**PROGRAM-2 (TYPE SCRIPT)**

1. **Write a Typescript program to understand simple and special types.**

**✅ Simple Types:**

* number, string, boolean

**✅ Special Types:**

* any, unknown, void, null, undefined, never

**#PROGRAM CODE**

// Simple Types

let myName: string = "Alice"; // string type

let myAge: number = 30; // number type

let isStudent: boolean = true; // boolean type

// Output simple types

console.log("Name:", myName);

console.log("Age:", myAge);

console.log("Is Student:", isStudent);

// Special Types

let notKnown: any = "Hello"; // any type (can be anything)

notKnown = 100; // can change to number

console.log("Any type value:", notKnown);

let unknownValue: unknown = "I might be anything";

// console.log(unknownValue.toUpperCase()); // ❌ Error without type check

if (typeof unknownValue === "string") {

console.log("Unknown as string:", unknownValue.toUpperCase());

}

// null and undefined

let nothingHere: null = null;

let notAssigned: undefined = undefined;

console.log("Null value:", nothingHere);

console.log("Undefined value:", notAssigned);

// void type - for functions that return nothing

function greet(): void {

console.log("Hello from a void function!");

}

greet();

// never type - for functions that never return

function throwError(message: string): never {

throw new Error(message);

}

// throwError("Something went wrong!"); // Uncomment to test

**Steps to Run:**

1. Save file as types.ts
2. Compile: tsc types.ts
3. node types.js

Install TypeScript Compiler (tsc):

**✅ Step 1: Install Node.js (if not already installed)**

1. Go to: <https://nodejs.org>
2. Download and install the **LTS version** (Long-Term Support)
3. After installation, open **Command Prompt** and check:

node -v

npm -v

**✅ Step 2: Install TypeScript globally**

Run this in Command Prompt:

**npm install -g typescript**

The -g flag installs it **globally** so you can run tsc from any folder.

**✅ Step 3: Verify the installation**

Run this to check if tsc is working:

tsc -v

**✅ Step 4: Compile and Run TypeScript File**

1. Save your code in a file like types.ts
2. Compile it:

tsc types.ts

It will generate a types.js file.

Run it with Node.js:

node types.js

**OUTPUT:**

Name: Alice

Age: 30

Is Student: true

Any type value: 100

Unknown as string: I MIGHT BE ANYTHING

Null value: null

Undefined value: undefined

Hello from a void function!

**b) Write a program to understand function parameter and return types.**

**#PROGRAM CODE**

// Function with number parameters and number return type

function add(a: number, b: number): number {

return a + b;

}

// Function with string parameters and string return type

function greet(name: string): string {

return `Hello, ${name}!`;

}

// Function with boolean parameter and void return type

function checkStatus(isActive: boolean): void {

if (isActive) {

console.log("Status: Active");

} else {

console.log("Status: Inactive");

}

}

// Function with optional parameter

function multiply(a: number, b?: number): number {

return b ? a \* b : a \* 1; // default to 1 if b is undefined

}

// Function with default parameter

function power(base: number, exponent: number = 2): number {

return base \*\* exponent;

}

// Calling the functions

console.log("Add:", add(10, 20));

console.log(greet("Alice"));

checkStatus(true);

console.log("Multiply:", multiply(5));

console.log("Power:", power(3));

**Output:**

Add: 30

Hello, Alice!

Status: Active

Multiply: 5

Power: 9

**C) Write a program to show the importance with Arrow function. Use optional, default and REST parameters.**

**ARROW function**

An **arrow function** in JavaScript is a shorter way to write function expressions. It was introduced in ES6 (ECMAScript 2015) and is especially useful for writing concise callbacks and anonymous functions.

**🔹 Basic Syntax:**

const add = (a, b) => a + b;

**🔹 Features of Arrow Functions:**

1. **Concise syntax** — cleaner and shorter.
2. **Does not have its own this** — it captures this from the surrounding context.
3. **Cannot be used as constructors** — new with arrow function throws an error.
4. **Does not have arguments object** — must use rest parameters instead.

**🔹 Examples:**

**✅ 1. Simple Arrow Function**

const greet = () => console.log("Hello, world!");

greet();

**✅ 2. Arrow Function with Parameters**

const multiply = (x, y) => x \* y;

console.log(multiply(4, 5)); // Output: 20

**✅ 3. With Default Parameter**

const welcome = (name = "Guest") => `Welcome, ${name}!`;

console.log(welcome()); // Output: Welcome, Guest!

console.log(welcome("Ambika")); // Output: Welcome, Ambika!

**✅ 4. With Optional Parameter (using undefined)**

const showMessage = (message, sender = "Admin") => {

console.log(`${sender}: ${message}`);

};

showMessage("This is a test"); // Admin: This is a test

**✅ 5. With Rest Parameters**

const sumAll = (...nums) => nums.reduce((acc, val) => acc + val, 0);

console.log(sumAll(1, 2, 3, 4)); // Output: 10

**✅ Summary Table**

| **F Signature** |  | **A Arrow Function** | **T Traditional Function** |
| --- | --- | --- | --- |
| S syntax |  | Concise | V verbose |
| this context |  | Lexical (this from scope) | Own this |
| arguments object |  | ❌ Not available | ✅ Available |
| Used as constructor |  | ❌ Not possible | ✅ Possible |

**What are Rest Parameters in JavaScript?**

**Rest parameters allow a function to accept any number of arguments as an array. It's a way to represent "the rest" of the arguments using the ... syntax.**

**🔹 Syntax:**

**function myFunction(...args) {**

**// args is an array**

**}**

**In arrow functions:**

**const myFunction = (...args) => {**

**console.log(args);**

**};**

**# PROGRAM CODE:**

// Arrow function with default parameters

const greet = (name: string = "Guest"): string => {

return `Hello, ${name}!`;

};

// Arrow function with optional parameter

const fullName = (firstName: string, lastName?: string): string => {

return lastName ? `${firstName} ${lastName}` : firstName;

};

// Arrow function with rest parameters

const sum = (...numbers: number[]): number => {

return numbers.reduce((total, num) => total + num, 0);

};

// Arrow function assigned to a variable (shows reusability and compact syntax)

const square = (n: number): number => n \* n;

// Using all functions

console.log(greet()); // default name

console.log(greet("Alice"));

console.log(fullName("John")); // optional last name

console.log(fullName("John", "Doe")); // with last name

console.log("Sum:", sum(1, 2, 3, 4, 5)); // REST parameters

console.log("Square of 6:", square(6));

**OUTPUT:**

Hello, Guest!

Hello, Alice!

John

John Doe

Sum: 15

Square of 6: 36

* 1. Write a program to understand the working of typescript with class, constructor, properties, methods and access specifiers.

**Explanation of the working of typescript with class, constructor, properties, methods and access specifiers.**

**TypeScript** works with **class**, **constructor**, **properties**, **methods**, and **access specifiers** — all concepts from object-oriented programming.

**✅ 1. Class in TypeScript**

A class is a blueprint for creating objects. It can contain:

* **Properties** (variables inside the class)
* **Constructor** (special method to initialize objects)
* **Methods** (functions inside the class)
* **Access specifiers** (to control visibility: public, private, protected)

**✅ 2. Basic Structure Example**

class Person {

// Properties

name: string;

age: number;

// Constructor

constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

// Method

greet(): void {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

}

// Create an object

const p1 = new Person("Alice", 30);

p1.greet(); // Output: Hello, my name is Alice and I am 30 years old.

**✅ 3. Access Specifiers in TypeScript**

| **Access Specifier** | **Meaning** |
| --- | --- |
| public | Accessible from anywhere (default) |
| private | Accessible only within the class |
| protected | Accessible within the class and its subclasses |

**🔹 Example with public, private, protected:**

class Employee {

public name: string; // accessible anywhere

private salary: number; // only inside this class

protected department: string; // inside this class and subclasses

constructor(name: string, salary: number, department: string) {

this.name = name;

this.salary = salary;

this.department = department;

}

showDetails(): void {

console.log(`${this.name} works in ${this.department}.`);

console.log(`Salary: ${this.salary}`); // valid

}

}

class Manager extends Employee {

showDepartment(): void {

console.log(`Manager is in ${this.department}`); // ✅ OK (protected)

// console.log(this.salary); // ❌ Error: salary is private

}

}

const emp = new Employee("John", 50000, "IT");

emp.showDetails();

// console.log(emp.salary); // ❌ Error: salary is private

// console.log(emp.department); // ❌ Error: department is protected

**✅ Summary:**

| **Concept** | **Purpose** |
| --- | --- |
| class | Defines a blueprint for objects |
| constructor | Initializes the object properties |
| property | Holds data related to the object |
| method | Performs actions (functions inside the class) |
| public | Can be accessed anywhere |
| private | Can only be accessed inside the class |
| protected | Can be accessed inside the class and subclasses |

**#PROGRAM CODE**

class Person {

// Properties with access specifiers

public name: string;

private age: number;

protected gender: string;

// Constructor to initialize properties

constructor(name: string, age: number, gender: string) {

this.name = name;

this.age = age;

this.gender = gender;

}

// Public method (accessible from outside)

public greet(): void {

console.log(`Hello, my name is ${this.name}.`);

}

// Private method (accessible only inside class)

private showAge(): void {

console.log(`I am ${this.age} years old.`);

}

// Protected method (accessible in subclass)

protected displayGender(): void {

console.log(`Gender: ${this.gender}`);

}

// Method to access private method

public displayDetails(): void {

this.showAge(); // private method

this.displayGender(); // protected method

}

}

// Subclass that inherits Person

class Student extends Person {

private studentId: number;

constructor(name: string, age: number, gender: string, studentId: number) {

super(name, age, gender);

this.studentId = studentId;

}

public showStudentInfo(): void {

console.log(`Student ID: ${this.studentId}`);

this.displayGender(); // Can access protected method

}

}

// Using the class

const person1 = new Person("Alice", 30, "Female");

person1.greet();

person1.displayDetails();

// person1.showAge(); âŒ Error: private method

// person1.displayGender(); âŒ Error: protected method

const student1 = new Student("Bob", 20, "Male", 101);

student1.greet();

student1.displayDetails();

student1.showStudentInfo();

**OUTPUT:**

Hello, my name is Alice.

I am 30 years old.

Gender: Female

Hello, my name is Bob.

I am 20 years old.

Gender: Male

Student ID: 101

Gender: Male

* 1. **Write a program to understand the working of namespaces and modules.**

In TypeScript, **namespaces** and **modules** are used to **organize and encapsulate code**.

**✅ DIFFERENCE AT A GLANCE:**

| **Feature** | **Namespace** | **Module** |
| --- | --- | --- |
| Scope | Global (if not wrapped in a module) | File-based (each file is its own module) |
| Syntax | namespace {} | export, import, and ES6 modules |
| Use Case | Grouping logically related code | Code reuse, separation, scalability |

**✅ Example Program for Namespace**

// TypeScript Namespace Example

namespace MathOperations {

export function add(a: number, b: number): number {

return a + b;

}

export function multiply(a: number, b: number): number {

return a \* b;

}

}

// Using the namespace

console.log("Addition:", MathOperations.add(5, 3)); // Output: 8

console.log("Multiplication:", MathOperations.multiply(5, 3)); // Output: 15

✅ export is used to make functions available outside the namespace.

**✅ Example Program for Module**

Assume you have two files:

**🔹 mathUtil.ts (Module File)**

// mathUtil.ts

export function subtract(a: number, b: number): number {

return a - b;

}

export function divide(a: number, b: number): number {

return a / b;

}

**🔹 main.ts (Importing Module)**

// main.ts

import { subtract, divide } from "./mathUtil";

console.log("Subtraction:", subtract(10, 4)); // Output: 6

console.log("Division:", divide(20, 4)); // Output: 5

🛠️ **To compile and run:**

1. Compile both files using tsc:

tsc mathUtil.ts main.ts

1. Run the resulting JavaScript:

node main.js

**✅ Summary:**

* **Namespaces** are good for organizing code in one file.
* **Modules** are file-based and preferred for modern projects, especially with tools like Webpack, ESBuild, etc.

**# PROGRAM CODE**

namespace MathUtils {

export function add(a: number, b: number): number {

return a + b;

}

export function subtract(a: number, b: number): number {

return a - b;

}

}

// Using the namespace

console.log("Namespace Add:", MathUtils.add(10, 5));

console.log("Namespace Subtract:", MathUtils.subtract(10, 5));

**OUTPUT:**

Namespace Add: 15

Namespace Subtract: 5

Let's simulate a module and a namespace **in one file** using a module-aware environment (like Node.js + tsc with ES module support):

1. **Write a program to understand generics with variables, functions and constraints.**

**Generics** in TypeScript — a powerful feature that allows you to write reusable, type-safe code.

**✅ What are Generics?**

**Generics** enable writing code that works with **any data type**, while still preserving **type safety**.

**🔹 Syntax:**

function identity<T>(arg: T): T {

return arg;

}

Here, T is a placeholder for a **type** that will be provided later.

**✅ 1. Generics with Variables**

let value: <T>(arg: T) => T;

value = function <T>(arg: T): T {

return arg;

};

console.log(value<string>("Hello")); // Output: Hello

console.log(value<number>(100)); // Output: 100

Here, value is a generic variable that holds a function accepting and returning the same type.

**✅ 2. Generics with Functions**

function getFirst<T>(items: T[]): T {

return items[0];

}

console.log(getFirst<string>(["apple", "banana"])); // Output: apple

console.log(getFirst<number>([1, 2, 3])); // Output: 1

🔹 T[] means an array of type T, and the function returns a value of type T.

**✅ 3. Generics with Constraints**

You can restrict a generic type to ensure it meets certain **criteria** using extends.

**🔹 Example: Constrain to objects with .length**

function logLength<T extends { length: number }>(item: T): void {

console.log(item.length);

}

logLength("Hello"); // ✅ string has length → Output: 5

logLength([1, 2, 3]); // ✅ array has length → Output: 3

// logLength(123); ❌ Error: number has no length

**✅ 4. Generics with Interfaces and Classes**

**🔹 Interface Example:**

interface Box<T> {

content: T;

}

let stringBox: Box<string> = { content: "Books" };

let numberBox: Box<number> = { content: 123 };

**🔹 Class Example:**

class Container<T> {

private value: T;

constructor(val: T) {

this.value = val;

}

getValue(): T {

return this.value;

}

}

const c1 = new Container<number>(42);

console.log(c1.getValue()); // Output: 42

**✅ Summary Table**

| **Use Case** | **Example** |
| --- | --- |
| Generic Function | function<T>(arg: T): T {} |
| Generic Variable | let val: <T>(x: T) => T; |
| Constraint | <T extends { length: number }> |
| Generic Interface | interface Box<T> { content: T; } |
| Generic Class | class Container<T> {} |

**#PROGRAM CODE**

// âœ… 1. Generic Variable

let genericValue: <T>(value: T) => T = function<T>(value: T): T {

return value;

};

console.log("Generic Variable (number):", genericValue<number>(42));

console.log("Generic Variable (string):", genericValue<string>("Hello"));

// âœ… 2. Generic Function

function identity<T>(arg: T): T {

return arg;

}

console.log("Generic Function (boolean):", identity<boolean>(true));

console.log("Generic Function (array):", identity<number[]>([1, 2, 3]));

// âœ… 3. Generic Function with Multiple Types

function pair<A, B>(first: A, second: B): [A, B] {

return [first, second];

}

console.log("Generic Pair:", pair<string, number>("Age", 25));

// âœ… 4. Generic with Constraint

interface HasLength {

length: number;

}

function printLength<T extends HasLength>(item: T): void {

console.log("Length is:", item.length);

}

printLength("TypeScript"); // string has length

printLength([1, 2, 3, 4]); // array has length

// printLength(123); âŒ Error: number doesn't have 'length'

// âœ… 5. Generic Class

class Box<T> {

private \_value: T;

constructor(value: T) {

this.\_value = value;

}

getValue(): T {

return this.\_value;

}

}

const stringBox = new Box<string>("Generic Box");

console.log("Box Value:", stringBox.getValue());

**OUTPUT:**

Generic Variable (number): 42

Generic Variable (string): Hello

Generic Function (boolean): true

Generic Function (array): [ 1, 2, 3 ]

Generic Pair: [ 'Age', 25 ]

Length is: 10

Length is: 4