Overview/Review of TypeScript and ES6

Part 1: ES6 and TypeScript

Instructions:

1. Write a small code snippet in ES6 that uses modern features like `let`, `const`, and arrow functions. Hint: Loop through an array of numbers.
2. Convert that snippet into TypeScript by adding explicit type annotations. Run the TypeScript code.

Answer:

```
// ES6 JavaScript Code
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(n => n * 2);
console.log(doubled);

// TypeScript Code with Type Annotations
const numbers: number[] = [1, 2, 3, 4];
const doubled: number[] = numbers.map((n: number): number
=> n * 2);
console.log(doubled);
```

Explanation:

In TypeScript, types (such as `number` and `number[]`) are explicitly declared for variables and function parameters. This helps catch errors during compilation, providing an extra layer of type safety compared to plain ES6 JavaScript.

Part 2: Angular, ES6, and TypeScript

Instructions:

1. Create a simple Angular component using TypeScript.
2. Use ES6 features such as arrow functions and template strings within the component.
3. Ensure that the component displays a title and includes a button that triggers a method.

Answer:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-simple',
  template: `
    <h1>{{ title }}</h1>
    <button (click)="sayHello()">Click Me</button>
  `
})
export class SimpleComponent {
  title: string = `Welcome to Angular!`;

  // Using an arrow function to maintain the correct 'this'
context
  sayHello = (): void => {
    console.log(`Hello from ${this.title}`);
  }
}
```

Part 3: Typing and Classes (ES6 and TypeScript)

Instructions:

1. Create a `User` class with properties for `name` (string) and `age` (number).
2. Include a method `greet` that returns a greeting string.
3. Instantiate the class and log the greeting to the console.

Answer:

```
class User {
  name: string;
  age: number;

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

  greet(): string {
    return `Hello, my name is ${this.name} and I'm $
{this.age} years old.`;
  }
}

const user = new User('Alice', 30);
console.log(user.greet());
```

Part 4: Abstract Classes and Interfaces

Instructions:

1. Create an abstract class `Shape` with an abstract method `area()`.
2. Define an interface `Colored` with a property `color` (string).
3. Implement a concrete class `Rectangle` that extends `Shape` and implements `Colored`.
4. Instantiate `Rectangle` and log its area and color.

Answer:

```
abstract class Shape {
  abstract area(): number;
}

interface Colored {
  color: string;
}

class Rectangle extends Shape implements Colored {
  width: number;
  height: number;
  color: string;

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

const rect = new Rectangle(5, 10, 'blue');
console.log(`Area: ${rect.area()}, Color: ${rect.color}`);
```

Part 5: Interface Patterns

Instructions:

1. Define an interface `Person` with properties `name` (string) and `age` (number).
2. Create another interface `Employee` that extends `Person` and adds an `employeeId` (number).
3. Construct an object that satisfies the `Employee` interface and log its details.

Answer:

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

interface Employee extends Person {
  employeeId: number;
}

const employee: Employee = {
  name: 'Bob',
  age: 25,
  employeeId: 1234
};

console.log(`Employee: ${employee.name}, Age: $
{employee.age}, ID: ${employee.employeeId}`);
```

Part 6: Generics

Instructions:

1. Create a generic function called `getFirstElement` that accepts an array of any type and returns its first element.
2. Ensure that the function works with arrays of numbers, strings, or any other type.
3. Demonstrate the usage with different array types.

Answer:

```
function getFirstElement<T>(arr: T[]): T | undefined {
  return arr.length > 0 ? arr[0] : undefined;
}

console.log(getFirstElement<number>([1, 2, 3]));    //
Output: 1
console.log(getFirstElement<string>(['a', 'b', 'c'])); //
Output: 'a'
```

Part 7: Optional Chaining

Instructions:

1. Define an interface `UserProfile` with a `name` (string) and an optional `address` property that may itself have optional properties like `street` and `city`.
2. Create an object that omits the `street` property.
3. Use optional chaining to safely access the `street` property without causing a runtime error, and log the result.

Answer:

```
interface UserProfile {
  name: string;
  address?: {
    street?: string;
    city?: string;
  };
}
```

```
const user: UserProfile = { name: 'Charlie', address:
{ city: 'New York' } };

// Using optional chaining to safely access the 'street'
property.
const street = user.address?.street;
console.log(street); // Output: undefined (without error)
```

Part 8: Nullish Coalescing

Instructions:

1. Write a function `getGreeting` that accepts a `name` parameter which might be `null` or `undefined`.
2. Use the nullish coalescing operator (`??`) to assign a default value ("Guest") when `name` is `null` or `undefined`.
3. Test the function with both a valid string and a nullish value.

Answer:

```
function getGreeting(name: string | null | undefined):
string {
  // If 'name' is null or undefined, default to 'Guest'
  return `Hello, ${name ?? 'Guest'}!`;
}

console.log(getGreeting(null));    // Output: Hello, Guest!
console.log(getGreeting('Diana')); // Output: Hello, Diana!
```

Part 9: Functional vs Procedural JavaScript

Instructions:

1. Write a function called `sumProcedural` that sums an array of numbers using a procedural approach (using a for-loop).
2. Refactor the function into a functional style using the array's `reduce` method in a function called `sumFunctional`.
3. Compare the outputs of both functions using the same array.

Answer:

```
// Procedural approach
function sumProcedural(numbers: number[]): number {
  let total = 0;
  for (let i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }
  return total;
}

// Functional approach
function sumFunctional(numbers: number[]): number {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}

const nums = [1, 2, 3, 4];
console.log(sumProcedural(nums)); // Output: 10
console.log(sumFunctional(nums)); // Output: 10
```