TYPESCRIPT

OUTLINE

- Introduction
- Basic Types
- Functions
- Interfaces and classes
- Advanced Types
- Decorators
- TypeScript vs JavaSript
- TPs

INTRODUCTION

INTRODUCTION TO TYPESCRIPT

- TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript.
- It adds optional static typing, classes, interfaces, and other features to JavaScript.
- The main difference between JavaScript and TypeScript lies in their type systems and how they are used in development:

- 1. Type System
- JavaScript:

- Dynamically typed: Variables can hold values of any type, and types are determined at runtime.

- No built-in support for type annotations or type checking.

- TypeScript:

INTRODUCTION TO TYPESCRIPT

- TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript.
- It adds optional static typing, classes, interfaces, and other features to JavaScript.
- **Why Use TypeScript?**
- Improves code quality and readability.
- Catches errors at compile-time rather than runtime.
- Enhances tooling support with features like autocompletion and refactoring.
- **Setting Up TypeScript**

  npm install -g typescript
- **Compile a TypeScript file:**

  tsc filename.ts

WRITING TYPESCRIPT

- We can write TypeScript in Visual Studio
- Visual Studio fully supports TypeScript development with built-in tools for compilation, debugging.
- Visual Studio 2022+ comes with TypeScript pre-installed.
- How to Set Up TypeScript in Visual Studio
- **Step 1: Install TypeScript SDK** if needed
  - Download from TypeScript SDK.
  - Install it.
  - Restart Visual Studio.

BASIC TYPES

- **Primitive Types:** number, string, boolean, null, undefined, symbol, bigint.

```
let age: number = 25;
let name: string = "John";
let isActive: boolean = true;
```

- **Arrays:** `let numbers: number[] = [1, 2, 3];`
- **Tuples:** `let person: [string, number] = ["John", 25];`
- **Enums:** Define a set of named constants:

```
enum Color {
  Red,
  Green,
  Blue
}
let color: Color = Color.Green;
```

FUNCTIONS

FUNCTIONS

- **Function Types:** Define parameter and return types:

```
function add(a: number, b: number): number {
  return a + b;
}
```

- **Default parameters:**

```
function greet(name: string, age: number = 25): string {
  return `${name} is ${age} years old`;
}
```

FUNCTIONS: OPTIONAL PARAMETERS

- In TypeScript, you can define optional parameters in functions using the **?** syntaThe age parameter is optional because of the ? after its name.
- The function works with or without the optional parameter.

```
function greet(name: string, age?: number): string {
  return age ? `${name} is ${age} years old` : `Hello, ${name}`;
}
```

OR

```
function greet(name: string, age?: number): string {
    if (age !== undefined) {
        return `Hello, ${name}! You are ${age} years old.`;
    } else {
        return `Hello, ${name}!`;
    }
}

console.log(greet("Alice")); // "Hello, Alice!"
console.log(greet("Bob", 30)); // "Hello, Bob! You are 30 years old."
```

FUNCTIONS: REST PARAMETERS

- In TypeScript, you can use rest parameters to accept multiple arguments as an array. Rest parameters are denoted using ... before the parameter name.

```
function sumNumbers(...numbers: number[]): number {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sumNumbers(1, 2, 3, 4, 5)); // Output: 15
console.log(sumNumbers(10, 20));       // Output: 30
console.log(sumNumbers());             // Output: 0
```

- The ...numbers: number[] parameter allows passing multiple numbers. The reduce function sums up all the numbers. The function works with any number of arguments (including none).

INTERFACES & CLASSES

INTERFACES

- **Define the structure of an object:**

```
interface Person {
  name: string;
  age: number;
  greet(): void;
}

let john: Person = {
  name: "John",
  age: 25,
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
};
```

USING AN INTERFACE

- The Person interface ensures that any object assigned to it has the required properties.The introduce function expects a Person object.

```
// Defining an interface
interface Person {
    name: string;
    age: number;
    greet(): string;
}
// Using the interface in a function
function introduce(person: Person): string {
    return `Hello, my name is ${person.name} and I am ${person.age} years old.`;
}
// Creating an object that follows the interface
const user: Person = {
    name: "Alice",
    age: 25,
    greet() {
        return "Hi!";
    }
};
console.log(introduce(user)); // Output: Hello, my name is Alice and I am 25 years old.
```

CLASSES

- **Define a class with properties and methods:**

```
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  speak(): void {
    console.log(`${this.name} makes a noise.`);
  }
}

let dog = new Animal("Dog");
dog.speak();
```

INHERITANCE

- **Extend a class:**

```
class Dog extends Animal {
  breed: string;

  constructor(name: string, breed: string) {
    super(name);
    this.breed = breed;
  }

  speak(): void {
    console.log(`${this.name} barks.`);
  }
}

let dog = new Dog("Buddy", "Golden Retriever");
dog.speak();
```

ADVANCED TYPES

UNION TYPES

- Union Types (|) a variable to have multiple possible types.
- When a variable can be of multiple types but only one at a time.

```
function display(value: string | number): string {
    if (typeof value === "number") {
        return `The number is ${value}`;
    } else {
        return `The string is "${value}"`;
    }
}

console.log(display(42));       // Output: The number is 42
console.log(display("Hello"));  // Output: The string is "Hello"
```

INTERSECTION TYPES

- Intersection Types (&) combine multiple types into one, meaning an object must have all properties from the combined types.
- When an object needs to have properties from multiple types.

```
interface A {
  a: string;
}
interface B {
  b: number;
}
let obj: A & B = { a: "Hello", b: 42 };
```

INTERSECTION TYPES

```
interface Employee {
    name: string;
    id: number;
}
interface Manager {
    department: string;
}
type ManagerEmployee = Employee & Manager;

const john: ManagerEmployee = {
    name: "John Doe",
    id: 101,
    department: "HR"
};

console.log(john);
// Output: { name: 'John Doe', id: 101, department: 'HR' }
```

CUSTOM TYPES

- Custom Types (Type Aliases) allows you to define a reusable type.
- When you need a specific structure but don't want to use an interface.

```
type User = {
    id: number;
    name: string;
    isAdmin: boolean;
};

function getUserInfo(user: User): string {
    return `User ${user.name} has ID ${user.id} and is ${user.isAdmin ? "an admin" : "a regular user"}.`;
}

const user1: User = { id: 1, name: "Alice", isAdmin: true };

console.log(getUserInfo(user1));
// Output: User Alice has ID 1 and is an admin.
```

GENERIC TYPES

- Generics allow you to create flexible, reusable components that work with different data types.
- Example : A function that works with any data type.

```
function identity<T>(value: T): T {
    return value;
}

console.log(identity<string>("Hello")); // Output: Hello
console.log(identity<number>(42));      // Output: 42
```

GENERIC TYPES

- Example : Generic Interface
- Fetching data where the type of data can vary.

```
interface ApiResponse<T> {
    success: boolean;
    data: T;
}

const userResponse: ApiResponse<User> = {
    success: true,
    data: { id: 2, name: "Bob", isAdmin: false }
};

console.log(userResponse);
// Output: { success: true, data: { id: 2, name: "Bob", isAdmin: false } }
```

MODULES AND NAMESPACES

MODULES

- **Modules** are a way to split code into separate files, each of which can export and import functionality.
  - **Export**: Expose variables, functions, classes, or interfaces from a module.
  - **Import**: Use exported functionality from another module.
- Use modules for organizing code across multiple files

```
// math.ts (module)
export function add(a: number, b: number): number {
    return a + b;
}
export function subtract(a: number, b: number): number {
    return a - b;
}
// app.ts (importing module)
import { add, subtract } from './math';

console.log(add(5, 3)); // Output: 8
console.log(subtract(5, 3)); // Output: 2
```

NAMESPACES

- Namespaces are a way to group related code under a single name. They are primarily used to avoid naming collisions in the global scope.
  - **Namespace Declaration:** Use the namespace keyword to define a namespace.
  - **Export**: Expose functionality within the namespace.

```
namespace Math {
  export function add(a: number, b: number): number {
    return a + b;
  }
}
console.log(Math.add(2, 3));
```

DECORATORS

DECORATORS

- In TypeScript, decorators are a special kind of declaration that can be attached to classes, methods, properties, or parameters to modify their behavior.
- Decorators are widely used in frameworks like Angular and NestJS to add metadata, enable dependency injection, or extend functionality.
- Decorators are an experimental feature in TypeScript, so you need to enable the experimentalDecorators option in your tsconfig.json file:

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

- **Types of Decorators**
  1. **Class Decorators**: Applied to a class constructor.
  2. **Method Decorators**: Applied to methods within a class.
  3. **Property Decorators**: Applied to properties of a class.
  4. **Parameter Decorators**: Applied to parameters of a method or constructor.

CLASS DECORATOR

- **Class Decorators:** Modify or extend a class definition. Itis applied to the constructor of a class.

```
function LogClass(target: Function) {
    console.log(`Class ${target.name} is created.`);
}
@LogClass
class Person {
    constructor(public name: string) {}
}
const person = new Person("Alice");
// Output: Class Person is created.
```

CLASS DECORATOR

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}

@sealed
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

METHOD DECORATOR

- A method decorator is applied to a method of a class. It can be used to modify or replace the method.

```
function LogMethod(target: any, key: string, descriptor: PropertyDescriptor) {
    console.log(`Method ${key} is called.`);
}

class Calculator {
    @LogMethod
    add(a: number, b: number): number {
        return a + b;
    }
}

const calc = new Calculator();
calc.add(2, 3);    // Output: Method add is called.
}
```

PROPERTY DECORATOR

- A property decorator is applied to a property of a class. It can be used to modify or observe the property.

```
function DefaultValue(value: string) {
    return function (target: any, key: string) {
        target[key] = value;
    };
}

class Person {
    @DefaultValue("John Doe")
    name: string;
}

const person = new Person();
console.log(person.name);              // Output: John Doe}
```

PARAMETER DECORATOR

- A parameter decorator is applied to a parameter of a method or constructor. It is often used for metadata reflection.

```
function LogParameter(target: any, key: string, index: number) {
    console.log(`Parameter ${index} of method ${key} is decorated.`);
}

class Greeter {
    greet(@LogParameter name: string) {
        console.log(`Hello, ${name}!`);
    }
}

const greeter = new Greeter();
greeter.greet("Alice");
// Output: Parameter 0 of method greet is decorated.
// Output: Hello, Alice!
```

TYPESCRIPT VS JAVASCRIPT

TYPESCRIPT VS JAVASCRIPT

- The main difference between JavaScript and TypeScript lies in their type systems and how they are used in development.
- **Type System**
  - *JavaScript:*
    - Dynamically typed: Variables can hold values of any type, and types are determined at runtime.
  - *TypeScript:*
    - Statically typed: Supports optional type annotations and type checking at compile time. Adds a type system on top of JavaScript, allowing developers to define types for variables, functions, and objects.
- **Compilation**
  - *JavaScript:*
    - Interpreted language: Runs directly in browsers or Node.js without the need for compilation.
  - *TypeScript:*
    - Requires compilation: TypeScript code is transpiled into JavaScript before it can run in browsers or Node.js.
    - The TypeScript compiler (tsc) checks for type errors and converts .ts files into .js files.

TYPESCRIPT VS JAVASCRIPT

- **Compatibility**
  - *JavaScript*:
    - Natively supported by all browsers and JavaScript runtimes.
  - *TypeScript*:
    - A superset of JavaScript: Any valid JavaScript code is also valid TypeScript code.
    - Requires a build step to convert TypeScript into JavaScript for execution.
- **Use Cases**
  - *JavaScript:*
    - Ideal for small projects, quick prototyping, or when type safety is not a priority.
  - *TypeScript:*
    - Better suited for large-scale applications where type safety, maintainability, and scalability are important.
    - Commonly used in enterprise-level projects and frameworks like Angular.

EXAMPLES

BUILDING A SIMPLE CALCULATOR

- **Example 1: Building a Simple Calculator**
  - Create a calculator class with methods for addition, subtraction, multiplication, and division.
- **Example 2: Creating a To-Do List**
  - Build a to-do list application using TypeScript classes and interfaces.
- **Final Project**
  - •Build a small TypeScript application with the following features:
  - 1. A Person class with properties and methods.
  - 2. A Calculator class with generic methods.
  - 3. A to-do list application using interfaces and classes.

RESOURCES

- **TypeScript Documentation**
  https://www.typescriptlang.org/docs/
- **TypeScript Tutorial (YouTube)**
  https://www.youtube.com/results?search_query=typescript+tutorial