PROGRAM-2
(TypeScript)

- **TypeScript is a syntactic superset of JavaScript which adds static typing** developed by Microsoft.
- TypeScript is a programming language that's a superset of JavaScript, which means it understands all of JavaScript's syntax and capabilities, while adding additional features.
- JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.
- TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match.
- TypeScript uses compile time type checking. Which means it checks if the specified types match **before** running the code, not **while** running the code and if there are type errors it will not successfully compile, which prevents bad code from going out to be run.

How Does TypeScript Work?

- Browsers and most other platforms like node that run JavaScript do not have native support for TypeScript. TypeScript comes with a **compiler called tsc** which will convert them into JavaScript so that the browser (or whatever platform you're using) will be able to run your code as if it was JavaScript.

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.
3. Run it with Node.js: **node types.js**

a. 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 your code in a file like types.ts
2. Compile it**:** **tsc types.ts**
   It will generate a types.js file.
3. 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.

Function Parameter Types

When you define a function in TypeScript, you can declare the type of each parameter the function accepts. This ensures that the function is used correctly by enforcing type rules at compile time.

Function Return Types

TypeScript lets you **explicitly specify the return type** of a function. This helps prevent bugs by ensuring that the function returns the expected type.

Optional Parameters

Sometimes, not all parameters are required. You can mark a parameter as **optional** using the ? symbol.

Default Parameters

You can also assign default values to parameters.

Arrow Functions with Types

Arrow functions also support type annotations.

Benefits of Using Function Types

- Prevents bugs by catching type mismatches
- Improves code readability and documentation
- Enables better autocomplete and error checking in IDEs

#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));
```

Steps to Run:

1. Save your code in a file like 2b.ts
2. Compile it**:** **tsc 2b.ts**
   It will generate a types.js file.
3. Run it with Node.js: **node 2b.js**

Output:

Add: 30

Hello, Alice!

Status: Active

Multiply: 5

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:**
  - ✓ **Concise syntax** — cleaner and shorter.
  - ✓ **Does not have its own this** — it captures this from the surrounding context.
  - ✓ **Cannot be used as constructors** — new with arrow function throws an error.
  - ✓ **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("AIML"));  // Output: Welcome, AIML!
```

✅ **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
```

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));
```

Steps to Run:

1. Save your code in a file like 2C.ts
2. Compile it**:** **tsc 2C.ts**
   It will generate a types.js file.

3. Run it with Node.js: **node 2C.js**

OUTPUT:

```
 Hello, Guest!
Hello, Alice!
John
John Doe
Sum: 15
Square of 6: 36
```

d. Write a program to understand 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.

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();
```

Steps to Run:

1. Save your code in a file like 2d.ts
2. Compile it**:** **tsc 2d.ts**
   It will generate a types.js file.

3. Run it with Node.js: **node 2d.js**

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

e) Write a program to understand the working of namespaces and modules.

- **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));
```

Steps to Run:

1. Save your code in a file like 2e.ts
2. Compile it**:** **tsc 2e.ts**
   It will generate a types.js file.
3. Run it with Node.js: **node 2e.js**

OUTPUT:

Namespace Add: 15

Namespace Subtract: 5

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

What are Generics?

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

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.

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

```
// 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"));


// 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]));


//  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));
```

```
// 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'


// 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());
```

Steps to Run:

1. Save your code in a file like 2F.ts
2. Compile it**:** **tsc 2F.ts**
   It will generate a types.js file.

3. Run it with Node.js: **node 2F.js**

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
```