Scalable TypeScript Patterns

Structures for Building Reliable, Maintainable Code

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Classes and Object-Oriented Programming

In TypeScript, classes are pivotal to writing modular and maintainable object-oriented applications. Classes help organize code into logical units with well-defined behavior and encapsulate data to foster reuse.

Basic Classes and Inheritance

Declaring Classes with Constructors, Properties, and Methods

Classes in TypeScript follow the familiar object-oriented paradigm. A class comprises properties to store data and methods to implement behaviors.

- **Constructor**: A special method that initializes object properties when a new instance is created.
- **Properties**: Variables associated with the class, typically initialized via the constructor or default values.
- **Methods**: Functions that define the behaviors of the class.

```
class Person {
    // Properties
    private firstName: string;
    private lastName: string;

    // Constructor
    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Method
    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}

// Creating an instance
const john = new Person("John", "Doe");
console.log(john.getFullName()); // Outputs: "John Doe"
```

Key Points:

- The `private` keyword ensures encapsulation, restricting access to these properties from outside the class.
- `getFullName()` is a public method, so it can be called directly on any instance.

Access Modifiers (public, private, protected)

TypeScript includes three access modifiers to control property and method visibility:

- **Public**: Accessible from anywhere (default).
- **Private**: Only accessible within the class.

- **Protected**: Accessible within the class and its subclasses.

```
class Car {
    // Public property
    public brand: string;

    // Private property
    private model: string;

    // Protected property
    protected year: number;

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

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

const myCar = new Car("Tesla", "Model 3", 2020);
console.log(myCar.brand); // Accessible
// console.log(myCar.model); // Error: private property
```

Key Takeaways:

- Access modifiers provide control over what information is exposed.
- Using modifiers correctly leads to more secure, modular code.

Inheritance and Method Overriding

Inheritance allows classes to extend other classes and inherit their properties and methods. This is useful for creating specialized classes without duplicating common functionality.

```
class Vehicle {
    protected brand: string;

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

    public start(): string {
        return `${this.brand} vehicle is starting.`;
    }
}

class Bike extends Vehicle {
    private type: string;

    constructor(brand: string, type: string) {
        super(brand);
        this.type = type;
    }
```

```
    public start(): string {
        // Method overriding
        return `${this.brand} bike (${this.type}) is starting.`;
    }
}

const motorcycle = new Bike("Honda", "Cruiser");
console.log(motorcycle.start()); // Outputs: "Honda bike (Cruiser) is
   starting."
```

Key Concepts:

- **Super**: Refers to the parent class. `super()` invokes the parent class constructor.
- **Overriding Methods**: Allows child classes to redefine inherited methods for specialized behavior.

Interfaces and Types

Interfaces and types are essential components of TypeScript's type system, providing a structured framework for your data models. They ensure objects have consistent shapes and enable developers to write code with greater confidence.

Understanding Interfaces

Defining and Implementing Interfaces

An interface acts as a blueprint that defines the required properties for an object. Classes implementing an interface ensure they include all specified properties.

```
// Interface definition
interface Employee {
    name: string;
    role: string;
    salary: number;
}

// Implementing the interface in a class
class Manager implements Employee {
    name: string;
    role: string;
    salary: number;

    constructor(name: string, salary: number) {
        this.name = name;
        this.role = "Manager";
        this.salary = salary;
    }

    getDetails(): string {
        return `${this.name}, Role: ${this.role}, Salary: $${this.
   salary}`;
    }
}
```

```
const manager = new Manager("Alice", 120000);
console.log(manager.getDetails()); // Outputs: "Alice, Role: Manager,
   Salary: $120000"
```

Explanation:

- The `Employee` interface defines the shape of an object with `name`, `role`, and `salary` properties.
- The `Manager` class implements the `Employee` interface, ensuring all three properties are present in the class.
- The `getDetails()` method outputs information based on these required properties.

Structural Typing

TypeScript employs structural typing (also called "duck typing"), meaning that object compatibility is determined by comparing shapes instead of explicit declarations.

```
interface Point {
    x: number;
    y: number;
}

function printCoordinates(point: Point): void {
    console.log(`X: ${point.x}, Y: ${point.y}`);
}

// Structural typing allows an object with matching properties to be
   compatible
const pointLike = { x: 10, y: 20, label: "Point A" };
printCoordinates(pointLike); // Outputs: "X: 10, Y: 20"
```

Explanation:

- The `Point` interface requires objects to have `x` and `y` properties.
- The `printCoordinates` function accepts any object with these properties, regardless of additional properties like `label`.
- The `pointLike` object, though containing an extra `label` property, is compatible with `Point` because it matches the required shape.

Using Optional and Readonly Properties

Interfaces can declare properties that are either optional or read-only.

```
// An interface with optional and read-only properties
interface Product {
    id: number;
    name: string;
    description?: string; // Optional property
    readonly price: number; // Read-only property
}

const product: Product = {
```

```
    id: 1,
    name: "Laptop",
    price: 999.99
};

// product.price = 899.99; // Error: Cannot assign to 'price' because
   it is a read-only property
```

Explanation:

- The `Product` interface uses a `?` to mark the `description` property as optional, meaning it can be omitted when creating a `Product` object.
- The `readonly` modifier ensures that `price` cannot be reassigned after the initial value has been set.

Types vs. Interfaces

Comparing Interfaces and Types

In TypeScript, both interfaces and type aliases define object shapes, but they have differences that influence their usage.

```
// Using a type alias to define a union type
type Status = "active" | "inactive" | "suspended";

function updateStatus(status: Status) {
    console.log(`Updating status to: ${status}`);
}

updateStatus("active"); // Valid
// updateStatus("archived"); // Error: Type '"archived"' is not
   assignable to type 'Status'
```

Explanation:

- The `Status` type alias defines a union type, restricting valid values to `"active"`, `"inactive"`, and `"suspended"`.
- When `updateStatus` is called with a valid status, it logs the update.
- Calling `updateStatus` with an invalid status (e.g., `"archived"`) results in a compile-time error.

Conclusion:

- Interfaces are excellent for defining the shape of objects that need to extend or be implemented across different files.
- Type aliases are more suitable for union types, intersections, and combining multiple types efficiently.

Error Handling

Effective error handling is important for building robust applications. TypeScript's static typing and enhanced features make it easier to catch issues early, but runtime errors still need careful management.

Error Handling Best Practices in TypeScript

- **Fail Early**: Detect and handle errors as soon as possible to prevent issues from propagating further down.
- **Use Exhaustive Checks**: Ensure all potential scenarios are accounted for with union types, type guards, and conditional statements.
- **Consistent Error Responses**: Return consistent error objects or messages, helping developers identify and address issues quickly.

```
type Response = { success: true; data: string } | { success: false;
    error: string };

function fetchData(url: string): Response {
    // Simulate a failed request for demonstration
    const failed = Math.random() > 0.5;

    if (failed) {
        return { success: false, error: "Network Error" };
    } else {
        return { success: true, data: "Fetched Data" };
    }
}

const response = fetchData("https://example.com/api");
if (response.success) {
    console.log(response.data);
} else {
    console.error(response.error); // Consistent error handling
}
```

Explanation:

- The `Response` union type enforces consistent handling of success and error cases.
- By checking the `success` property, the error message is consistently accessed in the failure scenario.

Using Custom Error Classes and `try-catch` Blocks

Custom error classes enable you to encapsulate error details and provide more meaningful error messages. The `try-catch` construct helps manage these errors gracefully.

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

function fetchResource(url: string): string {
    const failed = Math.random() > 0.5;

    if (failed) {
        // Throw custom error
```

```
        throw new NetworkError(500, `Unable to fetch data from ${url}`)
    ;
    }

    return "Resource Data";
}

try {
    const data = fetchResource("https://example.com/resource");
    console.log(data);
} catch (error) {
    if (error instanceof NetworkError) {
        console.error(`Error (${error.statusCode}): ${error.message}`);
    } else {
        console.error(`General Error: ${error.message}`);
    }
}
```

Explanation:

- `NetworkError` extends the `Error` class to include an additional `statusCode` property.
- The `try-catch` block handles the error gracefully, with specific logic for custom errors.
- `instanceof` ensures errors are checked specifically, providing detailed error information.

Strategies for Global Error Handling in Applications

Handling errors globally ensures that unhandled exceptions don't crash your application and provides a consistent fallback mechanism.

- **Global Error Handlers**: Implement global handlers to capture errors and provide fallback behavior. In Node.js, use `process.on` for global errors. In client-side frameworks, leverage global error boundaries or event listeners.
- **Logging and Monitoring**: Implement logging systems to capture error details, stack traces, and user actions leading up to the error. This aids debugging and future prevention.
- **Graceful Degradation**: Provide default or fallback values when data or functionality cannot be fetched or used due to errors.

```
process.on("uncaughtException", (error) => {
    console.error("Uncaught Exception:", error.message);
    // Log or notify the team about the exception
    process.exit(1); // Exit the application safely
});

process.on("unhandledRejection", (reason, promise) => {
    console.error("Unhandled Rejection at:", promise, "reason:", reason
    );
    // Handle the rejection properly here
});
```

Explanation:

- `process.on("uncaughtException")` listens for uncaught exceptions, logs the error message, and exits the application gracefully.
- `process.on("unhandledRejection")` ensures any unhandled promise rejection is properly logged and handled.

Debugging Techniques

Debugging is essential to maintaining code quality and resolving issues. TypeScript supports advanced debugging workflows using source maps, IDEs, and browser tools.

Integrating TypeScript Source Maps for Debugging

Source maps help map the compiled JavaScript code back to its original TypeScript source, making debugging significantly easier.

Enable Source Maps: Make sure that the `sourceMap` option is enabled in the `tsconfig.json` file.

```
{
    "compilerOptions": {
        "sourceMap": true,
        "outDir": "./dist"
    }
}
```

Explanation:

- The `"sourceMap": true` option instructs the TypeScript compiler to generate corresponding `.map` files for each JavaScript file.
- This allows debuggers to trace errors back to the original TypeScript source code.

Setting Up Effective Debugging Workflows with IDEs and Browser Tools

A well-structured debugging workflow can speed up issue resolution.

Visual Studio Code: This IDE provides an integrated debugger that works seamlessly with TypeScript projects.
Launch Configuration: Create or update the `launch.json` file to add breakpoints, inspect variables, and step through code.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Debug TypeScript",
            "program": "${workspaceFolder}/dist/index.js",
```

```
            "preLaunchTask": "tsc: build - tsconfig.json",
            "outFiles": ["${workspaceFolder}/dist/**/*.js"]
        }
    ]
}
```

Explanation:

- The configuration above launches a Node.js program with debugging capabilities.
- The `"preLaunchTask"` runs TypeScript compilation to generate up-to-date JavaScript files.
- The `"outFiles"` array tells the debugger to associate these compiled files with the original TypeScript code.

Browser Developer Tools: Chrome DevTools and Firefox Developer Tools can directly debug TypeScript code if source maps are enabled.

Open the developer tools and set breakpoints in the "Sources" tab to inspect variables and step through code.

Identifying and Solving Common Debugging Challenges in TypeScript

Scope Issues: Problems with variable scope can lead to unexpected values.

Solution: Use `let` and `const` to ensure variables are block-scoped and inspect variable values through the debugger.

Asynchronous Code Issues: Unhandled promise rejections and asynchronous behavior can cause errors that are challenging to track.

Solution: Use the `async/await` pattern or `.catch()` for error handling and ensure debugging workflows handle rejected promises.

```
async function fetchData(url: string): Promise<string> {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        return await response.text();
    } catch (error) {
        console.error(`Fetch error: ${error.message}`);
        return "Error occurred";
    }
}
```

Explanation:

- The `try-catch` block ensures any asynchronous error is caught and logged, providing clarity on failures.

Type Errors: Misaligned types often lead to runtime errors.

Solution: Enable strict type checking (`strict` option in `tsconfig.json`), use type guards, and ensure all variables are properly typed.

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

function printLength(value: unknown): void {
    if (isString(value)) {
        console.log(`Length: ${value.length}`);
    } else {
        console.error("Value is not a string");
    }
}
```

Explanation:

- `isString` is a type guard that checks whether a value is a string.
- The `printLength` function uses this guard to handle different types safely.

Built-In Utility Types

TypeScript provides a collection of built-in utility types to help manipulate and transform object types. These utilities, such as `Partial`, `Pick`, `Record`, and `Omit`, can simplify your code and reduce redundancy by creating new types from existing ones. Let's explore each type with practical examples.

Partial

Overview: The `Partial` type makes all properties of an object type optional, which is particularly useful when updating or modifying a subset of an object's properties.

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

// Making all properties optional using Partial
function updateUser(id: number, updates: Partial<User>): User {
    const originalUser: User = { id, name: "John Doe", email: "john.
   doe@example.com", isActive: true };

    // Apply updates using spread operator
    return { ...originalUser, ...updates };
}

const updatedUser = updateUser(1, { email: "new.email@example.com",
   isActive: false });
console.log(updatedUser); // Outputs: { id: 1, name: "John Doe", email:
    "new.email@example.com", isActive: false }
```

Explanation:

- `Partial<User>` creates a new type with all `User` properties set as optional.
- The `updateUser` function can accept an object with any combination of `User` properties for selective updates.

Pick

Overview: The `Pick` type extracts a subset of properties from an object type. This is helpful when working with APIs or specific data requirements.

```
interface BlogPost {
    id: string;
    title: string;
    content: string;
    author: string;
    createdAt: Date;
    updatedAt: Date;
}

// Creating a type that only includes title and author
type BlogPostPreview = Pick<BlogPost, "title" | "author">;

const previewPost: BlogPostPreview = { title: "TypeScript Tips", author
    : "Jane Smith" };
console.log(previewPost); // Outputs: { title: "TypeScript Tips",
    author: "Jane Smith" }
```

Explanation:

- `Pick<BlogPost, "title" | "author">` creates a new type with only the `title` and `author` properties of the `BlogPost` type.
- This reduces the data footprint when sharing preview information.

Record

Overview: The `Record` type creates a map-like object structure with specified keys and values, making it useful for constructing collections of consistent types.

```
type UserRoles = "admin" | "editor" | "viewer";
const roleDescriptions: Record<UserRoles, string> = {
    admin: "Has full access to all system features",
    editor: "Can edit and manage content",
    viewer: "Can only view published content",
};

console.log(roleDescriptions["admin"]); // Outputs: "Has full access to
    all system features"
```

Explanation:

- `Record<UserRoles, string>` creates an object type where keys are `UserRoles` and values are `string`.
- Each role description is strictly typed to match the specified roles, providing consistency.

Omit

Overview: The `Omit` type removes specific properties from an object type, often used to hide sensitive or unnecessary data.

```
interface FullUserProfile {
    id: number;
    name: string;
    email: string;
    password: string;
    createdAt: Date;
    lastLogin: Date;
}

// Creating a public profile without password or lastLogin
type PublicUserProfile = Omit<FullUserProfile, "password" | "lastLogin"
    >;

const publicProfile: PublicUserProfile = {
    id: 123,
    name: "Alice Smith",
    email: "alice.smith@example.com",
    createdAt: new Date("2024-01-01"),
};

console.log(publicProfile); // Outputs: { id: 123, name: "Alice Smith",
    email: "alice.smith@example.com", createdAt: Date }
```

Explanation:

- `Omit<FullUserProfile, "password" | "lastLogin">` creates a new type excluding sensitive information like `password` and `lastLogin`.
- The resulting `PublicUserProfile` ensures that data shared externally is safe.