# Typescript Fundamentals

## → Introduction

•) syntactic superset of Javascript. Additional layers on top of what is available in Javascript.

↓

opensource, maintained by Microsoft

↓

goal was to add types to Javascript

↓

using a compiler, typescript compiles out to readable Javascript.

•) comes in three parts:

i) programming language

ii) language server: piece of software behind our editing environment. Feeds the editor information that powers something like autocomplete.

iii) compiler

•) allows you as code author, to leave more "intent on the page". Whenever there are multiple interpretations of what are the constraints, and what's going on, and what was this designed to do, we're asking for trouble. So basically to increase certainty & confidence on a piece of code.

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

makes the code more clear, and would be alerted for any use of something like add that deviates from what the author originally intended.

- potential to move errors from runtime (where they impact users) to compile time.

→ Compiling a typescript program

- tsc --watch --preserveWatchOutput → every save doesn't clear console's output.

watch the source code, so that for every save, it will incrementally update the build result

↓

similar to webpack dev server or nodemon.

- tsconfig.json: contains all the instructions, and options to be passed to the compiler. 3 properties:

i) include: where to find the source code.

ii) compilerOptions:

a) outDir: where to put the output
(by default compiled files created alongside the source files)

b) target: language level of our build output.
(by default ES3, Internet Explorer 6)

•) All options can be passed to the tsc command as flags, but tsconfig is super helpful for complicated projects.

•) multi-target output: building for modern browsers and have a legacy browser to support

↓

compile to version for modern browsers
and using something like Babel to create a legacy build

•) post compilation, in the output directory, we would have found a `.d.ts` file. This is the declaration file. Only contains the type information.

↓

allows for people writing JS to compile the JS files and not really care about the type information.

↓

People writing typescript can reassemble them together.

↓

This is what allows for typescript based code to remain compatible with the general community.

') Types of modules :

i) standardized modules use 'export'

ii) common TS modules do not, and that is what node supports.

↓

So to use typescript based code in node, can specify the "module": "CommonJS"

↓

will change export → exports.addNumbers = addNumbers

→ Variables & Values

·) typescript is able to infer that age is a number, based on the fact that we've initializing it with a value as we are declaring it.

```
let age = 6;
```

↓

on trying to set age to a value that is incompatible with number, we get an error.

↓

In Typescript, variables are "born" with their types.

·) relatively easier to make types more specific than making them more general, once they are declared.

↓

important to make sure that when we define our variables, they have the types that we need them to have, and incase we need to generalize them, we go back to the declaration

·) const age = 6;

↳ age : 6 i.e. age is of type 6.

↓

literal type

↓

more specific kind of type

↓

i) variable cannot point to something else

ii) thing it currently points to cannot change.

(numbers are immutable)

↓

Typescript makes the most specific assumption w/o getting in anybody's way.

•) implicit 'any' and type annotations

'any' is the most flexible type in typescript

↓

we can think of 'any' as the normal way JS variables work, in that we can assign the variable to any type.

↓

to add more safety for variables that donot have initialization at declaration, we can use type annotation.

•) function type definitions

every code path should return the specified type

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

allows for error callout at the time

of declaration, and not invocation.

↓

Type Driven Development

•) any compromises with well typed code. It can hold anything and can also masquerade as anything. Its a wild card i.e. can accept anything, but can also present itself as anything.

→ collections : mutable value types : (objects, arrays & tuples)

•) In general, object types are defined by :

i) the name of the properties that are (or maybe) present

ii) the types of those properties

↓

```
{
    make: string
    model: string
    year: number
}
```

→ key-type pairs

↳ function printCar(car: {

```
    make: string
    model: string
    year: number
}) { };
```

•) optional properties : ? with the key

↓

signifies that the field may be there, but if it is then it is a number.

↓

chargeVoltage ? : number → number | undefined

type guard : a predicate for type in a control flow statement to consume variables safely. For example checking for undefined for an optional field.

↓

if ( typeof car.chargeVoltage !== "undefined" ) { }

↙ Typescript pushes us to use type guards to make the code variables safely consumable.

↘ type guard

*) reading error messages from the bottom

↓

difference b/w an optional property and a property that can take an undefined value

•) excess property checking → happen for literal objects that don't

have scope to be used anywhere.

↓

```
function printCar ( car : {
    make : string
    model : string
    year : number
    chargeVoltage ? : number
}) { }
```

extra property here causes an issue

```
printCar ( { } )
```

→ ↓ instead

```
printCar ( myCar )
```

↙

object with extra property is okay, since it is possible to be used somewhere else, other than the fn.

) index signatures

dictionaries : values of a consistent type are retrievable by arbitrary keys

↓

a phonebook for example

↓

to type such values, we need something called index signatures.

↳

```
const phoneBook : {
    [k : string] : {
        ... consistent type for value
    } | undefined
```

} = { }

makes consumption of the values safer, by prompting the use of type guard.

•) arrays : positional storage : string[]

↓

type to store values of

•) tuple : multi-element, ordered data structure, where position of each item has some special meaning or convention. This kind of structure is often called a tuple.

↓

For something like

```
let myCar = [2000, "Toyota", "Corolla"];
```

typescript infers it be an array of mixed type of arbitrary length, which is not exactly we have here, i.e. we want it to of fixed length & of the specified type.

↓

Typescript takes the former approach to allow for the more flexible use case, and wants to stay out of your way, but to provide as much safety as possible

↓

In this case, we need to define more explicitly the type of the tuple, i.e.

```
let myCar: [number, string, string] = [...];
```

- Limitations: limited support for enforcing tuple length constraints

→ Structural vs Nominal types

·) What is type checking?

has to do with answering a question around type equivalence as in, is the value that we send in as argument to a function is equivalent to what the function was designed to accept? This is type compatability or type equivalence.

↓

These can happen in situations like:

(i) function calls

(ii) assignment

(iii) return statement

·) static vs dynamic

type checking performed at compile time or runtime.

↓

typescript's type system is static

Javascript's type system is dynamic

·) Nominal vs Structural

typescript : structural ↗ care only about structure or shape

Nominal type systems are all about names → what is the

name of your class ?

↓

All type equivalence questions are based on the name of the associated class.