

Name : Manav Jawrani
Class : D15A
Roll no. : 19

Experiment – 2: TypeScript

Aim: To study Basic constructs in TypeScript.

Problem Statement:

1. Implement a simple Calculator to demonstrate the usage of different data types (include any, never)
2. Create a class with 3 properties with different levels of access modifier and demonstrate its access outside class, sub class and non subclass.
3. Create a Login page includes a function to validate the login credentials. Demonstrate the usage of Generics.

Theory:

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

TypeScript supports the following data types:

1. Number: for numbers of any kind, e.g. integers or floating-point values.
2. String: for sequences of characters, e.g. 'Hello World!'.
3. Boolean: for true/false values.
4. Array: for ordered lists of values, e.g. [1, 2, 3].
5. Tuple: for ordered lists with a specific number of elements, where each element can have a different type, e.g. [string, number].
6. Enum: for a collection of named constant values, e.g. enum Color {Red, Green, Blue}.
7. Any: for a value of any type, e.g. when you don't know the type of a value ahead of time.
8. Void: for functions that don't return a value.
9. Null and Undefined: for values that have the special values null or undefined. Object: for non-primitive data structures, e.g. classes or interfaces.

Type annotations in TypeScript are used to declare the type of a variable or function parameter. This allows the TypeScript compiler to catch type errors at compile time, before the code is executed. Type annotations are optional in TypeScript, but using them makes the code safer and more predictable.

For example:

```
let myName: string = 'John';
```

```
let myAge: number = 30;
```

Here, the type of the variables "myName" and "myAge" are declared using type annotations. The type annotations ensure that the value assigned to these variables is of the correct type, otherwise, the TypeScript compiler will throw an error.

2. How do you compile TypeScript files?

Install the TypeScript compiler (tsc) by running the following command in your terminal:

```
npm install -g typescript
```

Once the TypeScript compiler is installed, you can compile your TypeScript files by running the following command in your terminal:

```
tsc filename.ts
```

Replace "filename.ts" with the actual name of your TypeScript file. The compiler will generate a corresponding JavaScript file with the same name.

To run the compiled JavaScript file, use the following command in your terminal:

```
node filename.js
```

By following these steps, you will be able to compile and execute your TypeScript code.

3. What is the difference between JavaScript and TypeScript?

- a. Type System: TypeScript has a static type system, while JavaScript is dynamically typed.
- b. Type Annotations: TypeScript allows the use of optional type annotations, providing better type checking and error handling. JavaScript does not have type annotations.
- c. Compilation: TypeScript is a statically compiled language, while JavaScript is interpreted.
- d. Features: TypeScript includes features like interfaces, classes, and namespaces, while JavaScript has a more limited feature set.
- e. Use: TypeScript is a superset of JavaScript and can be used anywhere JavaScript can be used. It provides better support for developing large and complex applications. JavaScript is widely used for client-side web development and server-side with Node.js.

4. Compare how Javascript and Typescript implement Inheritance.

Javascript uses prototype-based inheritance, where an object can inherit properties and methods from its prototype. Typescript extends this concept with class-based

inheritance, where objects can inherit from classes, and classes can inherit from other classes. In Typescript, inheritance is achieved through the use of the "extends" keyword and the "super" keyword. Typescript also provides access modifiers (e.g. public, private, protected) to control access to class members and method overriding.

Here's an example of how inheritance is implemented in JavaScript:

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

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

// Derived class
class Dog extends Animal {
  constructor(name) {
    super(name);
  }

  bark() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Fido");
dog.speak(); // Output: Fido makes a sound.
dog.bark(); // Output: Fido barks.
```

And here's an equivalent example of inheritance in TypeScript:

```
// Base class
class Animal {
  constructor(public name: string) {}

  speak() {
```

```
    console.log(`${this.name} makes a sound.`);  
  }  
}
```

// Derived class

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

```
const dog = new Dog("Fido");  
dog.speak(); // Output: Fido makes a sound.  
dog.bark(); // Output: Fido barks.
```

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

In TypeScript, classes and interfaces are two different ways to define object types. Here are the key differences between them:

- Classes are used to create objects that have both properties and methods, while interfaces are used to define the shape of an object.
- Classes can be used to create instances, while interfaces cannot. An interface only provides a blueprint for the structure of an object.
- A class can implement an interface by providing an implementation for its members, while an interface cannot implement a class.
- A class can extend another class or multiple classes, while an interface can only extend another interface.

Interfaces are primarily used to define the shape of an object, which means they specify the names and types of properties that an object should have. Interfaces can also define function signatures, which specify the arguments and return types of a function.

Interfaces are often used in TypeScript to provide type checking for functions that take objects as parameters, or for functions that return objects. For example, a function that expects an object with a name property can be defined like this:

```
interface Person {  
  name: string;  
}
```

```
function sayHello(person: Person) {  
  console.log(`Hello, ${person.name}!`);  
}
```

The Person interface specifies that an object should have a name property of type string. The sayHello() function takes an object that matches this shape and uses it to print a greeting.

Overall, classes are used to define objects with behavior, while interfaces are used to define object shapes and provide type checking.

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

Generics in TypeScript provide a way to write flexible and reusable code that can work with multiple types. Here are some of the ways in which generics make code more flexible:

- **Reusability:** With generics, you can write code that can work with different types of data without having to rewrite the code for each data type. This makes the code more reusable and reduces duplication.
- **Type safety:** Generics provide type safety by allowing you to specify the type of data that a function or class should work with. This helps catch errors at compile time instead of at runtime.
- **Abstraction:** Generics allow you to abstract over the details of the data type and focus on the logic of the code. This makes the code easier to read and understand.
- **Functionality:** Generics enable powerful functionality like mapping, filtering, and reducing collections of data, which can be used with any type of data.

For example, consider a function that takes an array of numbers and returns the sum of the values:

```
function sum(numbers: number[]): number {  
  let total = 0;  
  for (let n of numbers) {  
    total += n;  
  }  
  return total;  
}
```

This function works only with arrays of numbers, and if we want to use it with arrays of other types, we would need to write a new function or modify the existing function. Instead, we can use generics to make the function more flexible:

```
function sum<T extends number>(numbers: T[]): number {  
  let total = 0;  
  for (let n of numbers) {  
    total += n;  
  }  
  return total;  
}
```

In this version of the function, we use a generic type parameter T that extends the number type. This allows the function to work with arrays of any type that extends numbers, such as number[] or Array<number>. By using generics, we have made the function more flexible and reusable.

Output:

Question 1:

Code:

```
import * as readlineSync from 'readline-sync';  
  
// Define a type for the calculator operation  
type Operation = 'add' | 'subtract' | 'multiply' | 'divide';  
  
// Define a function to perform the operation  
function calculate(operation: Operation, num1: number, num2: number): number |  
never {  
  // Check if the operation is valid  
  if (operation === 'add') {  
    return num1 + num2;  
  } else if (operation === 'subtract') {  
    return num1 - num2;  
  } else if (operation === 'multiply') {  
    return num1 * num2;  
  } else if (operation === 'divide') {  
    // Check if the second number is not zero  
    if (num2 !== 0) {  
      return num1 / num2;  
    } else {
```

```

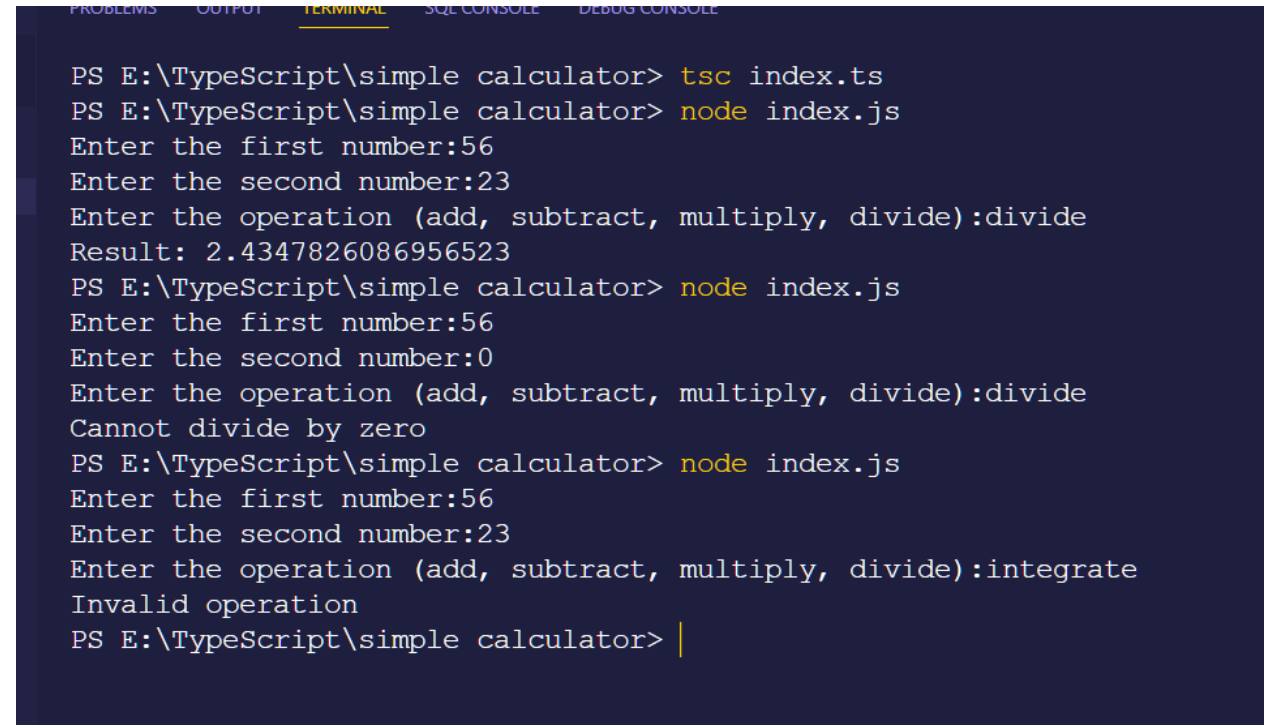
    // If the second number is zero, throw an error
    throw new Error('Cannot divide by zero');
  }
} else {
  // If the operation is not valid, throw an error
  throw new Error('Invalid operation');
}
}

// Prompt the user for input
const num1: number = Number(readlineSync.question('Enter the first number:')); //
num1 is of type number
const num2: number = Number(readlineSync.question('Enter the second number:')); //
num2 is of type number
const operation: Operation = readlineSync.question('Enter the operation (add, subtract,
multiply, divide:~) as Operation; // operation is of type Operation

// Test the calculator
try {
  const result: number = calculate(operation, num1, num2); // result is of type number
  console.log(`Result: ${result}`);
} catch (error: any) {
  console.error(error.message); // error is of type any, which can hold any value
}

```

Screenshot:



```
PS E:\TypeScript\simple calculator> tsc index.ts
PS E:\TypeScript\simple calculator> node index.js
Enter the first number:56
Enter the second number:23
Enter the operation (add, subtract, multiply, divide):divide
Result: 2.4347826086956523
PS E:\TypeScript\simple calculator> node index.js
Enter the first number:56
Enter the second number:0
Enter the operation (add, subtract, multiply, divide):divide
Cannot divide by zero
PS E:\TypeScript\simple calculator> node index.js
Enter the first number:56
Enter the second number:23
Enter the operation (add, subtract, multiply, divide):integrate
Invalid operation
PS E:\TypeScript\simple calculator> |
```

Question 2:

Code:

```
class Example {
  // Public property can be accessed from anywhere
  public publicProperty: number;
  // Protected property can be accessed from subclass
  protected protectedProperty: string;
  // Private property cannot be accessed from outside class
  private privateProperty: boolean;

  constructor(publicProp: number, protectedProp: string, privateProp: boolean) {
    this.publicProperty = publicProp;
    this.protectedProperty = protectedProp;
    this.privateProperty = privateProp;
  }

  // Public method can be accessed from anywhere
  public showPublicProperty() {
    console.log(`Public Property: ${this.publicProperty}`);
  }
}
```



```

// Protected method can be accessed from subclass
protected showProtectedProperty() {
  console.log(`Protected Property: ${this.protectedProperty}`);
}

// Private method cannot be accessed from outside class
private showPrivateProperty() {
  console.log(`Private Property: ${this.privateProperty}`);
}
}

class SubExample extends Example {
  constructor(publicProp: number, protectedProp: string, privateProp: boolean) {
    super(publicProp, protectedProp, privateProp);
  }

  // Protected method can be accessed from subclass
  public showProtectedPropertyOutside() {
    this.showProtectedProperty();
  }
}

// Accessing properties and methods outside the class
let example = new Example(42, "protected", true);
console.log("Outside the class")
console.log(example.publicProperty); // Public property can be accessed
example.showPublicProperty(); // Public method can be accessed

// Accessing properties and methods from subclass
let subExample = new SubExample(123, "subclass", false);
console.log("\nFrom Subclass")
console.log(subExample.publicProperty); // Public property can be accessed from subclass
subExample.showPublicProperty(); // Public method can be accessed from subclass
subExample.showProtectedPropertyOutside(); // Protected method can be accessed from subclass

// Accessing properties and methods from non-subclass
let example2 = new Example(456, "non-subclass", false);
console.log("\nFrom Non-Subclass")

```

```
console.log(example2.publicProperty); // Public property can be accessed from  
non-subclass  
example2.showPublicProperty(); // Public method can be accessed from non-subclass
```

Screenshot:



```
PROBLEMS OUTPUT TERMINAL SQL CONSOLE DEBUG CONSOLE  
  
Windows PowerShell  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows  
  
PS E:\TypeScript\class> tsc class.ts  
PS E:\TypeScript\class> node class.js  
Outside the class  
42  
Public Property: 42  
  
From Subclass  
123  
Public Property: 123  
Protected Property: subclass  
  
From Non-Subclass  
456  
Public Property: 456  
PS E:\TypeScript\class>
```

Question 3:

Code:

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <!-- Bootstrap CSS -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet"

integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65
VohhpuuCOMLAsJc" crossorigin="anonymous" />
  <title>Login Page</title>
</head>

<body>
  <section class="vh-100">
    <div class="container py-5 h-100">
      <div class="row d-flex justify-content-center align-items-center">
        <div class="col-12 col-md-8 col-lg-6 col-xl-5">
          <div class="card shadow-2-strong" style="border-radius: 1rem">
            <div class="card-body p-5 text-center">
              <h3 class="mb-5">Login Form</h3>
              <div class="form-floating mb-4">
                <input type="email" class="form-control" id="floatingInput"
placeholder="name@example.com" style="box-shadow: none"
required />
                <label for="floatingInput">Email address</label>
                <div id="emailShow"></div>
              </div>
              <div class="form-floating mb-4">
                <input type="password" class="form-control"
id="floatingPassword" placeholder="Password"
style="box-shadow: none" required />
                <label for="floatingPassword">Password</label>
                <div id="passwordShow"></div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </section>
</body>
</html>
```



```

    document.getElementById("passwordShow")
);

// Function to submit the user credentials
function submit() {
    // Get the value of the email and password fields
    const email: string = emailInput.value; // Value of the Email entered by the user;
    const password: string = passwordInput.value; // Value of the Password entered
    by the user

    // Create a Credentials object with the email and password values
    const credentials: Credentials = { email, password };

    // Check if the user-entered credentials are valid
    const validCredentials: boolean = validateCredentials(credentials);

    // If the user-entered credentials are valid, update the validation messages to show
    that the email and password are valid
    if (validCredentials) {
        console.log("Email and Password are valid");

        const validationMessage = "You have entered the correct credentials";

        emailValidate.innerHTML = '<p style="color: green">' + validationMessage +
"</p>";
        passwordValidate.innerHTML =
            '<p style="color: green">' + validationMessage + "</p>";
    } else {
        // If the user-entered credentials are not valid, update the validation messages
        to show that the email or password is invalid
        console.log("Email or Password is invalid");

        const validationMessage = "You have entered incorrect credentials";

        emailValidate.innerHTML = '<p style="color: red">' + validationMessage +
"</p>";
        passwordValidate.innerHTML =
            '<p style="color: red">' + validationMessage + "</p>";
    }
}

```

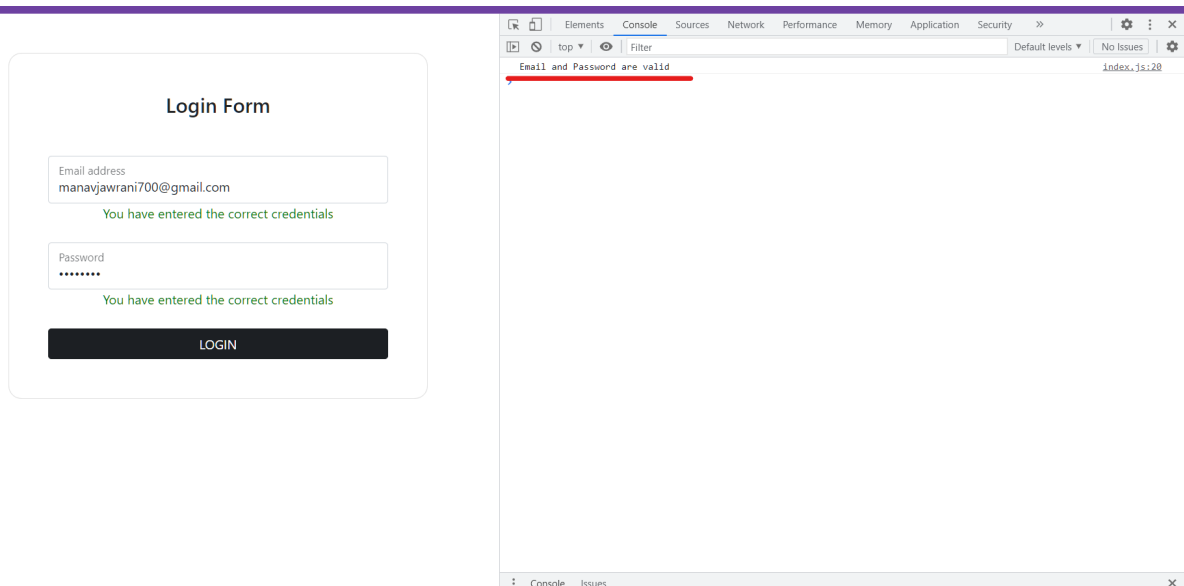
```

// Function to validate the user credentials
function validateCredentials(credentials: Credentials): boolean {
    // Check if the email and password match the expected values
    const validEmail: boolean = credentials.email ===
"manavjawrani700@gmail.com";
    const validPassword: boolean = credentials.password === "Manav123";

    // Return true if both email and password are valid, otherwise return false
    return validEmail && validPassword;
}

```

Screenshot:



Login Form

Email address

manavjawrani700@gmail.com

You have entered incorrect credentials

Password

You have entered incorrect credentials

LOGIN

