# TYPESCRIPT INTERVIEW QUESTIONS AND ANSWERS IN HINDI

# 1. What is TypeScript?

TypeScript is an object-oriented programming language that was designed by Microsoft in the year 2012. This language is considered as the superset of JavaScript. TypeScript extends the functionality of JavaScript by adding data types, classes, modules, interface and other object-oriented features with type-checking. JavaScript is well suited for small-scale applications but TypeScript is used for large-scale applications.

TypeScript cannot run directly on the browser. It needs a compiler to compile the file and generate a JavaScript file, which can run directly on the browser. The TypeScript source file is in ".ts" extension. We can use any valid ".js" file by renaming it to ".ts" file. TypeScript uses TSC (TypeScript Compiler) which convert Typescript code (.ts file) to JavaScript (.js file).

## History of TypeScript

In 2010, Anders Hejlsberg, a core member of the development team of C# language, started working on TypeScript at Microsoft. The first version of TypeScript was released to the public in the month of 1st October 2012 and was labeled as version 0.8. Now, it is maintained by Microsoft under the Apache 2 license.

# 2. Features of TypeScript?

TypeScript has the following features:

- **Tuple:** A Tuple is like an array but in a tuple each element can have different data type. Tuple is useful when we want to store fixed number of elements with different data type.

**Example:**

```
// define our tuple
    let ourTuple: [number, boolean, string];

// initialize correctly
    ourTuple = [5, false, 'The Digital Oceans'];
```

- **Enums:** Enums is used to group constants ,Enums comes into two types **string** and **number.** Numeric enums are auto incremented, means if we assign the first value it will auto increment rest of the values.

**Example:** enum Direction {Up = 1,Down,Left,Right,}

- **Decorators:** A decorator is a design pattern in programming in which you wrap something to change its behavior.Decorators are a powerful feature in TypeScript that allows developers to modify or extend the behavior of classes, methods, accessors, and properties. They offer an elegant way to add functionality or modify the behavior of existing constructs without altering their original implementation.

- **Generics:** it is used create reusable components or functions that can handle multiple data types. Generics ensures that the program is flexible as well as scalable in the long term. Generics provides type safety without compromising the performance, or productivity.

- **Conditional Types:** it is useful when we have to make a decision about type of output based on its input type.

**Example:**

type UnwrapArray<T> = T extends (infer U)[] ? U : never;

conditional type that returns the type of the input (U) if it's an array, otherwise, it returns"never"

- **Type Aliases:** Type Aliases allow defining types with a custom name (an Alias).

**Example:**

```
type CarYear = number
type Car = {
  year: CarYear
}
const carYear: CarYear = 2001
```

- **Namespaces:** The namespace is a way which is used for **logical grouping** of functionalities. It encapsulates the features and objects that share common relationships. It allows us to organize our code in a much cleaner way. TypeScript namespace removes the **naming collisions**.

# 3. Advantages of TypeScript?

TypeScript has the following advantages:

**Static Typing:** which allows us to specify and check the data type of your variable, function parameter, return type at compilation time. It helps us to catch errors early at compile time and also improve readability.

**OOP Features:** TypeScript supports object oriented features such as classes, inheritance, interface, polymorphism, abstraction etc, by using these features we can make our code reusable and scalable.

**Supports ECMAScript Features:** TypeScript supports ecmascript new features like arrow function, destructuring, template literals so we can use these features to keep our project up to date with latest syntax.

**Better IDE Support:** TypeScript can offer features like code navigation and autocompletion, providing accurate suggestions. All this helps you write maintainable code and results in a significant productivity boost.

# Q4. Difference between JavaScript and TypeScript?

| JavaScript | TypeScript |
|---|---|
| It is runtime language. | It is compile time language. |
| It does not support all features of object oriented. | It supports mostly features of object oriented. |
| It does not support strong data type checking. | It supports strong type checking. |
| It directly runs on the browser. | It does not run directly on the browser. |
| Netscape developed in 1995. | Anders Hejlsberg developed it in 2012. |
| Its source file is in .js extention. | Its source file is in .ts extenstion. |
| It is not suitable for large projects. | It is best suitable for large projects. |

# 5. Components of TypeScript?

TypeScript comprises three main components: Language, the TypeScript Compiler, and the TypeScript Language Service.

- **Language:** the syntax, keywords, and type annotations.
- **The TypeScript Compiler (TSC):** converts the instructions written in TypeScript to its JavaScript equivalent. It also performs the parsing, and type checking of our TypeScript code to JavaScript code.
- **The TypeScript Language Service:** an additional layer of editor-like applications, such as statement completion, signature help, code formatting, and colorization, among other things.

# 6. How to Install and Run TypeScript?

**To install TypeScript**, Enter the following command:

**npm install -g typescript**

**To Run TypeScript File,** Enter the following command:

**tsc filename**

**tsc hello.ts**

after running the above command it will create .js file of the ts

now run the  command

**node filename.js**
**node hello.js**

# 7. What is tsconfig.json file in TypeScript?

TypeScript compiler uses **tsconfig.json** to get configuration options for generating JavaScript code from TypeScript source-code. When you compile TypeScript code, compiler searches for configurations located in **tsconfig.json**.

**How to Create tsconfig.json file:**

tsc --init

This command will create **tsconfig.json** file in your project's root folder.

# 8. How to compile TypeScript file with real time changes?

To compile TypeScript file with real time changes , we write the following command:

**Syntax:**

tsc filename --watch

**Example:**

tsc hello.ts --watch

# 9. Can we combine multiple .ts file into single .js file?

Yes, we can combine multiple files. While compiling, we need to add –outFILE [OutputJSFileName] option.

tsc --outFile main.js file1.ts file2.ts file3.ts

This will compile all 3 “.ts” file and output into a single “main.js” file.

tsc --outFile file1.ts file2.ts file3.ts

If you don’t provide an output file name, file2.ts and file3.ts will be compiled and the output will be placed in file1.ts. So now your file1.ts contains JavaScript code.

# 10. List out the built-in Data Types of TypeScript?

There are following built-in Data Types, which are following:

**Number:** it is used to store number type data such as integer, float.

**Example:**

```
let a:number=1;
```

**String:** It is used to store string data in a variable.

**Example:**

```
let name1:string=‘The Digital Oceans';
```

**Boolean:** it is used to store true or false data.

**Example:**

```
let a:boolean=true;
let b:boolean=false;
```

**Void:** It is used with function when function does not return any value.

**Null:** It means nothing or no value. It is similar to the void type but we have to define it explicitly.

**Undefined:** The Undefined type denotes all uninitialized variables.

# 11. What is any type and when to use it?

The **any type** in TypeScript is a generic type used when a variable's type is unknown or when the variable's type hasn't yet been defined.

The any type is useful when converting existing JavaScript to TypeScript as it allows one to gradually opt-in and opt-out of type checking during compilation.

**Example:**

```
let coupon: any;
coupon = 26;
coupon = 'DEAL26';
coupon = true;
```

# 12. What is unknown type and when to use it?

TypeScript 3.0 introduced a **unknown** type which is the type-safe counterpart of the any type. Unknown type means that the type of variable is not known. We can assign anything to an unknown variable, but the unknown isn't assignable to any other types except to unknown and any. To use an unknown type, we need to first assert its type or narrow it to a more specific type.

You cannot perfom any operation on unknown type unless you perform a type check or type assertion.

**Example:**

```
let data: unknown;

data = true;
data = 10;
```

You can assign unknown to a variable of type unknown and any.

```
let data: unknown;

let data1:unknown = value; // OK
let data2:any = value;     // OK
```

But cannot assign unknown to any other types.

```
let data: unknown;

let data1: boolean = value;  // Error
let data2: number = value;   // Error
let data3: string = value;   // Error
```

**Type Assertion**

You can use the Type Assertion on an unknown type to let the compiler know the correct type.

```
let data: unknown;
let data2: string = data as string;     // OK
```

# 13. Explain Tuples in TypeScript?

TypeScript introduced a new data type called Tuple. Tuple can contain two values of different data types. Tuple stores a collection of values of different data types in a single variable.

**Example:**

var employee: [number, string] = [1, “Ram"];

You can declare an array of tuple also.

**Example:**

var employee: [number, string][];

employee = [[1, “Ram"], [2, “Shyam"], [3, “Sumit"]];

# 14. Explain Enum in TypeScript?

Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use enum. This is now available in TypeScript too.

In simple words, enum allow us to declare a set of named constants i.e. a collection of related values that can be numeric or string values.

There are three types of enums:

- **Numeric enum**
- **String enum**
- **Heterogeneous enum**

**Numeric Enums**

Numeric enums are **number-based** enums, which store values as numbers. It means we can assign the number to an instance of the enum.

**Example:**

```
enum Direction {
   Up = 1,
   Down,
   Left,
   Right,
}
```

## String Enums

String enums are a similar concept to numeric enums, except that the enum has some subtle runtime differences. In a string enum, each enum values are **constant initialized** with a string literal, or with another string enum member rather than numeric values.

String enums do not have **auto-incrementing** behavior. The benefits of using this enum is that string enums provides better **readability**.

**Example:**

```
enum AppStatus {
   ACTIVE = 'ACT',
   INACTIVE = 'INACT',
   ONHOLD = 'HLD',
   ONSTOP = 'STOP'
}
```

**Heterogeneous Enums**

Heterogeneous enums are enums that contain both string and numeric values.

**Example:**

```
enum Status {
 Active = 'ACTIVE',
Deactivate = 1,
Pending }
```

## 15. What are the object oriented Features Supported by TypeScript?

Object Oriented Features of TypeScript are following:

- Class
- Object
- Inheritance
- Encapsulation
- Polymorphism
- Abstraction

# 16. What are Modules in TypeScript?

The TypeScript code we write is in the global scope by default. If we have multiple files in a project, the variables, functions, etc. written in one file are accessible in all the other files.

**For example**, consider the following TypeScript files: file1.ts and file2.ts

**File1.ts**

```
var data : string = “The Digital Oceans”
```

**File2.ts**

```
console.log(data); //Prints Hello World!
```

The above variable data, declared in file1.ts is accessible in file2.ts as well. Not only it is accessible but also it is open to modifications. Anybody can easily override variables declared in the global scope without even knowing they are doing so! This is a dangerous space as it can lead to conflicts/errors in the code.

TypeScript provides modules and namespaces in order to prevent the default global scope of the code and also to organize and maintain a large code base.

Modules are a way to create a local scope in the file. So, all variables, classes, functions, etc. that are declared in a module are not accessible outside the module. A module can be created using the keyword export and a module can be used in another module using the keyword import.

## Example:

### addition.ts

```
export class Addition{
   constructor(private x?: number, private y?: number){
   }
   Sum(){
      console.log("SUM: " +(this.x + this.y));
   }
}
```

### app.ts

```
import {Addition} from './addition';

let addObject = new Addition(10, 20);

addObject.Sum();
```

# 17. What are Interfaces in TypeScript?

An Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface. We cannot instantiate the interface, but it can be referenced by the class object that implements it.

The TypeScript compiler does not convert interface to JavaScript. It uses interface for type checking. This is also known as "duck typing" or "structural subtyping“ whether the object has a specific structure or not.

**Example:**

```
interface Employee {
empCode: number;
empName: string;
 getSalary: (number) => number; // arrow function
getManagerName(number): string;
 }
```

# Interface as Function Type

TypeScript interface is also used to define a type of a function. This ensures the function signature.

**Example:**

```
interface KeyValueProcessor {
(key: number, value: string): void;
};

function addKeyValue(key:number, value:string):void
{
console.log('addKeyValue: key = ' + key + ', value = ' + value)
}

let kvp: KeyValueProcessor = addKeyValue;
 kvp(1, ‘The Digital Oceans');
```

**Output:** addKeyValue: key = 1, value = The Digital Oceans

# Interface for Array Type

An interface can also define the type of an array where you can define the type of index as well as values.

**Example:**

```
interface NumbersList {
[index:number]:number
}

let numberArr: NumbersList = [1, 2, 3];
numberArr[0];
numberArr[1];
```

# Extending Interfaces

Interfaces can extend one or more interfaces. This makes writing interfaces flexible and reusable.

**Example:**

```
interface Person {
name: string;
gender: string;
}

interface Employee extends Person
{ empCode: number; }

let empObj:Employee = {
empCode:1,
name:“Ravi",
gender:"Male" }
```

# 18. What are the access modifiers supported by TypeScript?

Access Modifiers is used to control the visibility of  the class members such as variable or function.

There are three types of access modifiers in TypeScript:

- **Public :** By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.

- **private :** The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

- **Protected:** The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

# 19. Explain Decorators in TypeScript?

Decorators are a powerful feature in TypeScript that allows developers to modify or extend the behavior of classes, methods, accessors, and properties. They offer an elegant way to add functionality or modify the behavior of existing constructs without altering their original implementation.

Decorators have a rich history in the TypeScript and JavaScript ecosystems. The concept of decorators was inspired by Python and other programming languages that use similar constructs to modify or extend the behavior of classes, methods, and properties.

Decorators are currently a stage 2 proposal in Javascript, but they are available as an experimental feature in Typescript. To use decorators, we have to set the experimentalDecorators compiler option in the tsconfig.json file.

Decorators are simply functions that are prefixed **@expression** symbol, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

## Decorator Factories

Since decorators are functions, we might need to pass in some additional options so as to customize how the decorator works. To do this, we make use of Decorator Factories. A *Decorator Factory* is simply a function that returns a function. This returned function would then implement the decorator.

## Syntax:

```
const decoratorFactory = (value: string) =>
{
// the decorator factory returns a decorator function
return (target: any) => {
// the returned decorator uses 'target' and 'value'
}
}
```

## Example: Class Decorator:

```
const addAgeToPerson = <T extends { new (...args: any[]): {} }>(
originalConstructor: T
) => {
return class extends originalConstructor {
 age: number;
constructor(...args: any[]) {
super();
this.age = 28;
}
}; };
@addAgeToPerson
class Person {}
const person = new Person();
console.log(person.age); // 28
```

# 20. Explain Generics in TypeScript?

Generics offer a way to create reusable components. Generics provide a way to make components work with any data type and not restrict to one data type. So, components can be called or used with a variety of data types. Generics in TypeScript is almost similar to C# generics.

Generics provides type safety without compromising the performance, or productivity. TypeScript uses generics with the type variable which denotes types.

Generics uses the type variable <T>, a special kind of variable that denotes types. The type variable remembers the type that the user provides and works with that particular type only. This is called preserving the type information.

**Example:**

```
function checkdatatype<T>(arg: T): T {
    return arg;
}
let output1 =  checkdatatype<string>("myString");
let output2 =  checkdatatype<number>( 100 );
```

# 21. Explain Namespaces in TypeScript?

The namespace is used for logical grouping of functionalities. A namespace can include interfaces, classes, functions and variables to support a single or a group of related functionalities.
A namespace can be created using the namespace keyword followed by the namespace name. All the interfaces, classes etc. can be defined in the curly brackets { }.

**Example:**
NameSpace file: **student**

```
namespace student{
    export function AnualFee(feeAmount: number, term: number){
    return feeAmount * term;
    }
}
```

Main File: **app.ts**

```
/// <reference path = "./student.ts" />

let TotalFee = student.AnualFee(1500, 4);

console.log("Output: " +TotalFee);
```

# 22. Explain Type Inference in TypeScript?

TypeScript is a typed language. However, it is not mandatory to specify the type of a variable. TypeScript infers types of variables when there is no explicit information available in the form of type annotations.

**Example:**

```
var a = “the digital oceans";
var b = 123;
```

# 23. Explain the use of TypeScript Definition Manager?

TypeScript Definition Manager (TSD) is a package manager used to search and install TypeScript definition files directly from the community-driven Definitely Typed repository.

**Steps to include TypeScript Definition File:**

- First, you have to install TSD.

  ```
  npm install tsd -g
  ```

- Next, in TypeScript directory, create a new TypeScript project by running:

  ```
  tsd init
  ```

- Then install the definition file for jQuery.

  ```
  tsd query jquery --action install
  ```

- Now, include the definition file by updating the TypeScript file to point to the jQuery definition.

  ```
  /// <reference path="typings/jquery/jquery.d.ts" />
  $(document).ready(function() { //To Do
  });
  ```

- Finally, compile again. This time js file will be generated without any error. Hence, the need for TSD helps us to get the type definition file for the required framework.

# 24. DOM Manipulation in TypeScript?

In a TypeScript we can do DOM manipulation like in JavaScript , but there is some difference in syntax, let's see the examples:

**Example: Select element by ID**

```
var btn=<HTMLElement>document.getElementById('btn')
```

**Example: Select element by Class**

```
let smallimg: HTMLCollectionOf<Element> = document.getElementsByClassName('smallimg');
```