Experiment – 1 a: TypeScript

Name of Student	**Gunjan Chandnani**
Class Roll No	**06**
D.O.P.	**21/01/2025**
D.O.S.	**28/01/2025**
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.

```
// Step 1: Declare basic data types
const studentName: string = "John Doe";
const subject1: number = 45;
const subject2: number = 38;
const subject3: number = 50;

// Step 2: Calculate the average marks
const totalMarks: number = subject1 + subject2 + subject3;
const averageMarks: number = totalMarks / 3;

// Step 3: Determine if the student has passed or failed
const isPassed: boolean = averageMarks >= 40;

// Step 4: Display the result
console.log(Student Name: ${studentName});
console.log(Average Marks: ${averageMarks});
```

console.log(Result: ${isPassed ? "Passed" : "Failed"});

3.Theory:

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

TypeScript provides static typing through various data types. Below are the main data types available:

1. Primitive Types

- The `string` type represents textual data. Example: `let name: string = "Alice";`.
- The `number` type is used for integers and floating-point numbers.

  Example: `let age: number = 25;`.

- The `boolean` type represents true or false values.

  Example: `let isAdmin: boolean = true;`.

- The `null` type represents an intentional absence of any object value.

  Example: `let empty: null = null;`.

- The `undefined` type represents a variable that has been declared but not yet assigned a value.

  Example: `let notAssigned: undefined = undefined;`.

2. Special Types

- The `any` type can hold any type of value and disables type checking.

  Example: `let anything: any = 5; anything = "Hello";`.

- The `unknown` type is similar to `any` but is safer.

Example: `let unknownValue: unknown = "test";`.

- The `void` type is used for functions that do not return a value.

  Example: `function log(): void { console.log("Logging..."); }`.

- The `never` type represents a value that never occurs, such as a function that always throws an error.

  Example: `function throwError(): never { throw new Error("Error occurred"); }`.

3. Object Type

- The `object` type represents non-primitive types. Example: `let person: object = { name: "John", age: 30 };`.

4. Array Types

- Using square brackets syntax.

  Example: `let numbers: number[] = [1, 2, 3];`.

- Using generic Array syntax.

  Example: `let strings: Array<string> = ["a", "b", "c"];`.

5. Tuple

- A tuple represents a fixed-length array with known types.

  Example: `let tuple: [string, number] = ["Alice", 25];`.

6. Enum

- Enums define a set of named constants.

  Example:
  `enum Direction { Up, Down, Left, Right } let dir: Direction = Direction.Up;`.

7. Union Types

- Union types allow a variable to hold multiple types.

  Example: `let id: string | number = 123; id = "ABC";`.

8. Literal Types

- Literal types restrict a variable to specific values.

  Example: `let status: "success" | "error" = "success";`.

Type Annotations in TypeScript

Type Annotations are used to explicitly specify the type of variables, function parameters, or return types. This helps enforce type safety and makes the code more predictable.

Syntax

The basic syntax is: `let variableName: type = value;`.

Examples

- Variable annotation: `let count: number = 10;`.
- Function parameter annotation: `function greet(name: string): void { console.log("Hello " + name); }`.
- Function return type annotation: `function add(x: number, y: number): number { return x + y; }`.

Benefits of Type Annotations

- Improves code readability.
- Prevents type-related bugs.
- Enhances editor support (e.g., autocompletion and type hints).

a. How do you compile TypeScript files?
b. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Type System	Dynamically typed	Statically typed
Compilation	Interpreted	Compiled to JavaScript
Advanced Features	Limited	Supports interfaces, enums, generics
Error Detection	Runtime	Compile-time
Tooling	Standard	Advanced with better IDE support
Usage	Directly in browsers, Node.js	Needs transpilation to JavaScript
Learning Curve	Easier for beginners	Slightly steeper due to static typing

c. Compare how Javascript and Typescript implement Inheritance.

1. JavaScript Inheritance

JavaScript implements inheritance using **prototypes** or **ES6 classes**. Modern JavaScript (ES6+) allows class-based inheritance that looks similar to traditional OOP languages.

Example in JavaScript:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}

let dog = new Dog("Buddy");
dog.speak(); // Buddy barks.
```

3. 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 in TypeScript allow us to create **reusable and type-safe components** that can work with a variety of data types instead of being restricted to a single one. Generics are like placeholders for types, which are specified when the function, class, or interface is used.

   Why Generics are Useful:

d. Type Safety: Generics ensure that the types remain consistent throughout the code, preventing accidental type errors.

Example:

```
function identity<T>(value: T): T {
  return value;
}

let num = identity<number>(5);  // Type-safe: num is guaranteed to be a number
```

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

How Generics Make Code Flexible

Generics in TypeScript allow developers to create **reusable, flexible, and type-safe components**. Instead of hardcoding specific types, generics act as placeholders for types, which can be provided when the component is used.

Why Generics are Useful

1. **Type Safety**: Generics preserve type information and prevent type mismatches. For example, you can define a function as `function identity<T>(value: T): T`, where `T` is a generic type. When you pass a number like `identity<number>(5)`, TypeScript will ensure that `5` is treated as a number throughout the function.

2. **Flexibility**: Instead of writing separate code for different types like string, number, or object, you can write one generic version. For example, `function identity<T>(value: T): T` can now accept and return any type specified at the time of calling the function.

3. **Reusability**: You avoid repetitive code. For instance, instead of writing multiple functions like `function printString(value: string)` or `function printNumber(value: number)`, you can write one generic function like `function print<T>(value: T)` to handle both.

4. **Better IntelliSense and Autocompletion**: Since generics retain type information, the IDE provides better auto-suggestions and detects possible type errors early.

Why Use Generics Over `any`

Using `any` disables TypeScript's type checking. For example, a function defined as `function identity(value: any): any` allows any type of input and returns any type, which means TypeScript will not alert you if you accidentally treat a string as a number.

In contrast, using generics like `function identity<T>(value: T): T` ensures that TypeScript knows what type is being passed and returned. So if you pass a string, TypeScript will expect string operations; if you pass a number, it will expect number operations. This provides **stronger type safety**.

Why Generics are Suitable for Lab Assignment 3

In **Lab Assignment 3**, you likely need to write a function or a class that handles **inputs of various types** (e.g., string, number, object).

If you were to use `any`, such as `function handleInput(value: any): any`, TypeScript wouldn't enforce any type rules. This means you could easily introduce

errors by passing an unexpected type or by performing operations that don't make sense for that type.

However, using a generic function like `function handleInput<T>(value: T): void` allows the function to accept different types while still enforcing type consistency. For example, when calling `handleInput<string>("Hello")`, TypeScript will ensure that `value` is treated as a string throughout the function.

Specific Benefits for Lab 3

1. **Multiple Data Types Handling**: Since Lab 3 deals with various inputs, generics allow a single implementation that can handle all these types safely.

2. **Reduced Errors**: TypeScript will catch type mismatches during compilation, preventing bugs before runtime.

3. **Code Reusability & Clean Structure**: Instead of duplicating the logic for each type, generics make the code clean and reusable across different data types without sacrificing type safety.

Practical Example for Lab 3

For instance, if you have a generic function defined as `function processInput<T>(input: T): void`, you can now handle `processInput<string>("Hello")`, `processInput<number>(123)`, and even `processInput<{name: string}>({name: "Alice"})` all with the same function, but each with proper type checking.

Conclusion

- **Generics** provide flexibility and type safety, allowing a single piece of code to handle multiple types without type errors.
- In **Lab Assignment 3**, generics are a superior choice compared to `any` because they prevent type-related bugs and make your code **more robust, maintainable, and scalable**.

4. **Output:**
    a) **calculator:**

```
// Define valid operations including string concatenation

type Operation = 'add' | 'subtract' | 'multiply' | 'divide' | 'concatenate';



function calculate(operation: Operation, value1: number | string, value2: number | string): number | string | never {

  if (typeof value1 !== typeof value2) {

      throw new Error('Both values must be of the same type (either numbers or strings).');

  }

  switch (operation) {

    case 'add':

      return ensureNumber(value1) + ensureNumber(value2);

    case 'subtract':

      return ensureNumber(value1) - ensureNumber(value2);

    case 'multiply':

      return ensureNumber(value1) * ensureNumber(value2);

    case 'divide':

      if (ensureNumber(value2) === 0) {

        throw new Error('Error: Division by zero is not allowed.');

      }

      return ensureNumber(value1) / ensureNumber(value2);

    case 'concatenate':
```

```
            return ensureString(value1) + ensureString(value2);
        default:
            return handleInvalidOperation(operation);
    }
}

function ensureNumber(value: any): number {
    if (typeof value !== 'number') {
        throw new Error('Invalid input: Expected a number.');
    }
    return value;
}

function ensureString(value: any): string {
    if (typeof value !== 'string') {
        throw new Error('Invalid input: Expected a string.');
    }
    return value;
}

function handleInvalidOperation(_operation: any): never {
        throw new Error('Error: Invalid operation. Use add, subtract, multiply, divide, or concatenate.');
}
```

```
// Example usage

try {

    console.log(calculate('add', 10, 5)); // Output: 15

    console.log(calculate('multiply', 3, 4)); // Output: 12

    console.log(calculate('divide', 8, 2)); // Output: 4

    console.log(calculate('subtract', 9, 3)); // Output: 6

     console.log(calculate('concatenate', 'Hello, ', 'World! by Gunjan/D15A-06')); // Output: Hello, World! by Gunjan/D15A-06

    console.log(calculate('modulus' as any, 5, 2)); // Throws an error

     console.log(calculate('add', 'text', 5)); // Throws an error: Both values must be of the same type

} catch (error) {

    console.error((error as Error).message);

}
```

Output:

```
Output:

15
12
4
6
Hello, World! by Gunjan/D15A-06
```

b) Database:

```
interface Subject {
    name: string;
    marks: number;
}
interface Student {
    name: string;
    rollNumber: string;
    subjects: Subject[];
}
function calculateAverage(student: Student): number {
    if (student.subjects.length === 0) {
        return 0;
    }
    const totalMarks = student.subjects.reduce((sum, subject) => sum + subject.marks, 0);
    return totalMarks / student.subjects.length;

function isPassed(averageMarks: number, passingThreshold: number = 40): boolean {
    return averageMarks >= passingThreshold;
}
function displayResult(student: Student, passingThreshold: number = 40): void {
    const averageMarks = calculateAverage(student);
    const passed = isPassed(averageMarks, passingThreshold);
```

```
    console.log(`Student Name: ${student.name}`);
    console.log(`Roll Number: ${student.rollNumber}`);
    student.subjects.forEach(subject => {
        console.log(`${subject.name}: ${subject.marks}`);
    });
    console.log(`Average Marks: ${averageMarks.toFixed(2)}`);
    console.log(`Result: ${passed ? "Passed" : "Failed"}`);
    console.log("--------------------");
}
const student1: Student = {
    name: "Gunjan Chandnani",
    rollNumber: "06",
    subjects: [
        { name: "Math", marks: 45 },
        { name: "Science", marks: 38 },
        { name: "English", marks: 50 },
    ],
};
const student2: Student = {
    name: "Awantika Chandnani",
    rollNumber: "07",
    subjects: [
        { name: "Math", marks: 75 },
```

```
        { name: "Science", marks: 82 },
        { name: "English", marks: 90 },
        { name: "History", marks: 65 },
    ],
};

const student3: Student = {
    name: "palak",
    rollNumber: "05",
    subjects: [],
};
// Displaying the results for each student
displayResult(student1);
displayResult(student2);
displayResult(student3);
const students: Student[] = [student1, student2, student3];
students.forEach(student => displayResult(student));
```

OutPut:

```
Output:

Student Name: Gunjan Chandnani
Roll Number: 06
Math: 45
Science: 38
English: 50
Average Marks: 44.33
Result: Passed
--------------------
Student Name: Awantika Chandnani
Roll Number: 07
Math: 75
Science: 82
English: 90
History: 65
Average Marks: 78.00
Result: Passed
--------------------
Student Name: palak
Roll Number: 05
Average Marks: 0.00
Result: Failed
```

```
Student Name: Gunjan Chandnani
Roll Number: 06
Math: 45
Science: 38
English: 50
Average Marks: 44.33
Result: Passed
--------------------
Student Name: Awantika Chandnani
Roll Number: 07
Math: 75
Science: 82
English: 90
History: 65
Average Marks: 78.00
Result: Passed
--------------------
Student Name: palak
Roll Number: 05
Average Marks: 0.00
Result: Failed
--------------------
```