4(a) TypeScript Program Using Namespaces to Organize Code Logically

Aim:
To write a TypeScript program that demonstrates the use of namespaces for organizing related code elements such as variables, functions, and classes in a logical manner.

Description:
Namespaces in TypeScript are used to group related code and avoid name conflicts in large projects. This program defines a namespace containing functions and classes related to mathematical operations and shows how to access them using the namespace name.

Program:

```
// Namespace Example in TypeScript

namespace MathOperations {

  export function add(a: number, b: number): number {

    return a + b;

  }

  export function subtract(a: number, b: number): number {

    return a - b;

  }

  export class Calculator {

    multiply(a: number, b: number): number {

      return a * b;

    }

    divide(a: number, b: number): number {

      return a / b;

    }

  }

}

// Using the namespace

const calc = new MathOperations.Calculator();

console.log("Addition:", MathOperations.add(10, 5));

console.log("Subtraction:", MathOperations.subtract(10, 5));

console.log("Multiplication:", calc.multiply(10, 5));

console.log("Division:", calc.divide(10, 5));
```

Output:
(In Terminal)

```
Addition: 15

Subtraction: 5

Multiplication: 50

Division: 2
```

4(b) TypeScript Program to Demonstrate Generics with Constraints for Type-Safe Functions

Aim:
To write a TypeScript program that demonstrates the use of generics with constraints to create type-safe and reusable functions.

Description:
Generics in TypeScript allow creating reusable components that work with multiple types. By adding constraints using the extends keyword, we can restrict the types that can be passed. This ensures type safety while maintaining flexibility.

Program:

```
// Generics with Constraints Example in TypeScript

// Generic function with constraint

function getProperty<T extends object, K extends keyof T>(obj: T, key: K): T[K] {

  return obj[key];

}

// Generic interface example

interface Lengthwise {

  length: number;

}

function logLength<T extends Lengthwise>(item: T): void {

  console.log("Length is:", item.length);

}

// Using the functions

const person = { name: "Vivek", age: 21 };

console.log("Name:", getProperty(person, "name"));

console.log("Age:", getProperty(person, "age"));

logLength("Hello World");

logLength([10, 20, 30]);
```

Output:
(In Terminal)

```
Name: Vivek

Age: 21

Lengthis: 11

Lengthis: 3
```