Name - Gaurang A Raorane Div - D15A Roll no - 49 Batch - C

**Experiment - 2**

**Aim -** To study Basic constructs in TypeScript.

Theory:-

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

In TypeScript, data types help define the type of values that variables can hold. Here are some of the basic data types in TypeScript:

**Boolean:** Represents a logical value (true or false).

let isDone: boolean = false;

**Number:** Represents both integer and floating-point numbers.

let decimal: number = 6;

let hex: number = 0xf00d;

let binary: number = 0b1010;

let octal: number = 0o744;

**String:** Represents textual data.

let color: string = "blue";

**Array:** Represents a list of elements of a specific type.

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

let fruits: Array<string> = ["apple", "banana", "orange"];

**Tuple:** Represents an array with a fixed number of elements, each with a specific type.

let tuple: [string, number] = ["hello", 10];

**Enum:** A way to define a set of named constant values.

enum Color {

Red,

Green,

Blue

}

let myColor: Color = Color.Green;

**Any:** Represents a variable that can hold values of any type.

let dynamicValue: any = 5;

dynamicValue = "hello";

**Void**: Represents the absence of a value.

function logMessage(): void {

console.log("This function returns nothing.");

}

**Null and Undefined:** Represents the absence of a value or uninitialized value.

let nullValue: null = null;

let undefinedValue: undefined = undefined;

2)How do you compile TypeScript files?

**Method I**

To compile TypeScript files, you need to use the TypeScript Compiler (tsc). TypeScript code is written in .ts files, and the compiler translates these files into JavaScript (.js) files. Here are the general steps to compile TypeScript files:

Install TypeScript:

* If you haven't installed TypeScript globally on your system, you can do so using npm (Node Package Manager) by running the following command in your terminal or

*npm install -g typescript*

* This installs the TypeScript compiler globally on your machine.

**Create a TypeScript file:**

* Write your TypeScript code in a file with a .ts extension. For example, you can create a file named example.ts.

// example.ts

let message: string = "Hello, TypeScript!";

console.log(message);

**Compile the TypeScript file:**

* Run the TypeScript compiler (tsc) followed by the name of your TypeScript file. In our example, you would run:

*tsc example.ts*

* This command generates a corresponding JavaScript file (example.js), along with other files if necessary (e.g., declaration files with a .d.ts extension).

**Run the JavaScript file:**

* Once the TypeScript file is compiled, you can run the generated JavaScript file using a JavaScript runtime or a web browser. In our example, you can run:

*node example.js*

* If you are working with TypeScript in a web development context, you can include the generated JavaScript file in your HTML file and open it in a browser.

**Method II**

Install ts-node using

Npm i ts-node

Then to run file run

npx ts-node <filename>

3)What is the difference between JavaScript and TypeScript?

JavaScript and TypeScript are both programming languages that are widely used for web development, but they have some key differences. Here are the main distinctions between JavaScript and TypeScript:

1. Type System:
   * JavaScript: Dynamically typed language. Variables are not explicitly declared with a type, and their types can change at runtime.
   * TypeScript: Statically typed language. Variables, parameters, and function return types are explicitly declared, and the type is enforced at compile-time.
2. Type Annotations:
   * JavaScript: No built-in syntax for type annotations.
   * TypeScript: Supports type annotations, allowing developers to specify the types of variables, parameters, and return values.
3. Compilation:
   * JavaScript: Interpreted language. Code is executed directly by the browser or server.
   * TypeScript: Compiled language. Code is transpiled into JavaScript before execution. The TypeScript compiler checks for type errors and provides helpful feedback during development.
4. Tooling and IDE Support:
   * JavaScript: Basic tooling and IDE support.
   * TypeScript: Enhanced tooling and IDE support. TypeScript-aware editors, like Visual Studio Code, offer features such as autocompletion, type checking, and refactoring tools.
5. Object-Oriented Programming:
   * JavaScript: Supports object-oriented programming with prototypal inheritance.
   * TypeScript: Adds features from classical object-oriented languages, such as classes, interfaces, and access modifiers.
6. Null and Undefined Handling:
   * JavaScript: Has null and undefined values that can lead to runtime errors.
   * TypeScript: Introduces a strictNullChecks option to help catch and prevent null and undefined errors during development.
7. Code Readability and Maintainability:
   * JavaScript: Can become challenging to maintain in large codebases due to the lack of strict typing.
   * TypeScript: Enhances code readability and maintainability by providing clearer documentation through type annotations.
8. Compatibility:
   * JavaScript: Compatible with all browsers and environments that support ECMAScript standards.
   * TypeScript: Needs to be transpiled to JavaScript before execution, but the resulting code is compatible with all JavaScript environments.
9. Community and Ecosystem:
   * JavaScript: Has a large and mature ecosystem with a vast number of libraries and frameworks.
   * TypeScript: Growing in popularity, and many JavaScript libraries are adopting TypeScript or providing TypeScript typings.
10. Adoption:
    * JavaScript: Ubiquitous and widely used for both frontend and backend development.
    * TypeScript: Increasing adoption, especially in large-scale projects and enterprise applications.

4)Compare how Javascript and Typescript implement Inheritance.

### JavaScript Inheritance:

* Prototypal Inheritance:
  + JavaScript uses prototypal inheritance, where objects inherit properties and methods from other objects through their prototypes.
  + Objects can serve as prototypes for other objects, forming a chain.
* prototype Property:
  + Each object in JavaScript has a prototype property that references another object. Properties and methods not found in the object itself are looked up in its prototype chain.
* Constructor Functions:
  + Constructors can be used to create objects with shared properties and methods.
  + The new keyword is used to instantiate objects from a constructor.

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

In TypeScript, both classes and interfaces play essential roles, but they serve different purposes. Let's explore the differences between classes and interfaces and understand where interfaces are commonly used.

### **Classes:**

* Definition:
  + Classes are a fundamental part of object-oriented programming in TypeScript.
  + They allow you to define blueprints for creating objects, including both properties and methods.
* Object Creation:
  + Classes are used to create instances of objects with specific behavior and properties.
  + They provide a way to encapsulate data and behavior within a single unit.
* Inheritance:
  + Classes support inheritance, allowing a class to inherit properties and methods from another class using the extends keyword.
* Access Modifiers:
  + Classes can use access modifiers (public, private, protected) to control the visibility of properties and methods.
* Constructor:
  + Classes have constructors that are executed when an instance is created, allowing you to initialize object properties.

class Animal {

private name: string;

constructor(name: string) {

this.name = name;

}

public sayHello() {

console.log(`Hello, I'm ${this.name}`);

}

}

class Dog extends Animal {

public bark() {

console.log("Woof!");

}

}

const myDog = new Dog("Buddy");

myDog.sayHello();

myDog.bark();

### Interfaces:

* Definition:
  + Interfaces are used to define contracts for the shape of objects.
  + They specify the structure that an object must adhere to but do not provide an implementation.
* Object Shape:
  + Interfaces define the structure of an object by listing the properties and their types, and optionally, method signatures.
* No Implementation:
  + Interfaces do not include the actual implementation of methods or properties. They focus solely on the shape of the object.
* Optional Properties:
  + Interfaces support optional properties, allowing you to define properties that may or may not be present in an implementing object.
* Declaration Merging:
  + TypeScript allows interfaces to be merged. If multiple interfaces with the same name are declared, they are merged into a single interface.

interface Animal {

name: string;

sayHello(): void;

}

interface Dog extends Animal {

bark(): void;

}

const myDog: Dog = {

name: "Buddy",

sayHello() {

console.log(`Hello, I'm ${this.name}`);

},

bark() {

console.log("Woof!");

},

};

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 by allowing types to be parameterized. Generics make it possible to create functions, classes, and interfaces that can work with different types without sacrificing type safety. Here are some key reasons why generics make code more flexible and why you might choose them over other types:

### 1. Type Flexibility:

* Generics allow you to write functions and data structures that work with a variety of types, including user-defined types.
* This flexibility enables you to create more versatile and reusable components.

### 2. Code Reusability:

* Generics promote code reusability by making it possible to write functions or classes that can operate on different types without duplication of code.
* You can write a generic function once and use it with various types.

### 3. Maintaining Type Safety:

* Generics maintain type safety by allowing you to specify the types that a function or class will work with.
* The TypeScript compiler enforces these types, preventing unintended type-related errors.

### 4. Flexible Data Structures:

* Generics are commonly used in the design of data structures, such as arrays, linked lists, and queues, to create structures that can hold elements of different types.
* This flexibility is crucial when working with collections of heterogeneous elements.

### 5. Functionality Agnostic to Data Type:

* Generics allow you to write functions or classes that focus on functionality without being tied to specific data types.
* This separation of concerns enhances the clarity and maintainability of your code.

### 6. Dynamic and Static Typing:

* Generics provide a bridge between dynamic and static typing. They allow you to write code that works with a variety of types while benefiting from TypeScript's static type checking.

**Implementation**

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

**Input :-**

type Operation = "add" | "subtract" | "multiply" | "divide";

function calculate(operator: Operation, num1: number, num2: number): number | string {

let result: any;

switch (operator) {

case "add":

result = num1 + num2;

break;

case "subtract":

result = num1 - num2;

break;

case "multiply":

result = num1 \* num2;

break;

case "divide":

result = num2 !== 0 ? num1 / num2 : "Cannot divide by zero";

break;

default:

const exhaustiveCheck: never = operator;

}

return result;

}

console.log(calculate("add", 5, 3));

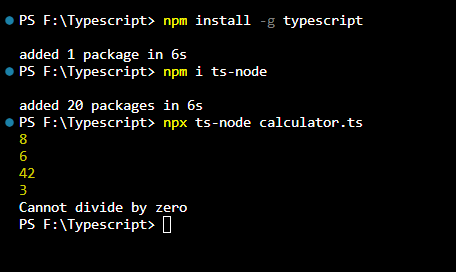
console.log(calculate("subtract", 10, 4));

console.log(calculate("multiply", 6, 7));

console.log(calculate("divide", 9, 3));

console.log(calculate("divide", 8, 0));

**Output:-**

****

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 {

public username: string;

protected email: string;

private password: string;

constructor(username: string, email: string, password: string) {

this.username = username;

this.email = email;

this.password = password;

}

displayUserInfo(): void {

console.log(`Username: ${this.username}, Email: ${this.email}`);

}

protected showPassword(): void {

console.log(`Password: ${this.password}`);

}

}

class Admin extends User {

public isAdmin: boolean;

constructor(username: string, email: string, password: string, isAdmin: boolean)

super(username, email, password);

this.isAdmin = isAdmin;

}

displayUserInfo(): void {

super.displayUserInfo();

console.log(`Admin: ${this.isAdmin}`);

}

showAdminStatus(): void {

console.log(`Admin status: ${this.isAdmin ? 'Yes' : 'No'}`);

}

}

class DataManager {

private data: string[];

constructor() {

this.data = [];

}

addData(item: string): void {

this.data.push(item);

}

private getData(): string[] {

return this.data;

}

displayData(): void {

console.log("Data Manager Contents:", this.getData());

}

}

const regularUser = new User("GaurangRaorane", "gaurang@gmail.com", "password123")

console.log("User's Username:", regularUser.username);

regularUser.displayUserInfo();

console.log("=".repeat(50));

const adminUser = new Admin("GaurangRaorane", "gaurang@gmail.com", "adminPassword", true);

console.log("Admin's Username:", adminUser.username);

adminUser.displayUserInfo()

console.log("Admin Status:", adminUser.isAdmin);

adminUser.showAdminStatus();

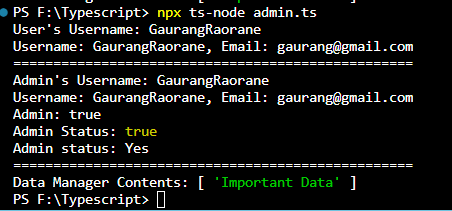
console.log("=".repeat(50));

const dataManager = new DataManager();

dataManager.addData("Important Data");

dataManager.displayData();

**Output:-**

****

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

abstract class Shape {

abstract calculateArea(): number;

abstract calculatePerimeter(): number;

}

class Circle extends Shape {

constructor(private radius: number) {

super();

}

calculateArea(): number {

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

}

calculatePerimeter(): number {

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

}

}

class Rectangle extends Shape {

constructor(private width: number, private height: number) {

super();

}

calculateArea(): number {

return this.width \* this.height;

}

calculatePerimeter(): number {

return 2 \* (this.width + this.height);

}

}

class Triangle extends Shape {

constructor(private sideA: number, private sideB: number, private sideC: number) {

super();

}

calculateArea(): number {

const s = (this.sideA + this.sideB + this.sideC) / 2;

return Math.sqrt(s \* (s - this.sideA) \* (s - this.sideB) \* (s - this.sideC));

}

calculatePerimeter(): number {

return this.sideA + this.sideB + this.sideC;

}

}

class ShapeCalculator {

static calculateTotalArea(shapes: Shape[]): number {

return shapes.reduce((totalArea, shape) => totalArea + shape.calculateArea(), 0);

}

static calculateTotalPerimeter(shapes: Shape[]): number {

return shapes.reduce((totalPerimeter, shape) => totalPerimeter + shape.calculatePerimeter(), 0);

}

}

const circle = new Circle(5);

const rectangle = new Rectangle(4, 6);

const triangle = new Triangle(3, 4, 5);

console.log("Circle - Area:", circle.calculateArea(), "Perimeter:", circle.calculatePerimeter());

console.log("Rectangle - Area:", rectangle.calculateArea(), "Perimeter:", rectangle.calculatePerimeter());

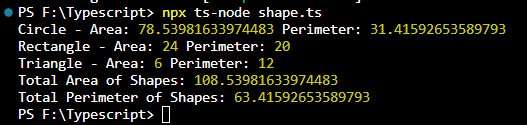
console.log("Triangle - Area:", triangle.calculateArea(), "Perimeter:", triangle.calculatePerimeter());

const shapesArray: Shape[] = [circle, rectangle, triangle];

console.log("Total Area of Shapes:", ShapeCalculator.calculateTotalArea(shapesArray));

console.log("Total Perimeter of Shapes:", ShapeCalculator.calculateTotalPerimeter(shapesArray));

**Output:-**

****

**Conclusion:-**

Through this experiment we understood Typescript how it is different from Javascript and many of its features that makes it superset of javascript also understood how it offers robust type-checking, object-oriented features, and enhanced tooling, promoting scalable, maintainable, and safer JavaScript development.