Week 13: Finalizing the Course

FH University of Applied Sciences
TECHNIKUM WIEN

Week 13: Finalizing the Course

Topics for This Week

1. Advanced Generics in TypeScript
2. Introduction to Decorators
3. Modular Programming Best Practices
4. Final Integration: Completing the Chat Application

1. Advanced Generics in TypeScript

What are Generics?

- Generics allow the creation of flexible, reusable components.
- They provide a way to handle data types dynamically while maintaining type safety.

Constrained Generics

- Restrict the type of arguments passed to the generic type.

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

const result = merge({ name: "Alice" }, { age: 30 });
console.log(result); // { name: 'Alice', age: 30 }
```

Default Generic Types

- Provide a default type for a generic parameter.

```
interface Box<T = string> {
    content: T;
}

const stringBox: Box = { content: "Hello" }; // Defaults to string
const numberBox: Box<number> = { content: 42 }; // Explicit type
```

Conditional Types

- Enable dynamic type selection based on conditions.

```
type IsString<T> = T extends string ? "Yes" : "No";

type Result1 = IsString<string>; // 'Yes'
type Result2 = IsString<number>; // 'No'
```

2. Introduction to Decorators

What are Decorators?

- A special syntax to modify or enhance classes, methods, properties, or parameters.
- Often used in frameworks like Angular.

Creating a Class Decorator

```
function Logger(constructor: Function) {
    console.log(`Class ${constructor.name} is created`);
}

@Logger
class Person {
    constructor(public name: string) {}
}

const person = new Person("Alice");
```

Method Decorators

- Modify the behavior of class methods.

```
function Log(target: any, propertyName: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;
    descriptor.value = function (...args: any[]) {
        console.log(`Method ${propertyName} called with arguments:`, args);
        return originalMethod.apply(this, args);
    };
}

class Calculator {
    @Log
    add(a: number, b: number): number {
        return a + b;
    }
}

const calc = new Calculator();
calc.add(2, 3); // Logs method call and arguments
```

3. Modular Programming Best Practices

Principles of Modular Design

1. **Separation of Concerns**
   - Each module handles a specific functionality.
2. **Reusability**
   - Design modules that can be used across applications.
3. **Extensibility**
   - Ensure that modules are easy to extend without modification.

Benefits of Modular Programming

1. **Improved Maintainability**
   - Smaller, well-defined modules are easier to manage.
2. **Enhanced Collaboration**
   - Teams can work on different modules independently.
3. **Scalability**
   - Modules can be replaced or enhanced without affecting the entire system.

4. Final Integration: Completing the Chat Application

Key Features to Finalize

1. **State Management**:
   - Use generics to handle dynamic state shapes.
2. **Logging and Validation**:
   - Use decorators for method logging and input validation.
3. **UI Enhancements**:
   - Dynamically display state information such as active users and message history.

Final Example: State Manager with Generics

```
export class StateManager<T> {
    private state: T;

    constructor(initialState: T) {
        this.state = initialState;
    }

    getState(): T {
        return this.state;
    }

    updateState(newState: Partial<T>): void {
        this.state = { ...this.state, ...newState };
    }
}

const userManager = new StateManager({ users: [] });
userManager.updateState({ users: ["Alice"] });
console.log(userManager.getState());
```

Final Example: Logging Decorator

```
function Log(target: any, propertyName: string, descriptor: PropertyDescriptor): void {
    const originalMethod = descriptor.value;

    descriptor.value = function (...args: any[]) {
        console.log(`Calling ${propertyName} with arguments:`, args);
        const result = originalMethod.apply(this, args);
        console.log(`Result:`, result);
        return result;
    };
}

class WebSocketManager {
    @Log
    sendMessage(message: string): void {
        console.log(`Sending message: ${message}`);
    }
}

const wsManager = new WebSocketManager();
wsManager.sendMessage("Hello World");
```

Summary and Q&A

Key Takeaways

1. Generics provide flexibility and type safety.
2. Decorators simplify repetitive tasks and enhance modularity.
3. Modular programming ensures maintainability and scalability.
4. Final integration ties together all concepts for a complete application.

Questions?