Week 7: Advanced TypeScript Concepts

FH
University of Applied Sciences
TECHNIKUM
WIEN

Week 7: Advanced TypeScript Concepts

This week we'll cover:

- Recap of TypeScript Basics
- Advanced TypeScript Features
- Managing Configuration and Options
- Refactoring the Memory Game with TypeScript
- Example Solution Overview

Recap of TypeScript Basics

Key Concepts:

1. **Type Annotations**:
   - Define types for variables, parameters, and return values.

```
let message: string = "Hello, TypeScript!";
```

2. **Interfaces**:
   - Enforce object structure with `interface`.

```
interface Card {
    id: number;
    symbol: string;
}
```

3. **Compilation**:
   - Use `tsc` to compile TypeScript to JavaScript.

```
tsc app.ts
```

Advanced TypeScript Features

1. Enums

- Define named constants for better code readability.

```
enum GameState {
    NotStarted,
    InProgress,
    Completed,
}
```

- Use in conditional logic:

```
if (gameState === GameState.InProgress) {
    console.log("Game is running");
}
```

2. Union Types

- Combine multiple types into a single variable.

```
type Player = string | null;

let currentPlayer: Player = "Alice";
currentPlayer = null; // Valid
```

3. Generics

- Add flexibility while maintaining type safety.

```
function shuffle<T>(array: T[]): T[] {
    return array.sort(() => Math.random() - 0.5);
}

const numbers = shuffle<number>([1, 2, 3, 4]);
```

4. Sync and Async Functions

- Understand synchronous and asynchronous functions for managing operations.

Synchronous Functions:

- Execute line by line, blocking further execution until complete.

```
function syncGreet(name: string): string {
    return `Hello, ${name}!`;
}
console.log(syncGreet("Alice")); // Immediate execution
```

Asynchronous Functions:

- Use `async` and `await` for non-blocking operations.

```
async function asyncGreet(name: string): Promise<string> {
    return new Promise((resolve) => {
        setTimeout(() => resolve(`Hello, ${name}!`), 1000);
    });
}

asyncGreet("Bob").then(console.log); // Delayed execution
```

Managing Configuration and Options

Define Game Settings

Use an `interface` to manage configurable options.

```typescript
interface GameSettings {
    numPairs: number;
    isMultiplayer: boolean;
    playerNames: string[];
}

const settings: GameSettings = {
    numPairs: 6,
    isMultiplayer: true,
    playerNames: ["Alice", "Bob"]
};
```

Example Solution: Memory Game Refactoring

Overview of Changes

1. **Core Components:**
    - `GameManager.ts`: Handles game logic such as card generation, shuffling, and state updates.
    - `StateManager.ts`: Manages application state and updates dynamically.
    - `UIManager.ts`: Manages user interface updates like the scoreboard and turn highlights.
2. **Key Features Implemented:**
    - Multiplayer support with score tracking and turn switching.
    - Dynamic card generation based on configurable number of pairs.
    - TypeScript features like `interfaces`, `enums`, and `generics` to ensure type safety and maintainability.

Example Code Snippets

Game Manager

Manages card generation and shuffling:

```
export class GameManager {
    constructor(cardSymbols: string[]) {
        this.cards = this.generateShuffledCards(cardSymbols);
    }

    private generateShuffledCards(symbols: string[]): Card[] {
        const deck = symbols.flatMap((symbol) => [
            { id: Math.random(), symbol, isFlipped: false, isMatched: false },
            { id: Math.random(), symbol, isFlipped: false, isMatched: false }
        ]);
        return this.shuffle(deck);
    }

    private shuffle<T>(array: T[]): T[] {
        return array.sort(() => Math.random() - 0.5);
    }
}
```

UI Manager

Updates the scoreboard dynamically:

```typescript
export class UIManager {
    updateScoreboard(players: { name: string; score: number }[]): void {
        const scoreboard = document.getElementById("scoreboard")!;
        scoreboard.innerHTML = players
            .map((player) => `${player.name}: ${player.score}`)
            .join("<br>");
    }
}
```

Weekly Exercise: Memory Game Refactoring

Exercise Objectives:

1. **Refactor to TypeScript**:
   - Convert the Memory Game to TypeScript, ensuring type safety.
2. **Add Configurability**:
   - Allow players to configure the number of pairs.
3. **Implement Multiplayer**:
   - Track scores for multiple players.
   - Switch turns automatically.
4. **Enhance the UI**:
   - Display scores, turns, and game state dynamically.
   - Highlight the active player.
5. **Ensure Maintainability**:
   - Use TypeScript features like interfaces and enums.

Submission Requirements

1. **Fully Refactored Project**:
    - TypeScript implementation.
2. **Working Features**:
    - Configurable settings.
    - Multiplayer mode.
3. **Code Documentation**:
    - Inline comments and explanation for all components.
4. **ZIP or GitHub Submission**.

Bonus Challenges

- Add a hint system.
- Implement animations for flips and matches.
- Display an end-game summary.

Summary and Q&A

- **Core Focus:**
  - Refactoring and TypeScript application.
  - Dynamic and maintainable game architecture.
  - UI updates and multiplayer functionality.

Questions?