<T extends FormSchema> = {
  isValid: boolean;
  errors: Partial<Record<keyof T, string[]>>;
};

// Form validator class
class FormValidator<T extends FormSchema> {
  constructor(private schema: T) {}

  validate(data: Partial<FormDataFromSchema<T>>): ValidationResult<T> {
    const errors: Partial<Record<keyof T, string[]>> = {};
    let isValid = true;

    for (const [fieldName, fieldSchema] of Object.entries(this.schema)) {
      const value = data[fieldName as keyof T];
      const fieldErrors: string[] = [];

      // Required validation
      if (fieldSchema.required && (value === undefined || value === null)) {
        fieldErrors.push(`${fieldName} is required`);
        isValid = false;
      }

      if (value !== undefined && value !== null) {
        // Type-specific validation
        if (fieldSchema.type === "email" && typeof value === "string") {
          const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
          if (!emailRegex.test(value)) {
            fieldErrors.push("Invalid email format");
            isValid = false;
          }
        }

        if (fieldSchema.type === "string" && typeof value === "string") {
          if (fieldSchema.min && value.length < fieldSchema.min) {
            fieldErrors.push(`Minimum length is ${fieldSchema.min}`);
            isValid = false;
          }
          if (fieldSchema.max && value.length > fieldSchema.max) {
            fieldErrors.push(`Maximum length is ${fieldSchema.max}`);
            isValid = false;
          }
          if (fieldSchema.pattern) {
            const regex = new RegExp(fieldSchema.pattern);
            if (!regex.test(value)) {
              fieldErrors.push("Invalid format");
              isValid = false;
            }
          }
        }

        if (fieldSchema.type === "number" && typeof value === "number") {
          if (fieldSchema.min && value < fieldSchema.min) {
            fieldErrors.push(`Minimum value is ${fieldSchema.min}`);
            isValid = false;
          }
          if (fieldSchema.max && value > fieldSchema.max) {
            fieldErrors.push(`Maximum value is ${fieldSchema.max}`);
            isValid = false;
          }
        }
      }

      if (fieldErrors.length > 0) {
        errors[fieldName as keyof T] = fieldErrors;
      }
    }

    return { isValid, errors };
  }
}

// Usage example
const userFormSchema = {
  name: { type: "string" as const, required: true, min: 2, max: 50 },
  email: { type: "email" as const, required: true },
  age: { type: "number" as const, min: 18, max: 120 },
  newsletter: { type: "boolean" as const },
  website: { type: "string" as const, pattern: "^https?://.+" },
} satisfies FormSchema;

type UserFormData = FormData<typeof userFormSchema>;
// {
//   name: string;
//   email: string;
//   age?: number;
//   newsletter?: boolean;
//   website?: string;
// }

const validator = new FormValidator(userFormSchema);

const formData: Partial<FormDataFromSchema<typeof userFormSchema>> = {
  name: "John Doe",
  email: "john@example.com",
  age: 25,
  newsletter: true,
  website: "https://johndoe.com",
};
```

```
const result = validator.validate(formData);
if (result.isValid) {
  console.log("Form is valid!");
} else {
  console.log("Validation errors:", result.errors);
}
```

API Response Type System

```
// Base API response structure
interface BaseApiResponse {
  success: boolean;
  timestamp: string;
  requestId: string;
}

interface SuccessResponse<T> extends BaseApiResponse {
  success: true;
  data: T;
}

interface ErrorResponse extends BaseApiResponse {
  success: false;
  error: {
    code: string;
    message: string;
    details?: Record<string, any>;
  };
}

type ApiResponse<T> = SuccessResponse<T> | ErrorResponse;

// API endpoint definitions
interface ApiEndpoints {
  "GET /users": {
    response: User[];
    query?: {
      page?: number;
      limit?: number;
      search?: string;
    };
  };
  "GET /users/:id": {
    response: User;
    params: { id: string };
  };
  "POST /users": {
    response: User;
    body: CreateUserRequest;
  };
  "PUT /users/:id": {
    response: User;
    params: { id: string };
    body: Partial<CreateUserRequest>;
  };
  "DELETE /users/:id": {
    response: void;
    params: { id: string };
  };
}

// Extract types from endpoint definitions
type ExtractResponse<T> = T extends { response: infer R } ? R : never;
type ExtractParams<T> = T extends { params: infer P } ? P : never;
type ExtractQuery<T> = T extends { query: infer Q } ? Q : never;
type ExtractBody<T> = T extends { body: infer B } ? B : never;

// Type-safe API client
class ApiClient {
  constructor(private baseUrl: string) {}

  async request<K extends keyof ApiEndpoints>(
    endpoint: K,
    options: {
      params?: ExtractParams<ApiEndpoints[K]>;
      query?: ExtractQuery<ApiEndpoints[K]>;
      body?: ExtractBody<ApiEndpoints[K]>;
    } = {}
  ): Promise<ApiResponse<ExtractResponse<ApiEndpoints[K]>>> {
    const [method, path] = endpoint.split(" ") as [string, string];

    // Replace path parameters
    let url = path;
    if (options.params) {
      for (const [key, value] of Object.entries(options.params)) {
        url = url.replace(`:${key}`, String(value));
      }
    }

    // Add query parameters
    if (options.query) {
      const searchParams = new URLSearchParams();
      for (const [key, value] of Object.entries(options.query)) {
        if (value !== undefined) {
          searchParams.append(key, String(value));
        }
      }
      if (searchParams.toString()) {
        url += `?${searchParams.toString()}`;
      }
    }

    const response = await fetch(`${this.baseUrl}${url}`, {
      method,
      headers: {
        "Content-Type": "application/json",
      },
      body: options.body ? JSON.stringify(options.body) : undefined,
    });

    return response.json();
  }
}

// Usage with full type safety
const apiClient = new ApiClient("https://api.example.com");

// Get all users with query parameters
const usersResponse = await apiClient.request("GET /users", {
  query: { page: 1, limit: 10, search: "john" },
});

if (usersResponse.success) {
  const users = usersResponse.data; // Type: User[]
  console.log("Users:", users);
} else {
  console.error("Error:", usersResponse.error.message);
}

// Get specific user
const userResponse = await apiClient.request("GET /users/:id", {
  params: { id: "123" },
});

// Create new user
const createResponse = await apiClient.request("POST /users", {
  body: {
    name: "Jane Doe",
    email: "jane@example.com",
    age: 28,
    isActive: true,
  },
});

// Update user
const updateResponse = await apiClient.request("PUT /users/:id", {
  params: { id: "123" },
  body: { name: "Jane Smith" },
});

// Delete user
const deleteResponse = await apiClient.request("DELETE /users/:id", {
  params: { id: "123" },
});
```

Best Practices

✅ Good Practices

```typescript
// Use built-in utility types when possible
type UserUpdate = Partial<Pick<User, "name" | "email" | "age">>;

// Create reusable utility types
type Optional<T, K extends keyof T> = Omit<T, K> & Partial<Pick<T, K>>;
type RequiredBy<T, K extends keyof T> = T & Required<Pick<T, K>>;

// Use meaningful names for complex types
type DatabaseEntity<T> = T & {
  id: string;
  createdAt: Date;
  updatedAt: Date;
};

// Combine utility types for complex transformations
type ApiCreateRequest<T> = Omit<T, "id" | "createdAt" | "updatedAt">;
type ApiUpdateRequest<T> = Partial<ApiCreateRequest<T>>;

// Use conditional types for flexible APIs
type EventPayload<T extends string> = T extends "user:created"
  ? { user: User }
  : T extends "user:updated"
  ? { user: User; changes: Partial<User> }
  : T extends "user:deleted"
  ? { userId: string }
  : never;
```

❌ Avoid

```typescript
// Don't create overly complex utility types
type OverlyComplex<T> = {
  [K in keyof T as T[K] extends Function
    ? never
    : K extends `${infer Prefix}_${infer Suffix}`
    ? `${Prefix}${Capitalize<Suffix>}`
    : K]: T[K] extends object ? OverlyComplex<T[K]> : T[K];
}; // Too complex, hard to understand and maintain

// Don't use utility types when simple types suffice
type SimpleString = Pick<{ value: string }, "value">["value"]; // Just use string

// Don't create utility types that are used only once
type OneTimeUse<T> = T & { timestamp: Date }; // Better to inline

// Don't ignore type constraints
type BadUtility<T> = T extends any ? T[] : never; // any defeats the purpose

// Don't create confusing type aliases
```

```
type A<T> = T; // Not descriptive
type B<T, U> = T | U; // Use Union<T, U> or inline
```

Summary Checklist

- □ Master built-in utility types (`Partial`, `Pick`, `Omit`, etc.)
- □ Create custom utility types for common patterns
- □ Use conditional types for flexible type logic
- □ Implement string manipulation utilities when needed
- □ Build type-safe configuration and form systems
- □ Create reusable utility types for your domain
- □ Use meaningful names for complex type transformations
- □ Combine utility types for sophisticated type manipulations
- □ Avoid overly complex utility types
- □ Document complex utility types with examples

Next Steps

Now that you understand utility types and type manipulations, let's explore declaration merging and ambient declarations.

Continue to: Declaration Merging and Ambient Declarations

Declaration Merging and Ambient Declarations

> Learn how to extend existing types, work with third-party libraries, and create ambient declarations for JavaScript code

Declaration Merging

Declaration merging allows TypeScript to combine multiple declarations with the same name into a single definition.

Interface Merging

```
// Basic interface merging
interface User {
  id: number;
  name: string;
}

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

// Merged interface has all properties
```

```typescript
const user: User = {
  id: 1,
  name: "John Doe",
  email: "john@example.com",
  age: 30,
};

// Interface merging with methods
interface Calculator {
  add(a: number, b: number): number;
}

interface Calculator {
  subtract(a: number, b: number): number;
  multiply(a: number, b: number): number;
}

interface Calculator {
  divide(a: number, b: number): number;
}

// All methods are available
const calc: Calculator = {
  add: (a, b) => a + b,
  subtract: (a, b) => a - b,
  multiply: (a, b) => a * b,
  divide: (a, b) => a / b,
};

// Interface merging with generics
interface Container<T> {
  value: T;
}

interface Container<T> {
  getValue(): T;
  setValue(value: T): void;
}

const stringContainer: Container<string> = {
  value: "hello",
  getValue() {
    return this.value;
  },
  setValue(value) {
    this.value = value;
  },
};

// Interface merging with function overloads
interface EventEmitter {
  on(event: "data", listener: (data: string) => void): void;
}
```

```
interface EventEmitter {
  on(event: "error", listener: (error: Error) => void): void;
}

interface EventEmitter {
  on(event: "close", listener: () => void): void;
}

// All overloads are merged
const emitter: EventEmitter = {
  on(event: any, listener: any) {
    // Implementation
  },
};

emitter.on("data", (data) => console.log(data)); // data is string
emitter.on("error", (error) => console.error(error)); // error is Error
emitter.on("close", () => console.log("closed")); // no parameters
```

Namespace Merging

```
// Basic namespace merging
namespace Utils {
  export function formatDate(date: Date): string {
    return date.toISOString().split("T")[0];
  }
}

namespace Utils {
  export function formatTime(date: Date): string {
    return date.toTimeString().split(" ")[0];
  }
}

namespace Utils {
  export interface Config {
    dateFormat: string;
    timeFormat: string;
  }

  export const defaultConfig: Config = {
    dateFormat: "YYYY-MM-DD",
    timeFormat: "HH:mm:ss",
  };
}

// All exports are available
const formattedDate = Utils.formatDate(new Date());
const formattedTime = Utils.formatTime(new Date());
const config = Utils.defaultConfig;
```

```
// Namespace merging with classes
namespace Database {
  export class Connection {
    constructor(public connectionString: string) {}

    connect(): void {
      console.log("Connecting to database...");
    }
  }
}

namespace Database {
  export interface ConnectionOptions {
    ssl: boolean;
    timeout: number;
    retries: number;
  }

  export function createConnection(
    connectionString: string,
    options?: ConnectionOptions
  ): Connection {
    return new Connection(connectionString);
  }
}

// Both class and function are available
const connection = Database.createConnection(
  "postgresql://localhost:5432/mydb"
);
const directConnection = new Database.Connection(
  "postgresql://localhost:5432/mydb"
);
```

Module Augmentation

```
// Augmenting existing modules

// global.d.ts - Augmenting global scope
declare global {
  interface Window {
    myApp: {
      version: string;
      config: Record<string, any>;
      utils: {
        formatCurrency(amount: number): string;
        debounce<T extends (...args: any[]) => any>(fn: T, delay: number): T;
      };
    };
  }
  interface Array<T> {
    last(): T | undefined;
    first(): T | undefined;
    isEmpty(): boolean;
  }

  interface String {
    toTitleCase(): string;
    truncate(length: number, suffix?: string): string;
  }

  interface Number {
    toPercent(decimals?: number): string;
    toCurrency(currency?: string): string;
  }
}

// Implementation (would be in a separate .ts file)
if (typeof window !== "undefined") {
  window.myApp = {
    version: "1.0.0",
    config: {},
    utils: {
      formatCurrency: (amount: number) => `$${amount.toFixed(2)}`,
      debounce: <T extends (...args: any[]) => any>(
        fn: T,
        delay: number
      ): T => {
        let timeoutId: NodeJS.Timeout;
        return ((...args: any[]) => {
          clearTimeout(timeoutId);
          timeoutId = setTimeout(() => fn(...args), delay);
        }) as T;
      },
    },
  };
}

Array.prototype.last = function <T>(this: T[]): T | undefined {
  return this[this.length - 1];
};

Array.prototype.first = function <T>(this: T[]): T | undefined {
  return this[0];
};

Array.prototype.isEmpty = function <T>(this: T[]): boolean {
  return this.length === 0;
};

String.prototype.toTitleCase = function (this: string): string {
  return this.replace(
    /\w\S*/g,
    (txt) => txt.charAt(0).toUpperCase() + txt.substr(1).toLowerCase()
  );
};

String.prototype.truncate = function (
  this: string,
  length: number,
  suffix = "..."
): string {
  return this.length > length ? this.substring(0, length) + suffix : this;
};

Number.prototype.toPercent = function (this: number, decimals = 2): string {
  return `${(this * 100).toFixed(decimals)}%`;
};

Number.prototype.toCurrency = function (
  this: number,
  currency = "USD"
): string {
  return new Intl.NumberFormat("en-US", {
    style: "currency",
    currency,
  }).format(this);
};

// Usage
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.first()); // 1
console.log(numbers.last()); // 5
console.log([].isEmpty()); // true

const text = "hello world";
console.log(text.toTitleCase()); // 'Hello World'
console.log(text.truncate(5)); // 'hello...'

const value = 0.1234;
console.log(value.toPercent()); // '12.34%'
console.log(value.toCurrency()); // '$0.12'
```

Third-Party Library Augmentation

```
// Augmenting Express.js
import { User } from "./types/user";

declare module "express-serve-static-core" {
  interface Request {
    user?: User;
    requestId: string;
    startTime: number;
    correlationId?: string;
  }
  interface Response {
    success(data?: any): Response;
    error(message: string, code?: number): Response;
  }
}

// Implementation (middleware)
import express from "express";

const app = express();

// Add custom methods to Response
app.use((req, res, next) => {
  res.success = function (data?: any) {
    return this.json({
      success: true,
      data,
      timestamp: new Date().toISOString(),
      requestId: req.requestId,
    });
  };

  res.error = function (message: string, code = 500) {
    return this.status(code).json({
      success: false,
      error: { message },
      timestamp: new Date().toISOString(),
      requestId: req.requestId,
    });
  };

  next();
});

// Add request tracking
app.use((req, res, next) => {
  req.requestId = Math.random().toString(36).substr(2, 9);
  req.startTime = Date.now();
  req.correlationId = req.headers["x-correlation-id"] as string;
  next();
});

// Usage with augmented types
app.get("/users/:id", async (req, res) => {
  try {
    const userId = req.params.id;
    const user = await getUserById(userId);

    if (!user) {
      return res.error("User not found", 404);
    }

    res.success(user);
  } catch (error) {
    res.error("Internal server error");
  }
});

// Augmenting Lodash
import _ from "lodash";

declare module "lodash" {
  interface LoDashStatic {
    isValidEmail(value: string): boolean;
    formatPhoneNumber(value: string): string;
    generateId(prefix?: string): string;
  }

  interface LoDashImplicitWrapper<TValue> {
    isValidEmail(): boolean;
    formatPhoneNumber(): LoDashImplicitWrapper<string>;
  }
}

// Implementation
_.mixin({
  isValidEmail: (value: string): boolean => {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value);
  },

  formatPhoneNumber: (value: string): string => {
    const cleaned = value.replace(/\D/g, "");
    const match = cleaned.match(/^(\d{3})(\d{3})(\d{4})$/);
    return match ? `(${match[1]}) ${match[2]}-${match[3]}` : value;
  },

  generateId: (prefix = "id"): string => {
    return `${prefix}_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
  },
});

// Usage
const email = "user@example.com";
console.log(_.isValidEmail(email)); // true
console.log(_(email).isValidEmail()); // true

const phone = "1234567890";
console.log(_.formatPhoneNumber(phone)); // '(123) 456-7890'
console.log(_(phone).formatPhoneNumber().value()); // '(123) 456-7890'

const id = _.generateId("user"); // 'user_1634567890123_abc123def'

function getUserById(id: string): Promise<User | null> {
  // Implementation
  return Promise.resolve(null);
}
```

Ambient Declarations

Ambient declarations describe the shape of existing JavaScript code without providing implementation.

Basic Ambient Declarations

```
// Declaring global variables
declare const VERSION: string;
declare const BUILD_TIME: number;
declare const IS_PRODUCTION: boolean;

// Declaring global functions
declare function gtag(command: string, targetId: string, config?: object): void;
declare function fbq(
  command: string,
  eventName: string,
  parameters?: object
): void;

// Declaring global objects
declare const analytics: {
  track(event: string, properties?: Record<string, any>): void;
  identify(userId: string, traits?: Record<string, any>): void;
  page(name?: string, properties?: Record<string, any>): void;
};

// Usage (no implementation needed)
console.log(`Version: ${VERSION}`);
gtag("config", "GA_MEASUREMENT_ID");
analytics.track("page_view", { page: "/home" });

// Declaring classes
declare class ThirdPartyWidget {
  constructor(element: HTMLElement, options?: WidgetOptions);
  render(): void;
  destroy(): void;
  on(event: string, callback: Function): void;
  off(event: string, callback?: Function): void;
}

interface WidgetOptions {
  theme?: "light" | "dark";
  size?: "small" | "medium" | "large";
  autoplay?: boolean;
}

// Usage
const widget = new ThirdPartyWidget(document.getElementById("widget")!, {
  theme: "dark",
  size: "large",
});
```

```
widget.render();
widget.on("ready", () => console.log("Widget is ready"));

// Declaring modules
declare module "legacy-library" {
  export interface Config {
    apiKey: string;
    baseUrl: string;
    timeout?: number;
  }

  export class Client {
    constructor(config: Config);
    request(endpoint: string, data?: any): Promise<any>;
    upload(file: File, options?: UploadOptions): Promise<UploadResult>;
  }

  export interface UploadOptions {
    onProgress?: (progress: number) => void;
    maxSize?: number;
  }

  export interface UploadResult {
    id: string;
    url: string;
    size: number;
  }

  export function initialize(config: Config): Client;
  export const version: string;
}

// Usage
import { initialize, Client, Config } from "legacy-library";

const config: Config = {
  apiKey: "your-api-key",
  baseUrl: "https://api.example.com",
  timeout: 5000,
};

const client = initialize(config);
client.request("/users").then((users) => console.log(users));
```

Environment-Specific Declarations

```
// Node.js environment declarations
declare namespace NodeJS {
  interface ProcessEnv {
    NODE_ENV: "development" | "production" | "test";
    PORT: string;
    DATABASE_URL: string;
    JWT_SECRET: string;
    REDIS_URL?: string;
    SMTP_HOST?: string;
    SMTP_PORT?: string;
    SMTP_USER?: string;
    SMTP_PASS?: string;
    AWS_ACCESS_KEY_ID?: string;
    AWS_SECRET_ACCESS_KEY?: string;
    AWS_REGION?: string;
    STRIPE_SECRET_KEY?: string;
    STRIPE_WEBHOOK_SECRET?: string;
  }

  interface Global {
    __DEV__: boolean;
    __TEST__: boolean;
    __PROD__: boolean;
  }
}

// Browser environment declarations
declare interface Window {
  gtag?: (
    command: "config" | "event" | "set",
    targetIdOrName: string,
    configOrParameters?: object
  ) => void;

  fbq?: (
    command: "track" | "trackCustom" | "init",
    eventNameOrPixelId: string,
    parameters?: object
  ) => void;

  dataLayer?: any[];

  Stripe?: {
    (publishableKey: string, options?: StripeOptions): StripeInstance;
  };

  PayPal?: {
    Buttons(options: PayPalButtonsOptions): PayPalButtonsInstance;
  };

  grecaptcha?: {
    ready(callback: () => void): void;
    execute(siteKey: string, options: { action: string }): Promise<string>;
  };
}

interface StripeOptions {
  apiVersion?: string;
  locale?: string;
}

interface StripeInstance {
  elements(): StripeElements;
  confirmCardPayment(clientSecret: string, data?: any): Promise<any>;
  createToken(element: any, data?: any): Promise<any>;
}

interface StripeElements {
  create(type: string, options?: any): StripeElement;
}

interface StripeElement {
  mount(selector: string): void;
  on(event: string, callback: Function): void;
}

interface PayPalButtonsOptions {
  createOrder?: (data: any, actions: any) => Promise<string>;
  onApprove?: (data: any, actions: any) => Promise<void>;
  onError?: (error: any) => void;
  style?: {
    layout?: "vertical" | "horizontal";
    color?: "gold" | "blue" | "silver" | "white" | "black";
    shape?: "rect" | "pill";
    label?: "paypal" | "checkout" | "buynow" | "pay";
  };
}

interface PayPalButtonsInstance {
  render(selector: string): Promise<void>;
}

// Usage
if (typeof window !== "undefined") {
  // Google Analytics
  window.gtag?.("config", "GA_MEASUREMENT_ID");

  // Facebook Pixel
  window.fbq?.("track", "PageView");

  // Stripe
  if (window.Stripe) {
    const stripe = window.Stripe("pk_test_...");
    const elements = stripe.elements();
    const cardElement = elements.create("card");
    cardElement.mount("#card-element");
  }

  // PayPal
  window.PayPal?.Buttons({
    createOrder: async (data, actions) => {
      // Implementation
```

```
        return "order-id";
      },
      onApprove: async (data, actions) => {
        // Implementation
      },
    }).render("#paypal-button-container");
  }

  // Node.js usage
  if (typeof process !== "undefined") {
    const port = process.env.PORT || "3000";
    const isDev = process.env.NODE_ENV === "development";
    const dbUrl = process.env.DATABASE_URL;
  }
```

Creating Type Definition Files

```
// types/my-library.d.ts

// For a simple JavaScript library
declare module "simple-validator" {
  interface ValidationRule {
    required?: boolean;
    minLength?: number;
    maxLength?: number;
    pattern?: RegExp;
    custom?: (value: any) => boolean | string;
  }

  interface ValidationSchema {
    [field: string]: ValidationRule;
  }

  interface ValidationResult {
    isValid: boolean;
    errors: Record<string, string[]>;
  }

  export class Validator {
    constructor(schema: ValidationSchema);
    validate(data: Record<string, any>): ValidationResult;
    addRule(name: string, rule: ValidationRule): void;
  }

  export function validate(
    data: Record<string, any>,
    schema: ValidationSchema
  ): ValidationResult;

  export const rules: {
    email: ValidationRule;
    url: ValidationRule;
    numeric: ValidationRule;
    alpha: ValidationRule;
    alphanumeric: ValidationRule;
  };
}

// For a library with default export
declare module "date-formatter" {
  interface FormatOptions {
    locale?: string;
    timezone?: string;
    format?: string;
  }

  interface DateFormatter {
    format(date: Date, options?: FormatOptions): string;
    parse(dateString: string, format?: string): Date;
    addDays(date: Date, days: number): Date;
    addMonths(date: Date, months: number): Date;
    addYears(date: Date, years: number): Date;
    isBefore(date1: Date, date2: Date): boolean;
    isAfter(date1: Date, date2: Date): boolean;
    isSame(date1: Date, date2: Date, unit?: "day" | "month" | "year"): boolean;
  }

  const formatter: DateFormatter;
  export = formatter;
}

// For a library with mixed exports
declare module "http-client" {
  interface RequestConfig {
    method?: "GET" | "POST" | "PUT" | "DELETE" | "PATCH";
    headers?: Record<string, string>;
    timeout?: number;
    retries?: number;
    baseURL?: string;
  }

  interface Response<T = any> {
    data: T;
    status: number;
    statusText: string;
    headers: Record<string, string>;
  }

  interface HttpClient {
    get<T = any>(url: string, config?: RequestConfig): Promise<Response<T>>;
    post<T = any>(
      url: string,
      data?: any,
      config?: RequestConfig
    ): Promise<Response<T>>;
    put<T = any>(
      url: string,
      data?: any,
      config?: RequestConfig
    ): Promise<Response<T>>;
    delete<T = any>(url: string, config?: RequestConfig): Promise<Response<T>>;
    patch<T = any>(
      url: string,
      data?: any,
      config?: RequestConfig
    ): Promise<Response<T>>;
  }

  export function create(config?: RequestConfig): HttpClient;
  export function get<T = any>(
    url: string,
    config?: RequestConfig
  ): Promise<Response<T>>;
  export function post<T = any>(
    url: string,
    data?: any,
    config?: RequestConfig
  ): Promise<Response<T>>;

  export default create;
}

// For a jQuery plugin
declare namespace JQuery {
  interface JQuery {
    myPlugin(options?: MyPluginOptions): JQuery;
    myPlugin(method: "destroy"): JQuery;
    myPlugin(method: "update", data: any): JQuery;
    myPlugin(method: "getValue"): any;
  }
}

interface MyPluginOptions {
  theme?: "light" | "dark";
  animation?: boolean;
  duration?: number;
  onInit?: () => void;
  onChange?: (value: any) => void;
}

// Usage examples
import { Validator, validate, rules } from "simple-validator";
import dateFormatter from "date-formatter";
import httpClient, { create, get } from "http-client";

// Simple validator
const validator = new Validator({
  email: { ...rules.email, required: true },
  password: { required: true, minLength: 8 },
});

const result = validator.validate({
  email: "user@example.com",
  password: "password123",
});

// Date formatter
const formatted = dateFormatter.format(new Date(), {
  locale: "en-US",
  format: "YYYY-MM-DD",
});

// HTTP client
const client = create({ baseURL: "https://api.example.com" });
const response = await client.get<User[]>("/users");

// jQuery plugin
$("#my-element").myPlugin({
  theme: "dark",
  animation: true,
  onChange: (value) => console.log("Value changed:", value),
});
```

Working with @types Packages

Installing and Using Type Definitions

```
// Install type definitions for popular libraries
// npm install --save-dev @types/node @types/express @types/lodash @types/jest

// Using Express with types
import express, { Request, Response, NextFunction } from "express";
import { User } from "./types/user";

const app = express();

// Middleware with proper typing
const authMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const token = req.headers.authorization;

  if (!token) {
    return res.status(401).json({ error: "No token provided" });
  }

  // Verify token and attach user
  req.user = { id: "1", name: "John Doe" } as User;
  next();
};

// Route handlers with typing
```

```
app.get("/users/:id", authMiddleware, async (req: Request, res: Response) => {
  const userId = req.params.id;
  const currentUser = req.user; // TypeScript knows this exists due to augmentation

  try {
    const user = await getUserById(userId);
    res.json(user);
  } catch (error) {
    res.status(500).json({ error: "Internal server error" });
  }
});

// Using Lodash with types
import _ from "lodash";

const users: User[] = [
  {
    id: "1",
    name: "John",
    email: "john@example.com",
    age: 30,
    isActive: true,
    createdAt: new Date(),
    updatedAt: new Date(),
  },
  {
    id: "2",
    name: "Jane",
    email: "jane@example.com",
    age: 25,
    isActive: false,
    createdAt: new Date(),
    updatedAt: new Date(),
  },
];

// TypeScript knows the return types
const activeUsers = _.filter(users, "isActive"); // User[]
const userNames = _.map(users, "name"); // string[]
const groupedByAge = _.groupBy(users, "age"); // Record<string, User[]>
const sortedUsers = _.sortBy(users, ["age", "name"]); // User[]

// Using Jest with types
import { describe, it, expect, beforeEach, afterEach } from "@jest/globals";

describe("User Service", () => {
  let userService: UserService;

  beforeEach(() => {
    userService = new UserService();
  });

  afterEach(() => {
    // Cleanup
  });

  it("should create a user", async () => {
    const userData = {
      name: "Test User",
      email: "test@example.com",
      age: 25,
      isActive: true,
    };

    const user = await userService.createUser(userData);

    expect(user).toBeDefined();
    expect(user.name).toBe(userData.name);
    expect(user.email).toBe(userData.email);
  });

  it("should throw error for invalid email", async () => {
    const userData = {
      name: "Test User",
      email: "invalid-email",
      age: 25,
      isActive: true,
    };

    await expect(userService.createUser(userData)).rejects.toThrow(
      "Invalid email"
    );
  });
});

class UserService {
  async createUser(
    userData: Omit<User, "id" | "createdAt" | "updatedAt">
  ): Promise<User> {
    // Implementation
    return {
      id: "1",
      ...userData,
      createdAt: new Date(),
      updatedAt: new Date(),
    };
  }
}
```

Creating Custom @types Packages

```
// package.json for @types/my-custom-library
{
  "name": "@types/my-custom-library",
  "version": "1.0.0",
  "description": "TypeScript definitions for my-custom-library",
  "types": "index.d.ts",
  "typesPublisherContentHash": "...",
  "typeScriptVersion": "4.0",
  "dependencies": {},
  "devDependencies": {
    "typescript": "^4.0.0"
  }
}

// index.d.ts
// Type definitions for my-custom-library 2.1
// Project: https://github.com/example/my-custom-library
// Definitions by: Your Name <https://github.com/yourusername>
// Definitions: https://github.com/DefinitelyTyped/DefinitelyTyped

export interface LibraryConfig {
  apiKey: string;
  baseUrl?: string;
  timeout?: number;
  debug?: boolean;
}

export interface ApiResponse<T = any> {
  success: boolean;
  data?: T;
  error?: string;
  timestamp: string;
}

export class MyLibrary {
  constructor(config: LibraryConfig);

  get<T = any>(endpoint: string): Promise<ApiResponse<T>>;
  post<T = any>(endpoint: string, data: any): Promise<ApiResponse<T>>;
  put<T = any>(endpoint: string, data: any): Promise<ApiResponse<T>>;
  delete<T = any>(endpoint: string): Promise<ApiResponse<T>>;

  setConfig(config: Partial<LibraryConfig>): void;
  getConfig(): LibraryConfig;
}

export function createClient(config: LibraryConfig): MyLibrary;
export function isValidConfig(config: any): config is LibraryConfig;

export const version: string;
export const defaultConfig: Partial<LibraryConfig>;

// For libraries with global side effects
declare global {
  interface Window {
    MyLibrary?: typeof MyLibrary;
  }
```

```
}

// tests/index.ts - Test the type definitions
import { MyLibrary, createClient, LibraryConfig, ApiResponse } from 'my-custom-library';

// Test basic usage
const config: LibraryConfig = {
  apiKey: 'test-key',
  baseUrl: 'https://api.example.com',
  timeout: 5000
};

const client = new MyLibrary(config);
const client2 = createClient(config);

// Test API calls
client.get<{ users: any[] }>('/users').then((response: ApiResponse<{ users: any[] }>) => {
  if (response.success && response.data) {
    console.log(response.data.users);
  }
});

client.post('/users', { name: 'John' }).then((response: ApiResponse) => {
  console.log(response.success);
});

// Test configuration
client.setConfig({ timeout: 10000 });
const currentConfig: LibraryConfig = client.getConfig();
```

Best Practices

✅ Good Practices

```
// Use interface merging for extending existing types
interface User {
  id: string;
  name: string;
}

interface User {
  email: string; // Extends the User interface
}

// Create specific ambient declarations for third-party libraries
declare module "specific-library" {
  export interface Config {
    apiKey: string;
    baseUrl: string;
  }

  export class Client {
    constructor(config: Config);
    request(endpoint: string): Promise<any>;
  }
}

// Use namespace merging for organizing related functionality
namespace Utils {
  export function formatDate(date: Date): string {
    return date.toISOString();
  }
}

namespace Utils {
  export function parseDate(dateString: string): Date {
    return new Date(dateString);
  }
}

// Augment global types carefully and specifically
declare global {
  interface Window {
    mySpecificLibrary?: {
      version: string;
      init(): void;
    };
  }
}

// Use proper type constraints in ambient declarations
declare function processData<T extends Record<string, any>>(data: T): T;
```

❌ Avoid

```
// Don't over-augment global types
declare global {
  interface Window {
    [key: string]: any; // ❌ Too broad
  }

  interface Array<T> {
    customMethod(): any; // ❌ Pollutes all arrays
  }
}

// Don't create conflicting interface merges
interface User {
  id: string;
}
```

```
interface User {
  id: number; // ❌ Conflicts with previous declaration
}

// Don't use any in ambient declarations
declare module "some-library" {
  export function doSomething(data: any): any; // ❌ Not type-safe
}

// Don't create overly broad ambient declarations
declare const globalVariable: any; // ❌ Should be more specific

// Don't ignore existing type definitions
// If @types/library exists, don't create your own unless necessary
```

Summary Checklist

- □ Use interface merging to extend existing types
- □ Leverage namespace merging for organizing code
- □ Create ambient declarations for JavaScript libraries
- □ Augment third-party library types when needed
- □ Use module augmentation for extending external modules
- □ Create proper type definition files for custom libraries
- □ Install and use @types packages for popular libraries
- □ Be specific and careful with global augmentations
- □ Test your type definitions thoroughly
- □ Document your ambient declarations clearly

Next Steps

Now that you understand declaration merging and ambient declarations, let's explore working with third-party types and creating custom type definitions.

Continue to: Working with Third-Party Types