**TYPESCRIPT**

TypeScript is Typed JavaScript. TypeScript adds types to JavaScript to help you speed up the development by catching errors before you even run the JavaScript code.

TypeScript being a "Syntactic Superset" means that it shares the same base syntax as JavaScript, but adds something to it.

**Why typescript?**

* JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.
* In JavaScript, function parameters and variables don't have any information! So developers need to look at documentation, or guess based on the implementation.
* 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.
* For example, TypeScript will report an error when passing a string into a function that expects a number. JavaScript will not.
* TypeScript uses compile time type checking. Which means it checks if the specified types match **before** running the code, not **while** running the code.

**Install typescript**

* For using typescript, we should have node js installed.
* Typescript command: **npm install -g typescript**

To run the typescript file,

**tsc filename.ts** : transpile a js file with any type correction if exists.

**node filename.js**: to run js code.

**tsc filename - - watch**: to automatically recompile code if there is any change

# **TypeScript Simple Types**

There are three main primitives in JavaScript and TypeScript.

let isBeginner: boolean = true;

 let total: number=0;

 let name: string ='Vaishnavi';

 let sentence: string = `My name is ${name}

 I am a beginner in Typescript `;

 console.log(sentence);

* + boolean - true or false values
  + number - whole numbers and floating point values
  + string - text values like "TypeScript"

**Type annotation**

TypeScript uses type annotations to explicitly specify types for identifiers such variables, functions, objects, etc.

TypeScript uses the syntax : type after an identifier as the type annotation, where type can be any valid type.

let variableName: type;

let variableName: type = value;

const constantName: type = value;

## Type Assignment

When creating a variable, there are two main ways TypeScript assigns a type:

* Explicit and Implicit

**Explicit**

Writing out the type

let message: string='learning typescript';

**Implicit**

TypeScript will "guess" the type, based on the assigned value

let message='learning typescript';

Having TypeScript "guess" the type of a value is called **infer**.

**Typescript special types**

## Type: any

any is a type that disables type checking and effectively allows all types to be used

let anytype:any;

  anytype=20;

  anytype=true;

## Type: unknown

unknown is a similar, but safer alternative to any

## Type: never

never effectively throws an error whenever it is defined. (rarely used)

## Type: undefined & null

undefined and null are types that refer to the JavaScript primitives undefined and null respectively.

let n:null=null;

let u:undefined=undefined;

**Typescript Arrays**

Syntax:

**const names: string[]=[];**

  let list1: number[] = [1,2,3];

  let list2: Array<number>=[1,2,3];

## Readonly

The readonly keyword can prevent arrays from being changed

const names: readonly string[] = [“John”];  
names.push("Jack");

# **TypeScript Tuples**

## Typed Arrays:

A **tuple** is a typed [array](https://www.w3schools.com/js/js_arrays.asp) with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

 let person1: [string,number]=['Apple',200];

Eg:

let employee: [number, string ,string]

employee = [34,'Vaishnavi', 'Bosch']

# **Object type**

*//object type*

  const car: { type: string, model: string, year: number } = {

  type: "Toyota",

  model: "Corolla",

  year: 2009

  };

console.log(car);

# **TypeScript Enums**

An **enum** is a special "class" that represents a group of constants (unchangeable variables).

Enums come in two flavors string and numeric

let w:Week = Week.tuesday;

  console.log(w);

  enum StatusCodes {

    NotFound = 404,

    Success = 200,

    Accepted = 202,

    BadRequest = 400

  }

  console.log(StatusCodes.NotFound);

  console.log(StatusCodes.Success);

# **TypeScript Union Types**

**Union types** are used when a value can be more than a single type.

Such as when a property would be string or number

let multitype:number | boolean;

  multitype=20;

  multitype=true;

# **Typescript Functions**

TypeScript has a specific syntax for typing function parameters and return values.

function add(num1: number,num2?:number):number {

*if*(num2)

*return* num1+ num2;

*else*

*return* num1;

  }

  console.log(add(5,10));

  add(5);

The number here represents that the function returns a number. If no return type is defined, TypeScript will attempt to infer it through the types of the variables or expressions returned.

## Void Return Type

The type void can be used to indicate a function doesn't return any value.

function printHello(): void {

    console.log('Hello!');

  }

## Optional Parameters

By default TypeScript will assume all parameters are required, but they can be explicitly marked as optional.

*// the `?` operator here marks parameter `c` as optional*

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

*return* a + b + (c || 0);

}

**Default parameters**

For parameters with default values, the default value goes after the type annotation

function pow(value: number, exponent: number = 10) {

*return* value \*\* exponent;

  }

    console.log(pow(10))

# **TypeScript Casting**

# There are times when working with types where it's necessary to override the type of a variable, such as when incorrect types are provided by a library.

Casting is the process of overriding a type.

## Casting with as

The as keyword, which will directly change the type of the given variable.

 let b: unknown = 'hello';

console.log((b *as* string).length);

## Casting with <>

Using <> works the same as casting with as

let b: unknown = 'hello';

  console.log((<string>b).length);

**Typescript Classes**

TypeScript adds types and visibility modifiers to JavaScript classes

Members visibility

There are three main visibility modifiers in TypeScript.

* public - (default) allows access to the class member from anywhere
* private - only allows access to the class member from within the class
* protected - allows access to the class member from itself and any classes that inherit it.

eg:

class Person {

    private name: string;

    public constructor(name: string) {

*this*.name = name;

    }

    public getName(): string {

*return* *this*.name;

    }

  }

  const person = new Person("Issac Newton");

  console.log(person.getName());

## Inheritance: Implements

A class can reuse the properties and methods of another class. This is called inheritance in TypeScript.

The class which inherits properties and methods is called the **child class**. And the class whose properties and methods are inherited is known as the **parent class**.

Interfaces can be used to define the type a class must follow through the implements keyword. i.e., it specifies a set of methods that the class has to implement.

interface Shape {

    getArea: () => number;

  }

  class Rectangle implements Shape {

    public constructor(protected readonly width: number, protected readonly height: number) {}

    public getArea(): number {

*return* *this*.width \* *this*.height;

    }

  }

**Static property**

A static property is shared among all instances of a class. It is declared with static keyword. To access a static property, use className.propertyName syntax.

class Employee {

    static headcount: number = 0;

    constructor(

        private firstName: string,

        private lastName: string,

        private jobTitle: string) {

        Employee.headcount++;

    }

}

Here,headcount is a static property that initialized to zero. Its value is increased by 1 whenever a new object is created.

**Static method**

Similar to the static property, a static method is also shared across instances of the class. To declare a static method, you use the static keyword before the method name.

class Employee {

    private static headcount: number = 0;

    constructor(

        private firstName: string,

        private lastName: string,

        private jobTitle: string) {

        Employee.headcount++;

    }

    public static getHeadcount() {

*return* Employee.headcount;

    }

}

1. Change the access modifier of the headcount static property from public to private so that its value cannot be changed outside of the class without creating a new Employee object.
2. add the getHeadcount() static method that returns the value of the headcount static property.

**Utility types**

TypeScript comes with a large number of types that can help with some common type manipulation, usually referred to as utility types

## Partial

## Partial changes all the properties in an object to be optional.

## Required

Required changes all the properties in an object to be required.

interface Car {

    make: string;

    model: string;

    mileage?: number;

  }

  let myCar: Required<Car> = {

    make: 'Ford',

    model: 'Focus',

    mileage: 12000 *// `Required` forces mileage to be defined*

  };

  console.log(myCar);

## Record

Record is a shortcut to defining an object type with a specific key type and value type

*//record*

  const nameAgeMap: Record<string, number> = {

    'boy': 7,

    'girl': 3

  };

  console.log(nameAgeMap);

## Omit

Omit removes keys from an object type.

*//omit*

  interface Person {

    Fulname: string;

    age: number;

    location?: string;

  }

  const b: Omit<Person, 'age' | 'location'> = {

    Fulname: 'Thomas'

*// `Omit` has removed age and location from the type and they can't be defined here*

  };

  console.log(b);

## Pick

Pick removes all but the specified keys from an object type

*//pick*

  interface Person {

    Fulname: string;

    age: number;

    location?: string;

  }

  const don: Pick<Person, 'Fulname'> = {

    Fulname: 'Don'

*// `Pick` has only kept name, so age and location were removed from the type and they can't be defined here*

  };

  console.log(don);

## Exclude

Exclude removes types from a union

type Primitive = string | number | boolean;

  const value: Exclude<Primitive, string> = true;   *// a string cannot be used here since Exclude removed it from the type.*

  console.log(typeof value);

## ReturnType

ReturnType extracts the return type of a function type.

## Parameters

Parameters extracts the parameter types of a function type as an array.

**ReadlineSync**

Synchronous Readline allows script to have a conversation with the user via a console

**Synchronous readline**

npm i - -save readline-sync

npm install - -save @types/readline-sync

refer: <https://www.npmjs.com/package/readline-sync>

*import* { question } *from* "readline-sync";

type Operator= '+'|'-'|'\*'|'/';

function main() : void

 {

    const firststr: string = question('Enter first number:\n');

    const operator: string = question('Enter operator:\n');

    const secondstr: string = question('Enter second number:\n');

    const validInput:boolean=isNumber(firststr)&&isOperator(operator)&&isNumber(secondstr);

*if*(validInput)

    {

        const firstNum:number = parseInt(firststr);

        const secondNum:number = parseInt(secondstr);

        const result = calculate(firstNum,operator *as* Operator,secondNum);

        console.log(result);

    }

*else*{

        console.log('\nInvalid input\n');

        main()

    }

 }

function calculate(firstNum: number,operator:Operator,secondNum:number)

{

*switch*(operator)

    {

*case* '+':

*return* firstNum + secondNum;

*case* '-':

*return* firstNum - secondNum;

*case* '\*':

*return* firstNum \* secondNum;

*case* '/':

*return* firstNum / secondNum;

    }

}

function isOperator(operator:string):boolean{

*switch*(operator)

    {

*case* '+':

*case* '-':

*case* '\*':

*case* '/':

*return* true;

*default*:

*return* false;

    }

}

function isNumber(str: string): boolean

{

    const maybeNum = parseInt(str) *// parses a string and returns the first integer*

    const isNum:boolean = !isNaN(maybeNum); *// flips the output*

*return* isNum;

}

main();

Output:

