Callback and Overloaded Functions in TypeScript

In TypeScript, functions can have special features like **callback functions** and **function overloading**, making them more powerful and flexible.

Callback Functions

A **callback function** is a function that is passed as an argument to another function and gets executed later.

Why Use Callback Functions?

- Useful when you want a function to execute **only after another function completes**.
- Common in **asynchronous operations** like API calls, file handling, or event handling.

Example: Callback Function in TypeScript

```
// Function that takes a callback as an argument
function greet(name: string, callback: (message: string) => void) {
   console.log(name);
   callback("Hello"); // Calling the callback function
}
// Callback function
function showMessage(message: string) {
   console.log(message);
}
// Calling the function by passing the callback
greet("Pavan", showMessage);
```

Output:

Pavan

Hello

👉 Here, greet calls showMessage after displaying the name.

Function Overloading in TypeScript

Function overloading allows you to define **multiple versions** of a function with the same name but different parameters or return types.

Why Use Function Overloading?

- Helps create functions that work **differently based on input types**.
- Improves **code readability** and reusability.

Basic Rules for Function Overloading

1. **Define Overload Signatures:** List all possible ways the function can be called.
2. **Single Implementation Signature:** There must be **one actual function implementation** that handles all cases.
3. **Ensure Compatibility:** The implementation must be compatible with all overload signatures.

Correct Function Overloading Examples

1. Overloading with Different Parameter Types

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

// Implementation function
function display(value: number | string | boolean): string {
   return `Value is: ${value}`;
}

console.log(display(100));      // ✅ "Value is: 100"
console.log(display("Hello")); // ✅ "Value is: Hello"
console.log(display(true));     // ✅ "Value is: true"
```

✓ **Valid** because the implementation function handles all specified types.

2. Overloading with Different Number of Parameters

```
// Overload signatures
function add(a: number, b: number): number;
function add(a: number, b: number, c: number): number;

// Implementation function
function add(a: number, b: number, c?: number): number {
   return c !== undefined ? a + b + c : a + b;
}

console.log(add(2, 3));      // ✅ 5
console.log(add(2, 3, 4));   // ✅ 9
```

✓ **Valid** because c is optional, ensuring compatibility with both overloads.

3. Overloading with Different Return Types

```
// Overload signatures
function processInput(input: string): string;
function processInput(input: number): number;

// Implementation function
function processInput(input: string | number): string | number {
   return typeof input === "string" ? input.toUpperCase() : input * 2;
}

console.log(processInput("hello"));  // ✅ "HELLO"
console.log(processInput(10));       // ✅ 20
```

✓ **Valid** because the function correctly returns a string for string input and a number for number input.

Incorrect Function Overloading Examples

1. Overloading Without an Implementation Function

```
function test(value: number): string;
function test(value: string): string;
```

❌ **Invalid** because there's no function implementation to handle the overloads.

2. Incorrect Return Type in Implementation

```
function calculate(a: number): number;
function calculate(a: string): string;

// Implementation with incorrect return type
function calculate(a: number | string): boolean {
   return true;  // ❌ Wrong! Should return number or string
}
```

❌ **Invalid** because the function must return **either a number or a string**, not a boolean.

3. Overloading with Identical Parameter Types but Different Return Types

```
function demo(a: number): string;
function demo(a: number): number; // ❌ Cannot overload identical signature

function demo(a: number): string {
   return a.toString();
}
```

❌ **Invalid** because TypeScript doesn't allow overloads with **identical parameter types** but **different return types**.

4. Overloading with Incompatible Parameter Types

```
function fetchData(url: string): string;
function fetchData(url: number): string; // ❌ Incompatible parameter type

// Implementation with incorrect return type
function fetchData(url: string | number): number {
  return 42;  // ❌ Should return a string
}
```

❌ **Invalid** because the implementation function should return a **string** (not a number) to match the overload signatures.

5. Overloading with Optional Parameters That Conflict

```
function add(a: number, b?: number): number;
function add(a: number, b: number, c: number): number;

function add(a: number, b?: number, c?: number): number {
  return a + (b ?? 0) + (c ?? 0);
}
```

❌ **Invalid** because the second overload expects **three parameters**, but the implementation treats b as optional. This can cause unexpected behavior when calling add(1, 2, undefined).

Conclusion

✅ **Callback Functions** allow functions to be executed later, useful in asynchronous programming.

✅ **Function Overloading** enables defining multiple function signatures for different parameter types or return types.