Exercise 1

Generics

Expected Time: 15 min

Scenario: You're developing a complex application that involves a variety of data manipulation tasks. One of the tasks is to create a function that takes an array of values and a callback function. The function should iterate through the array and apply the callback function to each element. The result of the callback function should replace the original element in the array. However, you want to ensure that the data types of the elements and the return values of the callback function are aligned. Testo

Problem: Write a TypeScript function called `transformArray` that uses advanced generics to take an array of values and a callback function. The callback function should have a type parameter that matches the input array's type, and it should return a value of the same type. The `transformArray` function should iterate through the input array, apply the callback function to each element, and replace the original element with the result. Test the function with various data types and callback functions.

Here's a starting point for your code:

```
// Example usage
const numberArray = [1, 2, 3, 4, 5];
const doubledArray = transformArray(numberArray, (num) => num * 2);
console.log(doubledArray); // Output: [2, 4, 6, 8, 10]

const stringArray = ['apple', 'banana', 'cherry'];
const uppercasedArray = transformArray(stringArray, (str) => str.toUpperCase());
console.log(uppercasedArray); // Output: ['APPLE', 'BANANA', 'CHERRY']
```

Union and Intersection Types

Expected Time: 15 min

Scenario: You're developing a complex e-commerce platform that supports a wide range of products. Each product can have multiple variations, such as different sizes,

colors, and materials. You need to create a flexible type system that can accurately represent these variations while maintaining type safety.

Problem: Define TypeScript types for products and their variations. Create a type for the base product, then use advanced union and intersection types to accurately represent variations based on size, color, and material. Each variation should include properties specific to that variation type, while the product should have common properties like `name`, `price`, and `description`.

Here's a starting point for your code:

```
const product: Product = {
    name: 'T-Shirt',
    price: 29.99,
    description: 'A comfortable and stylish T-shirt.',
};

const sizeVariation: Product = {
    name: 'T-Shirt',
    price: 34.99,
    description: 'A comfortable and stylish T-shirt.',
    variationType: 'size',
    size: 'L',
};

const colorVariation: Product = {
    name: 'T-Shirt',
    price: 39.99,
    description: 'A comfortable and stylish T-shirt.',
    variationType: 'color',
    color: 'blue',
};

const materialVariation: Product = {
    name: 'T-Shirt',
    price: 44.99,
    description: 'A comfortable and stylish T-shirt.',
    variationType: 'material',
    material: 'cotton',
};
```

Type Guards

Expected Time: 15 min

Problem: You have a function `extractCategoryName` that can either return a string or `null` depending on the input type. Write the type guards `isProduct` and `isString`.
Here's a starting point for your code:

```
interface Category {
  name: string;
}

interface Product {
  category: Category;
}

function processData(input: Category | string | null): string | null {
  if (isCategory(input)) {
    return input.name;
  } else if (isString(input)) {
    return input;
  }
  return null;
}
```