***Typescript Basics***

**1- Types**

a) Primitive Types: 1) number 2) boolean 3) void 4) undefined 5) string 6) symbol 7) null

b) Object Types: 1) functions 2) arrays 3) classes 4) objects

*Note*: there is another type also: Date.

2- Type Annotations vs Type Inference (for variables)

a) Type Annotations:- Code we add to tell typescript what type of value a variable will refer to (developers tell the type)

b) Type Inference:- Typescript tries to figure out what type of value a variable refers to (typescript guesses the type)

3- When to use Type Annotations:- (for variables)

a) When a function returns the 'any' type and we need to clarify the value; e.g:- JSON.parse() returns 'any' type

b) When we declare a variable on one line then initialize it later;

c) When we want a variable to have a type that can't be infered. e.g:- we have assigned a Type to a variable and

then want to reassign with a different type on the basis of some condition. We fix this by

const numberAboveZero = number | boolean = false;

4- Type Annotations vs Type Inference (for functions)

a) Type Annotations:- Code we add to tell typescript what type of arguments a function will receive and what type of values it will return

b) Type Inference:- Typescript tries to figure out what type of value a function will return. (not the type of arguments it will take)

5- Type 'never':- e.g:- function throwError(message: string): never {throw new Error(message)}

6- Typed Arrays:- Arrays where each element is some consistent type of value

=> Why do we care?

a) TS can do type inference when extracting values from an array; e.g: const names = ['fareed', 'ahsan']; const name = names[0]; now here the typeof name will be string

b) TS can prevent us from adding incompatible values to the array;e.g: can't assign numbers to string array types

c) We can get help with 'map', 'forEach', 'reduce' functions; e.g:

d) Flexible - arrays can still contain multiple different types;

=> Where to use typed arrays?

a) Any time we need to represent a collection of records with some arbitrary sort order

7- Tuples:- Array-like structure where each element represents some property of a record

e.g: type Drink = [string, boolean, number]; const pepsi: Drink = ['brown', true, 50]; const sprite: Drink = ['red', false, 30]

we may use Tuples with CSV files

8- Interfaces:- interfaces + classes = How we get really strong code reuse in TS

definition:- Creates a new type, describing the property names and value types of an object

Notes:-

a) interface definition is not limited to primitive types but it can take other types as 'Date' or even 'functions'

General Strategy for reusable code in TS:-

a) Create functions that accept arguments that are typed with interfaces

b) Objects/classes can decide to 'implement' a given interface to work with a function

9- Classes:- Blueprint to create an object with some fields (values) and methods (functions) to represent a 'thing'

modifiers: a) public b) private c) protected

10- TypeGuards:- We use type guards any time we want to restore access to a set of properties in a union type.

1) typeof: used for primitive types as string, number, boolean, symbol. e.g; if(typeof 'asad' === 'string') {}

2) instanceof: every other value created with a constructor function. e.g; if([33, 45] instanceof Array) {}

11- Quick note on Union operator:- Union operator gives access to common properties/methods described on each type.

e.g; const field: number | string;

now this means field variable is allowed to access only common methods. like split() method.

We use type guards to get access to all the methods in a string | number

**Problems**

1. Map Problem
2. Sorting Problem
   1. Strings do not act like Numbers in JS with reference to
      1. Comparison; (we cannot compare X > a) and
      2. Swapping. (we cannot swap a character of string like: red[0] = ‘y’)
3. Data Problem

**Abstract Classes**

|  |
| --- |
| Can’t be used to create an object directly. e.g: we can’t write as const ahmed = new Person(); |
| Only used as a parent class |
| Can contain real implementation for some methods |
| The implemented methods can refer to other methods that don’t actually exist yet (we still have to provide names and types for the un-implemented methods) |
| Can make child classes promise to implement some other method |

**Difference between Interfaces and Inheritance/Abstract classes**

|  |  |
| --- | --- |
| **Interfaces** | **Inheritance/Abstract Classes** |
| Sets up a contract b/w different classes | Sets up a contract b/w different classes |
| Use when we have very different objects that we want to work together  e.g user and company are different from objects from map | Use when we are trying to build up a definition of an object  e.g sorter example |
| Promotes loose coupling | Strongly couples classes together |