EXPERIMENT NO. 1A : TypeScript

Name of Student	**Komal Milind Deolekar**
Class Roll No	**14**
D.O.P.	**28-01-2025**
D.O.S.	**11-02-2025**
Sign and Grade	

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

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"});
```

Github Link :

https://github.com/KomalDeolekar0607/Webx_Lab/tree/main/Webx_Lab_Exp_1a

Theory:

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

TypeScript includes several built-in data types:

Primitive Types (same as JavaScript)

- number: let x: number = 10;
- string: let name: string = "Komal";
- boolean: let isActive: boolean = true;
- null: let value: null = null;
- undefined: let value: undefined = undefined;
- symbol: let sym: symbol = Symbol('id');
- bigint: let bigNum: bigint = 9007199254740991n;

Special Types

- any: Can hold any value (let x: any = "Hello";)
- unknown: Similar to any but requires type checking before usage
- void: Used for functions that do not return a value (function log(): void { console.log("Hello"); })
- never: Used for functions that never return (function error(): never { throw new Error("Error"); })

Complex Types

- Array: let arr: number[] = [1, 2, 3];
- Tuple: let tuple: [string, number] = ["hello", 42];
- Object: let obj: { name: string; age: number } = { name: "Komal", age: 22 };
- Enum: enum Color { Red, Green, Blue }
- Union: let val: string | number = "Hello";

Type Annotations in TypeScript :

Type annotations explicitly declare the type of a variable, function, or object.

Example:

```
let age: number = 25;

let name: string = "Komal";

function add(x: number, y: number): number {

  return x + y;

}
```

Uses :

- Helps in catching errors early during development.
- Improves code readability and maintainability.
- Ensures better tooling and IDE support.

b. How do you compile TypeScript files?

1. Install TypeScript (if not already installed)

```
npm install -g typescript
```

2. Compile a TypeScript file

Run the TypeScript compiler (tsc) on your .ts file:

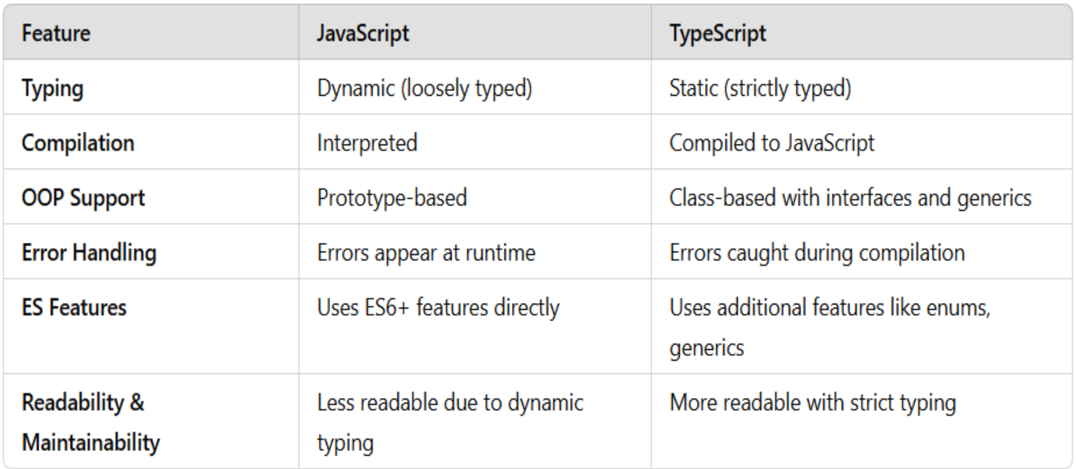
```
tsc filename.ts
```

This generates a filename.js file in the same directory.

3. Now you can run this javascript file with node

```
Node filename.js
```

c. What is the difference between JavaScript and TypeScript?



Feature	JavaScript	TypeScript
Typing	Dynamic (loosely typed)	Static (strictly typed)
Compilation	Interpreted	Compiled to JavaScript
OOP Support	Prototype-based	Class-based with interfaces and generics
Error Handling	Errors appear at runtime	Errors caught during compilation
ES Features	Uses ES6+ features directly	Uses additional features like enums, generics
Readability & Maintainability	Less readable due to dynamic typing	More readable with strict typing

d. Compare how Javascript and Typescript implement Inheritance.

Both JavaScript and TypeScript use class-based inheritance, but TypeScript enforces type safety.

JavaScript Inheritance

```
class Animal {
 constructor(name) {
   this.name = name;
 }
 makeSound() {
   console.log("Some sound");
 }
}
class Dog extends Animal {
 constructor(name, breed) {
   super(name);
   this.breed = breed;
 }
}
const myDog = new Dog("Buddy", "Labrador");
console.log(myDog.name); // Buddy
```

TypeScript Inheritance

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  makeSound(): void {
    console.log("Some sound");
  }
}
class Dog extends Animal {
  breed: string;
  constructor(name: string, breed: string) {
    super(name);
    this.breed = breed;
  }
}
const myDog = new Dog("Buddy", "Labrador");
console.log(myDog.name); // Buddy
```

TypeScript provides strict type checking, preventing unexpected behaviors.

JavaScript Inheritance Features:

- Uses class and extends for inheritance.
- super() is used to call the parent class constructor.
- No strict type checking; properties and methods can be reassigned dynamically.

TypeScript Inheritance Features:

Strict Type Checking: Prevents unintended property changes.

Access Modifiers:

- public: Accessible everywhere (default).
- private: Accessible only within the same class.
- protected: Accessible in the class and its subclasses.

Enforces Method Return Types (: void) for better predictability.

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

Generics make the code flexible while maintaining type safety.

Benefits of Generics:

- Preserves Type Information: Unlike any, generics retain the actual type.
- Reusability: Can be used with different types without rewriting the function.
- Prevents Type Errors: Catches type mismatches during compilation.

Example: Without Generics (any Type)

```
function getValue(value: any): any {

  return value;

}

let result: number = getValue("Hello"); // No error, but incorrect type usage.
```

#This can lead to unexpected runtime errors.

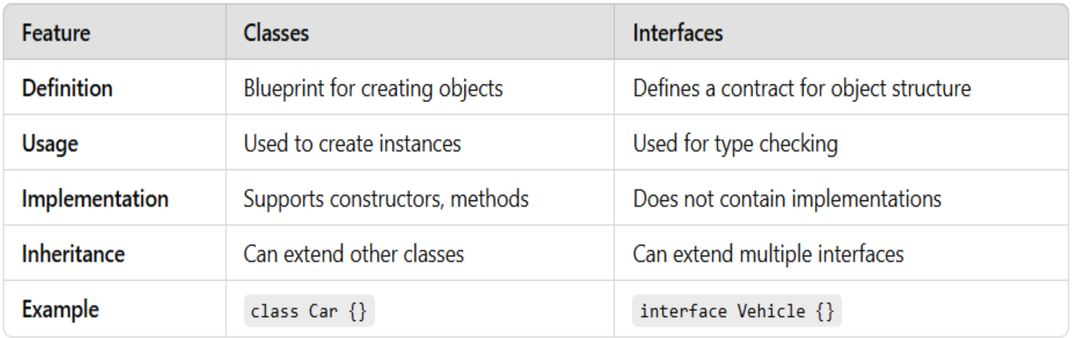
Example: Using Generics

```
function getValue<T>(value: T): T {
  return value;
}
let result: number = getValue<number>(10); // Ensures correct type usage
```

Generics provide better type safety than any.

Generics ensure that the function works with different types without losing type safety, unlike any, which makes debugging harder.

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



Feature	Classes	Interfaces
Definition	Blueprint for creating objects	Defines a contract for object structure
Usage	Used to create instances	Used for type checking
Implementation	Supports constructors, methods	Does not contain implementations
Inheritance	Can extend other classes	Can extend multiple interfaces
Example	`class Car {}`	`interface Vehicle {}`

Example of Class:

```
class Car {
  brand: string;
  constructor(brand: string) {
    this.brand = brand;
  }
}
```

Example of Interface:

```
interface Vehicle {
  brand: string;
}
let myCar: Vehicle = { brand: "Tesla" };
```

Where are interfaces used?

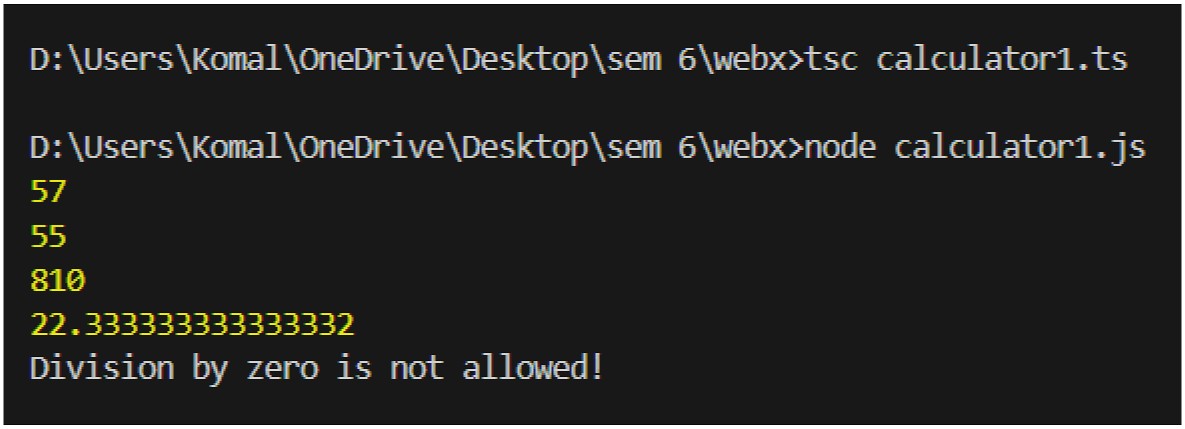
- Defining object structures
- Ensuring consistency in APIs
- Extending types without modifying original classes

Code :

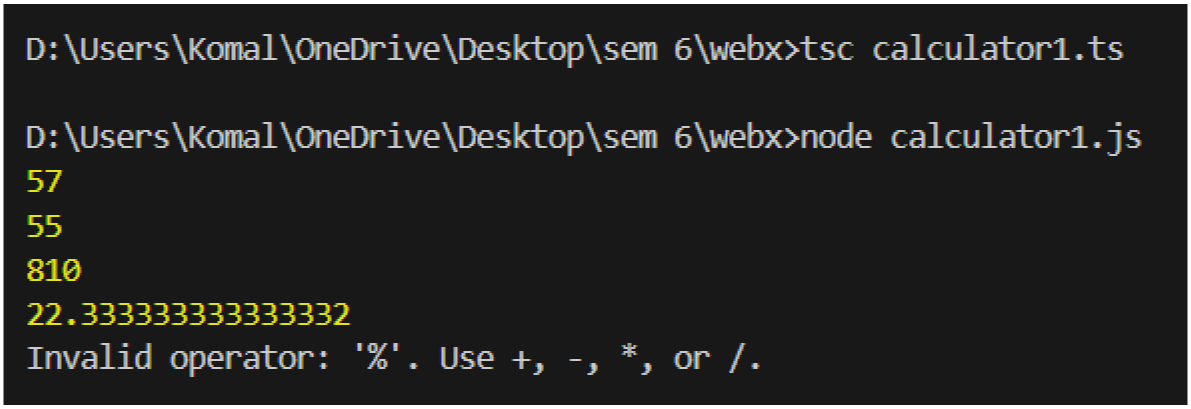
Calculator.tsc

```
function calculator(a: number, b: number, operator: string): number | never {
  switch (operator) {
   case "+":
    return a + b;
   case "-":
    return a - b;
   case "*":
    return a * b;
   case "/":
    if (b === 0) {
     throw new Error("Division by zero is not allowed!"); // Throws error, function never
returns a value
    }
    return a / b;
   default:
    throw new Error(`Invalid operator: '${operator}'. Use +, -, *, or /.`); // Throws error, function
never returns a value
  }
 }


 try {
  console.log(calculator(52, 5, "+"));
  console.log(calculator(87, 32, "-"));
  console.log(calculator(45, 18, "*"));
  console.log(calculator(67, 3, "/"));
  console.log(calculator(10, 0, "/")); // Throws Error: Division by zero is not allowed!
  console.log(calculator(10, 2, "%")); // Throws Error: Invalid operator
 } catch (error) {
  console.error((error as Error).message);
 }
```

Output :



```
D:\Users\Komal\OneDrive\Desktop\sem 6\webx>tsc calculator1.ts

D:\Users\Komal\OneDrive\Desktop\sem 6\webx>node calculator1.js
57
55
810
22.333333333333332
Division by zero is not allowed!
```



```
D:\Users\Komal\OneDrive\Desktop\sem 6\webx>tsc calculator1.ts

D:\Users\Komal\OneDrive\Desktop\sem 6\webx>node calculator1.js
57
55
810
22.333333333333332
Invalid operator: '%'. Use +, -, *, or /.
```

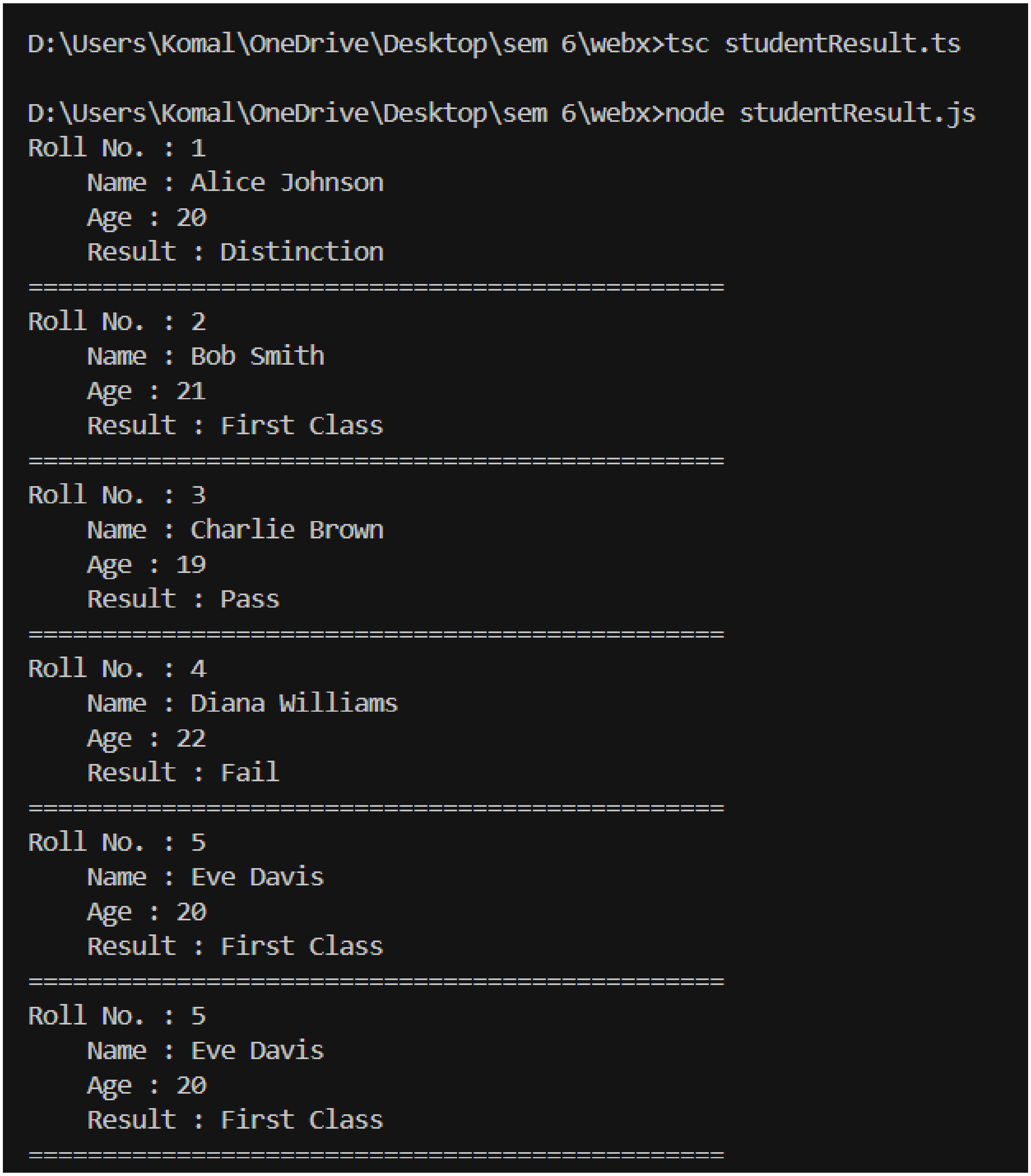
Code :

StudentResult.ts

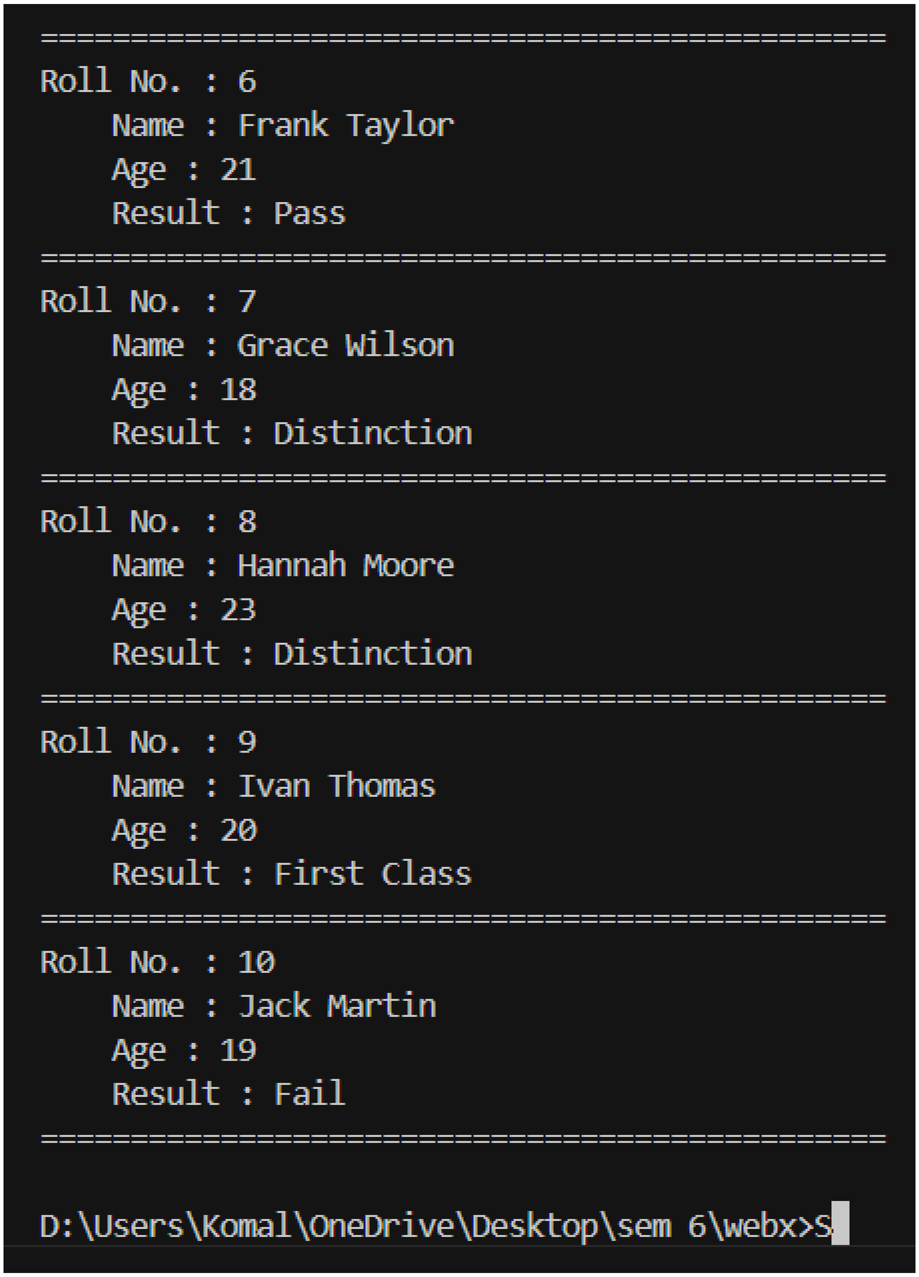
```
type Student = {
    id : number,
    name : String ,
    surname : String,
    marks : number ,
    age : number
}

const students : Student[] = [
{ id: 1, name: "Alice", age: 20, marks: 85 ,surname : "Johnson"},
  { id: 2, name: "Bob", age: 21, marks: 72 , surname:"Smith" },
  { id: 3, name: "Charlie", age: 19, marks: 55 , surname: "Brown"},
  { id: 4, name: "Diana", age: 22, marks: 38 , surname:"Williams"},
  { id: 5, name: "Eve", age: 20, marks: 60, surname:"Davis" },
  { id: 5, name: "Eve", surname: "Davis", age: 20, marks: 60 },
  { id: 6, name: "Frank", surname: "Taylor", age: 21, marks: 47 },
  { id: 7, name: "Grace", surname: "Wilson", age: 18, marks: 78 },
  { id: 8, name: "Hannah", surname: "Moore", age: 23, marks: 91 },
  { id: 9, name: "Ivan", surname: "Thomas", age: 20, marks: 66 },
  { id: 10, name: "Jack", surname: "Martin", age: 19, marks: 33 },
]

students.forEach((student) =>{
    var result :String = "";
    if(student.marks < 40){
        result = "Fail";
    }
    else if(student.marks < 60){
        result = "Pass"
    }
    else if(student.marks < 75){
        result = "First Class"
    }
    else{
        result = "Distinction"
    }

    console.log(`Roll No. : ${student.id}
    Name : ${student.name} ${student.surname}
    Age : ${student.age}
```

```
        Result : ${result}`)

        console.log("===============================================")
    })
```

Output :



```
D:\Users\Komal\OneDrive\Desktop\sem 6\webx>tsc studentResult.ts

D:\Users\Komal\OneDrive\Desktop\sem 6\webx>node studentResult.js
Roll No. : 1
    Name : Alice Johnson
    Age : 20
    Result : Distinction
===============================================
Roll No. : 2
    Name : Bob Smith
    Age : 21
    Result : First Class
===============================================
Roll No. : 3
    Name : Charlie Brown
    Age : 19
    Result : Pass
===============================================
Roll No. : 4
    Name : Diana Williams
    Age : 22
    Result : Fail
===============================================
Roll No. : 5
    Name : Eve Davis
    Age : 20
    Result : First Class
===============================================
Roll No. : 5
    Name : Eve Davis
    Age : 20
    Result : First Class
===============================================
```



```
===============================================
Roll No. : 6
    Name : Frank Taylor
    Age : 21
    Result : Pass
===============================================
Roll No. : 7
    Name : Grace Wilson
    Age : 18
    Result : Distinction
===============================================
Roll No. : 8
    Name : Hannah Moore
    Age : 23
    Result : Distinction
===============================================
Roll No. : 9
    Name : Ivan Thomas
    Age : 20
    Result : First Class
===============================================
Roll No. : 10
    Name : Jack Martin
    Age : 19
    Result : Fail
===============================================

D:\Users\Komal\OneDrive\Desktop\sem 6\webx>S
```

Conclusion :

This TypeScript program effectively demonstrates the use of basic data types (number, string, and boolean) along with arithmetic and logical operators. By implementing fundamental operations like calculations and condition checking, it highlights TypeScript's strong typing system and its benefits in preventing type-related errors.Through this exercise, we understand how TypeScript enforces type safety, making the code more structured, readable, and error-resistant compared to JavaScript. Additionally, this program lays a foundation for more complex applications by introducing the core concepts of data handling, computation, and decision-making in TypeScript.