Experiment – 1 a: TypeScript

Name of Student	Aryan Dangat
Class Roll No	12
D.O.P.	
D.O.S.	
Sign and Grade	

Experiment – 1 a: TypeScript

1. **Aim:** Write a simple TypeScript program using basic data types (number, string, boolean) and operators.
2. **Problem Statement:**

   a. Create a calculator in TypeScript that uses basic operations like addition, subtraction, multiplication, and division. It also gracefully handles invalid operations and division by zero..

   b. Design a Student Result database management system using TypeScript.

Theory:

a) What are the different data types in TypeScript? What are Type Annotations in Typescript?

Different Data Types in TypeScript

TypeScript supports several built-in data types that are essential for declaring variables, function parameters, and object properties. These data types help ensure that the code is robust and maintainable by catching type-related errors at compile time. Here are the primary data types in TypeScript:

Data Type	Keyword	Description
Number	`number`	Represents both integers and floating-point numbers. It can also handle binary, octal, decimal, and hexadecimal literals.
String	`string`	Used for textual data. Strings can be enclosed in single quotes, double quotes, or backticks (template literals).
Boolean	`boolean`	Represents logical values: `true` or `false`.
Void	`void`	Typically used for function return types to indicate that a function does not return any value.
Null	`null`	Represents the intentional absence of any object value.
Undefined	`undefined`	Denotes the value given to uninitialized variables.
Any	`any`	Allows a variable to hold any type of value.
Symbol	`symbol`	A unique and immutable primitive introduced in ES2015.
Object	`object`	Represents instances of user-defined classes, arrays, functions, etc..
Never	`never`	Represents values that never occur.

Type Annotations in TypeScript

Type annotations in TypeScript are used to explicitly specify the type of variables, function parameters, and return values. This helps the compiler check for type errors at compile time, ensuring that the code behaves as expected.

Syntax and Usage

Type annotations are added using the syntax : type after the identifier (variable, function parameter, etc.). For example:

```typescript
Typescript
let age: number = 25; // Variable annotation
let name: string = "John"; // Variable annotation
let active: boolean = true; // Variable annotation

function greet(name: string): string { // Function parameter and return type annotation
    return `Hello, ${name}!`;
}
```

Benefits

1. Error Prevention: Type annotations help prevent type-related errors by ensuring that variables and function parameters are used with the correct data types.
2. Code Clarity: They improve code readability by clearly indicating the expected types of variables and function parameters.
3. IDE Support: Many IDEs provide better code completion and inspection when type annotations are used, enhancing the development experience.

Type annotations are crucial in TypeScript for maintaining robust and maintainable codebases.

b) How do you compile TypeScript files?

- Install TypeScript Compiler
  npm install -g typescript

- Create a Typescript file
  ```typescript
  let message: string = "Hello, TypeScript!";
  console.log(message);
  ```

- Compile TypeScript to JavaScript
  tsc example.ts

- Run on compiled JavaScript File
  Node example.js

c) What is the difference between JavaScript and TypeScript?

1. Typing System
   JavaScript is dynamically typed, meaning variables can hold any type, and type checking happens at runtime. This flexibility can sometimes lead to unexpected errors. TypeScript introduces static typing, requiring variables to have defined types, allowing errors to be caught during compilation rather than execution.
2. Compilation
   JavaScript runs directly in browsers or Node.js without the need for compilation. In contrast, TypeScript must be compiled into JavaScript using the TypeScript compiler (tsc), adding an extra step but improving code reliability and maintainability.
3. Error Handling
   JavaScript allows errors to appear only at runtime, which can make debugging difficult. TypeScript detects errors at compile time, helping developers catch potential issues early and reducing the risk of runtime failures.
4. Feature Support
   JavaScript follows ECMAScript standards, but developers must wait for browser support to use new features. TypeScript provides early access to upcoming JavaScript features and also adds additional functionalities like interfaces, generics, and decorators.
5. Object-Oriented Programming (OOP)
   JavaScript supports OOP with ES6 classes but lacks features like interfaces and strong encapsulation. TypeScript enhances OOP with interfaces, access modifiers (private, protected, public), and generics, making it more structured for large-scale applications.
6. Use Case
   JavaScript is best suited for small-scale projects, quick scripting, and front-end development. TypeScript is ideal for large applications, enterprise software, and projects where code maintainability and scalability are important.

d) Compare how JavaScript and Typescript implement Inheritance.

1. Class-Based Inheritance
   Both JavaScript and TypeScript use ES6 classes for inheritance. JavaScript allows class-based inheritance using the class keyword, with child classes extending parent classes using extends. TypeScript follows the same approach but adds static typing for better type safety.

JavaScript Example:

```javascript
class Animal {
   constructor(name) {
      this.name = name;
   }

   makeSound() {
      console.log("Some generic sound");
   }
}

class Dog extends Animal {
   makeSound() {
      console.log("Bark!");
   }
}

const dog = new Dog("Buddy");
dog.makeSound(); // Output: Bark!
```

TypeScript Example:

```typescript
class Animal {
   name: string;

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

   makeSound(): void {
      console.log("Some generic sound");
   }
}

class Dog extends Animal {
   makeSound(): void {
      console.log("Bark!");
   }
}

const dog = new Dog("Buddy");
dog.makeSound(); // Output: Bark!
```

Key Difference: TypeScript enforces type safety (string, void), while JavaScript does not.

2. Access Modifiers
   JavaScript does not support private, protected, or public access modifiers for
   encapsulation. TypeScript introduces these, allowing better control over
   properties and methods.

   TypeScript Example (with access modifiers):

```
class Animal {
    protected name: string;  // Accessible in subclasses but not outside

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

    public makeSound(): void {
       console.log("Some generic sound");
    }
}

class Dog extends Animal {
    constructor(name: string) {
       super(name);
    }

    public makeSound(): void {
       console.log(`${this.name} says Bark!`);
    }
}

const dog = new Dog("Buddy");
dog.makeSound(); // Output: Buddy says Bark!
```

   Key Difference: TypeScript allows private, protected, and public modifiers,
   improving data security and access control.

3. Interfaces and Abstract Classes
   JavaScript does not support interfaces or abstract classes, while TypeScript
   does. This makes TypeScript more suitable for enforcing strict inheritance
   structures.

   TypeScript Example (using an abstract class):

```
abstract class Animal {
    abstract makeSound(): void;  // Must be implemented by subclasses
}
```

```
class Dog extends Animal {
   makeSound(): void {
      console.log("Bark!");
   }
}

const dog = new Dog();
dog.makeSound(); // Output: Bark!
```

Key Difference: TypeScript allows abstract classes, ensuring child classes implement specific methods, while JavaScript does not.

e) How generics make the code flexible and why we should use generics over other types. In the lab assignment 3, why the usage of generics is more suitable than using any data type to handle the input.

Generics allow code to work with multiple types while maintaining type safety. Unlike any, which removes type restrictions, generics adapt to different types dynamically while ensuring correctness at compile time. This makes the code more reusable, flexible, and less error-prone**.**
In Lab Assignment 3**,** generics are preferred because they allow handling various data types without losing type safety. Using any would permit unintended types, leading to potential errors. Generics ensure controlled flexibility, making the program more reliable and maintainable.

f) What is the difference between Classes and Interfaces in Typescript? Where are interfaces used?

1) **Definition and Purpose**

- **Class:** A blueprint for creating objects, defining properties and methods that determine an object's behavior. Classes can have constructors and support inheritance.
- **Interface:** A contract that defines the structure of an object but does not provide implementations. It only specifies what properties and methods an object should have.

2) **Implementation**

- **Class:** Can have implementations for methods and constructors. It defines how an object behaves.
- **Interface:** Only declares method signatures and properties but does not provide implementations. A class must implement an interface to ensure it follows the specified structure.

3) **Inheritance and Extension**

- **Class:** Supports inheritance using the extends keyword. A class can inherit properties and methods from another class.
- **Interface:** Uses extends to inherit properties from multiple interfaces but cannot extend classes. A class can implement multiple interfaces using the implements keyword.

4) **Object Creation**

- **Class:** Can be instantiated using the new keyword.
- **Interface:** Cannot be instantiated directly. It only defines the expected structure of an object

3. **Output:**

```
function calculator(a: number, b: number, operator: string): number | never {

 switch (operator) {

   case "+":

   return a + b;

   case "-":

   return a - b;

   case "*":
```

```javascript
      return a * b;

    case "/":

      if (b === 0) {

        throw new Error("Division by zero is not allowed!");

      }

      return a / b;

    default:

      throw new Error(`Invalid operator: '${operator}'. Use +, -, *, or /.`);

  }

}


// Example Usage

try {

  console.log(calculator(10, 2, "+")); // Output: 12

  console.log(calculator(10, 2, "-"));

  console.log(calculator(10, 2, "*"));

  console.log(calculator(10, 2, "/"));

  console.log(calculator(10, 0, "/"));

  console.log(calculator(10, 2, "%"));

} catch (error) {

  if (error instanceof Error) {

    console.error(error.message);

  } else {
```
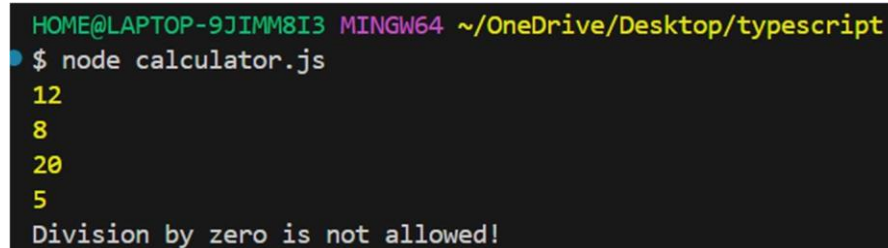
```
        console.error("An unknown error occurred.");

      }

    }
```

```
HOME@LAPTOP-9JIMM8I3 MINGW64 ~/OneDrive/Desktop/typescript
$ node calculator.js
12
8
20
5
Division by zero is not allowed!
```

```typescript
class Student {
  private id: number;
  private name: string;

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

  public getId(): number {
     return this.id;
  }

  public getName(): string {
     return this.name;
  }
}

class Result {
  private studentId: number;
  private subject: string;
  private score: number;

  constructor(studentId: number, subject: string, score: number) {
     this.studentId = studentId;
```

```typescript
        this.subject = subject;
        this.score = score;
    }

    public getStudentId(): number {
        return this.studentId;
    }

    public getSubject(): string {
        return this.subject;
    }

    public getScore(): number {
        return this.score;
    }
}

class Database {
    private students: Student[] = [];
    private results: Result[] = [];

    public addStudent(student: Student): void {
        this.students.push(student);
    }

    public addResult(result: Result): void {
        this.results.push(result);
    }

    public getStudentResults(studentId: number): Result[] {
        return this.results.filter(result => result.getStudentId() === studentId);
    }

    public getAllStudents(): Student[] {
        return this.students;
    }

    public getAllResults(): Result[] {
        return this.results;
    }
```

```javascript
}

// Create a database instance
const db = new Database();

// Create students
const student1 = new Student(1, "Spandan Deb");
const student2 = new Student(2, "Ram Palan");

// Add students to the database
db.addStudent(student1);
db.addStudent(student2);

// Create results for students
const result1 = new Result(1, "Math", 90);
const result2 = new Result(1, "Science", 85);
const result3 = new Result(2, "Math", 95);
const result4 = new Result(2, "Science", 80);

// Add results to the database
db.addResult(result1);
db.addResult(result2);
db.addResult(result3);
db.addResult(result4);

// Retrieve and display results for a student
const student1Results = db.getStudentResults(1);
console.log(`Results for ${student1.getName()}:`);
student1Results.forEach(result => {
   console.log(`Subject: ${result.getSubject()}, Score: ${result.getScore()}`);
});

// Retrieve and display all students and their results
console.log("\nAll Students and Their Results:");
db.getAllStudents().forEach(student => {
   const results = db.getStudentResults(student.getId());
   console.log(`Student: ${student.getName()}`);
   results.forEach(result => {
      console.log(`Subject: ${result.getSubject()}, Score: ${result.getScore()}`);
   });
```

```
  });
```

```
PS C:\Users\Archit\Downloads\imp> tsc student.ts
PS C:\Users\Archit\Downloads\imp> node student.js
Results for Aryan Dangat:
Subject: Math, Score: 90
Subject: Science, Score: 85

All Students and Their Results:
Student: Aryan Dangat
  Subject: Math, Score: 90
  Subject: Science, Score: 85
Student: Ram Palan
  Subject: Math, Score: 95
  Subject: Science, Score: 80
```