**Experiment 2**

**Shashwat Tripathi  
D15A Batch C  
RollNo: 64**

**AIM : To study basic constructs in Typescript.**

**THEORY:**

**1. What are the different data types in TypeScript? What are Type Annotations in Typescript?**

1. Basic Data Types:

a. Number: Represents numeric values, including both integers and floating-point numbers.

b. String: Represents sequences of characters, commonly used for text.

c. Boolean: Represents logical values - true or false.

d. Array: Represents ordered lists of values.

e. Tuple: Represents ordered arrays with a fixed number of elements, where each element can have a different data type.

f. Enum: Provides a way to define a set of named constant values.

g. Any: Represents a variable that can have any data type. It is used when the type is not known during development.

h. Void: Typically used as the return type of functions that do not return a value.

2. User-Defined Types:

a. Classes: A blueprint for creating objects with properties and methods.

b. Interfaces: Defines contracts for object shapes, specifying the structure that objects should adhere to.

Type Annotations:

Type Annotations involve explicitly specifying the data type of a variable, parameter, or return type in TypeScript. It is done using a colon (:) followed by the type.

For example:

const num1: number;

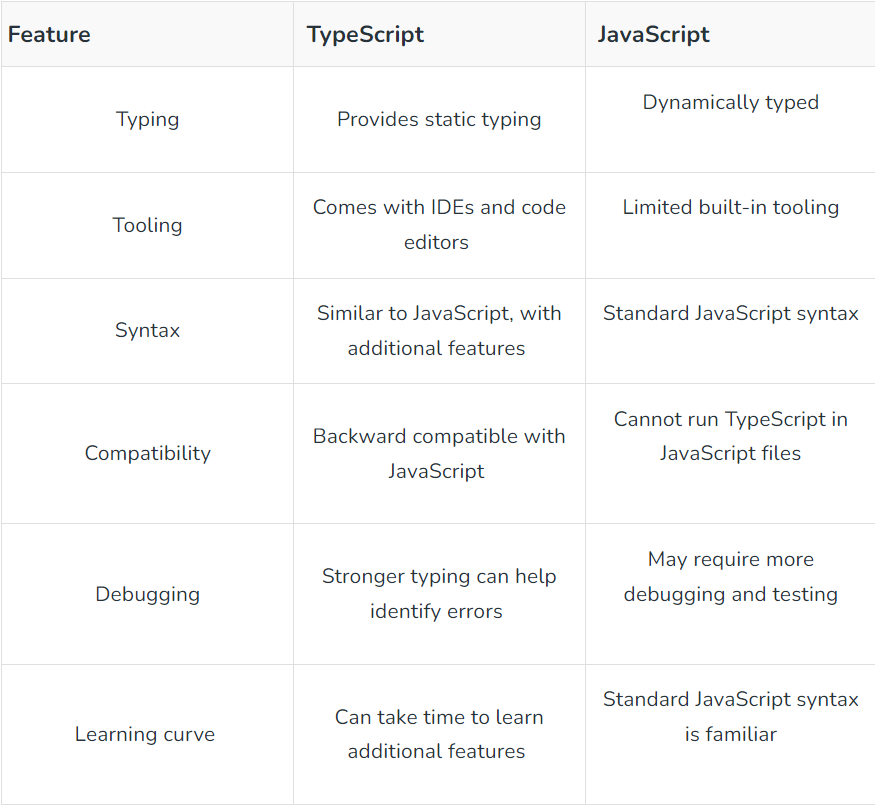
**2. How do you compile TypeScript files?**

To compile TypeScript files, you use the TypeScript Compiler, tsc. TypeScript code is written in .ts files, and the TypeScript Compiler translates these files into JavaScript code (.js files) that can be executed by a JavaScript runtime.

tsc shashwat.ts

node shashwat.js

**3. What is the difference between JavaScript and TypeScript?**

****

**4. Compare how Javascript and Typescript implement Inheritance.**

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and methods from another class. Both JavaScript and TypeScript support inheritance, but the implementation differs due to the dynamic and prototypal nature of JavaScript and the static typing features introduced by TypeScript.

Javascript:

JavaScript primarily uses prototypal inheritance, where objects inherit from other objects. n JavaScript, inheritance is achieved through prototype chaining. The child class prototype is set to an instance of the parent class, creating a chain of prototypes. The call function is used to invoke the parent class constructor within the child class constructor.

Typescript:

TypeScript supports classical inheritance and adds features like access modifiers and type annotations. In TypeScript, the class syntax is used to define classes, and the extends keyword is used for inheritance. Access modifiers like private, protected, and public can be used to control the visibility of class members. The super keyword is used to call the parent class constructor.

**5. What is the difference between Classes and Interfaces in Typescript? Where are interfaces used.**

Classes are used for creating objects with properties and methods, while interfaces are used to define contracts for object shapes. Interfaces play a crucial role in promoting code organization, loose coupling, and defining the structure of objects in TypeScript. They are particularly useful when dealing with multiple implementations that need to adhere to a common structure.

Purpose of Classes:

Definition: Classes in TypeScript are used to create objects with properties and methods.

Instantiation: You can create instances (objects) of a class using the new keyword.

Purpose of Interfaces:

Definition: Interfaces in TypeScript are used to define contracts for object shapes, specifying the structure that objects should adhere to.

Declaration: Interfaces do not provide an implementation; they declare the expected structure.

**6. How generics make the code flexible and why should we use generics over other types.**

**Flexibility:** Generics allow code to operate on various types, making it adaptable and versatile.

**Dynamic Typing:** They provide dynamic typing with type safety determined at runtime.

**Function and Class Reusability:** Code written with generics can be reused with different types, reducing duplication.

**Increased Abstraction:** Generics enable abstract and generalized operations, enhancing code abstraction.

**Improved Code Readability:** They express code intent without being tied to specific types, improving readability.

**Collections and Data Structures:** Generics are commonly used for creating reusable structures that hold elements of different types.

**Code Maintainability:** Code with generics is more maintainable, requiring fewer modifications for new types or changes.

**Type Inference:** TypeScript often allows automatic type inference with generics, reducing the need for explicit annotations.

**Library and Framework Design:** Generics are crucial in designing libraries and frameworks, providing extensibility points for diverse scenarios.

**Problem Statement:**

**1. Implement a simple Calculator to demonstrate the usage of different data types ( include any, never)**

import \* as readlineSync from "readline-sync";

function add(num1: number, num2: number): number {

return num1 + num2;

}

function subtract(num1: number, num2: number): number {

return num1 - num2;

}

function multiply(num1: number, num2: number): number {

return num1 \* num2;

}

function divide(num1: number, num2: number): number {

if (num2 === 0) {

console.log("ERROR! Cannot divide by zero");

// process.exit(1);

return -1;

}

return num1 / num2;

}

function main() {

const num1: number = parseFloat(

readlineSync.question("Enter the number 1: ")

);

const num2: number = parseFloat(

readlineSync.question("Enter the number 2: ")

);

console.log(`

----------------Menu-------------------

1. Add

2. Subtract

3. Multiply

4. Divide

---------------------------------------

`);

let choice: any;

do {

choice = parseFloat(readlineSync.question("Enter your choice: "));

let result: number;

switch (choice) {

case 1: {

result = add(num1, num2);

break;

}

case 2: {

result = subtract(num1, num2);

break;

}

case 3: {

result = multiply(num1, num2);

break;

}

case 4: {

result = divide(num1, num2);

break;

}

default: {

console.log("Exiting...");

// process.exit(1);

result = -1;

break;

}

}

console.log(`Result: ${result}`);

} while (choice < 5);

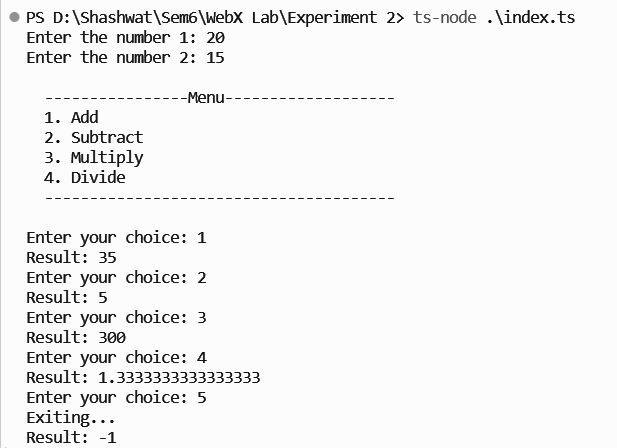
}

main();

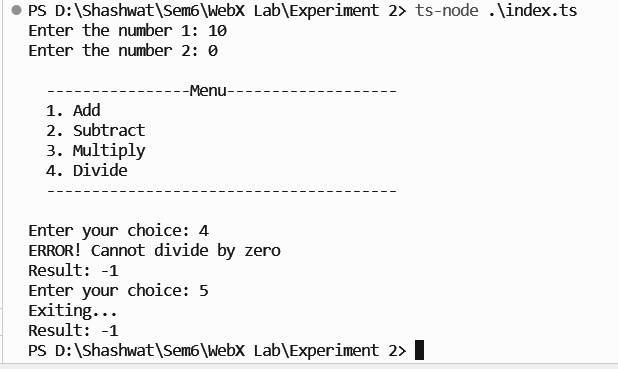
The basic functions of +, -, \*, / written in different function bodies (with divide by 0 handled).

Taking input using readline-sync library/ module and providing a menu-card for user to perform different operations.

**Output:**

****

**All the operations performed.**

****

**Division by 0 handled.**

**2. Develop a TypeScript application for inheritance hierarchy. Create classes User, Admin (inherits from User), and DataManager. Implement access specifiers (public, private, protected) to regulate data visibility.**

class User {

private username: string;

protected email: string;

constructor(username: string, email: string) {

this.username = username;

this.email = email;

}

public getUserInfo(): string {

return `Username: "${this.username}" & Email "${this.email}"`;

}

}

class Admin extends User {

private adminName: string;

protected adminEmail: string;

constructor(

username: string,

email: string,

adminName: string,

adminEmail: string

) {

super(username, email);

this.adminEmail = adminEmail;

this.adminName = adminName;

}

public grantAccess(user: User): any {

console.log(

`Admin "${this.adminName}" granted access to "${user.getUserInfo()}"`

);

}

}

class DataManager {

private data: string;

constructor(data: string) {

this.data = data;

}

private getData(admin: Admin): any {

return `The data: [${

this.data

}] was accessed by Admin "${admin.getUserInfo()}".`;

}

public getDataForAdmin(admin: Admin): any {

return this.getData(admin);

}

}

// main driver code:

const regularUser = new User("regular", "regularUser@gmail.com");

console.log(`Regular User Information: ${regularUser.getUserInfo()}`);

console.log("\n");

const adminUser = new Admin(

"adminUser",

"adminUser@gmail.com",

"adminABC",

"adminABC@gmail.com"

);

console.log(`Admin User Information: ${adminUser.getUserInfo()}`);

console.log("\n");

console.log("Granting the access to regular User...");

adminUser.grantAccess(regularUser);

const dataManager = new DataManager(

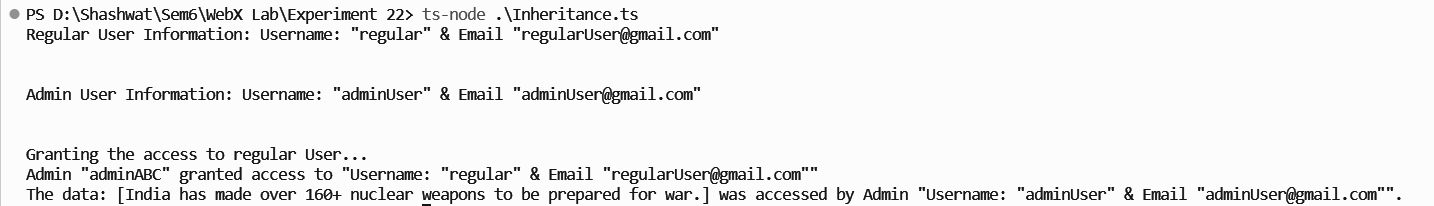
"India has made over 160+ nuclear weapons to be prepared for war."

);

console.log(dataManager.getDataForAdmin(adminUser));

* The User class has private username and protected email properties, and a public getUserInfo method.
* The Admin class inherits from User and has private adminName and adminEmail properties. It also has a public grantAccess method to demonstrate accessing user information.
* The DataManager class has private data property and a protected getData method. The method can only be accessed by an Admin, demonstrating the use of protected access.

**Output:**

****

**3. Create a TypeScript program for working with geometric shapes.**

// creating Shape interface

interface Shape<T> {

calculateArea(): T;

calculatePerimeter(): T;

}

// Circle class implementing the Shape interface

class Circle implements Shape<number> {

private radius: number;

constructor(radius: number) {

this.radius = radius;

}

calculateArea(): number {

return Math.PI \* this.radius \* this.radius;

}

calculatePerimeter(): number {

return 2 \* Math.PI \* this.radius;

}

}

// Rectangle

class Rectangle implements Shape<number> {

private length: number;

private breadth: number;

constructor(length: number, breadth: number) {

this.length = length;

this.breadth = breadth;

}

calculateArea(): number {

return this.length \* this.breadth;

}

calculatePerimeter(): number {

return 2 \* (this.length + this.breadth);

}

}

// Triangle

class Triangle implements Shape<number> {

private a: number;

private b: number;

private c: number;

constructor(a: number, b: number, c: number) {

this.a = a;

this.b = b;

this.c = c;

}

calculateArea(): number {

const s = (this.a + this.b + this.c) / 2;

return Math.sqrt(s \* (s - this.a) \* (s - this.b) \* (s - this.c));

}

calculatePerimeter(): number {

return this.a + this.b + this.c;

}

}

// main driver code

const circle = new Circle(10);

console.log(`Circle:

Area: ${circle.calculateArea()}

Perimeter: ${circle.calculatePerimeter()}

`);

const rectangle = new Rectangle(7, 5);

console.log(`Rectangle:

Area: ${rectangle.calculateArea()}

Perimeter: ${rectangle.calculatePerimeter()}

`);

const triangle = new Triangle(5, 12, 13);

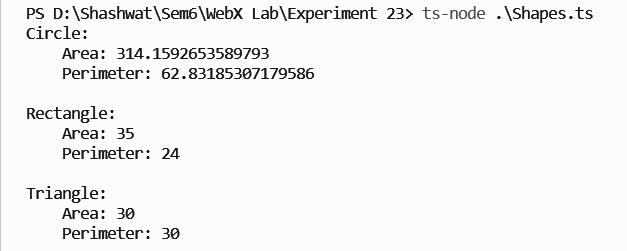
console.log(`Triangle:

Area: ${triangle.calculateArea()}

Perimeter: ${triangle.calculatePerimeter()}

`);

**Output:**



**CONCLUSION:** Thus, we studied the various new concepts introduced in TypeScript and implemented them in our code.