Introduction to TypeScript

Session-1

CLARUSWAY

WAY TO REINVENT YOURSELF

Table of Contents

TypeScript

Did you finish Typescript pre-class material?

Play Kahoot!

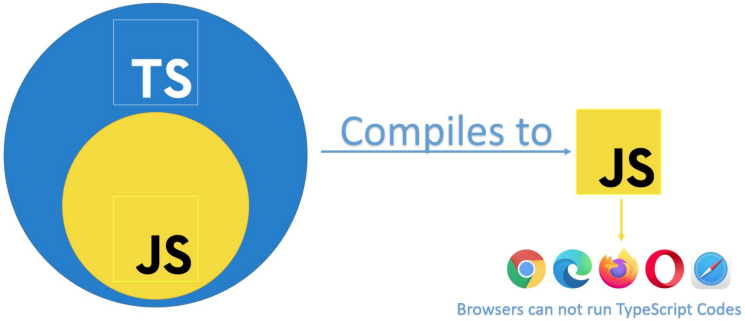
1

What is the Typescript?

TypeScript

A brief history of typescript,

- ***Anders Hejlsberg*** (the author of typescript) began working on typescript at Microsoft in 2010, and the first version of typescript was given to the public in 2012. (typescript 0.8).
- It is a superset of JavaScript.
  That is, it's JavaScript with a bunch of additional features.
- After you've written your code, it compiles into JavaScript. Your valid JS code is also valid TS code.



TypeScript

- Object oriented language
- Static type-checking
- Optional static typing
- DOM manipulation
- ES6 features

TypeScript Pros

Pros

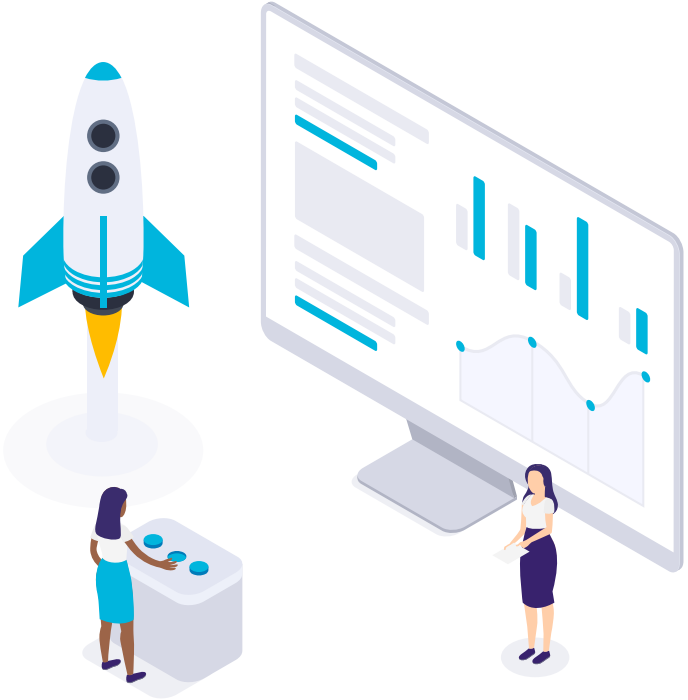
- Type Safety
- Better expressibility
- Opt in to types
- Rich IDE support
- Readability
- Power of OOPs concept
- Cross-browser & cross platform compatibility

Cons

- Additional built step required
- Learning curve
- Bloated Code
- Not true static type
- Transpiling

2

Setup Development Environment



CLARUSWAY
WAY TO REINVENT YOURSELF

TypeScript

- Install typescript using node.Js package manager (npm).
- Install the typescript plug-in in your IDE.

```
> npm install -g typescript
>tsc -v
 Version 4.x.x
```

tsconfig.json

- The presence of a **tsconfig.json** file in a directory indicates that the directory is the root of a TypeScript project.
- The tsconfig.json file specifies the root files and the compiler options required to compile the project.

  >tsc - -init

- tsc —init command creates a configuration file called tsconfig.json
- tsc command will generate .js files for all .ts files.

tsconfig.json

allowJs	Allow JavaScript files to be compiled. Default value is false.
alwaysStrict	Parse in strict mode and emit “use strict” for each source file. Default value is false.
target	Specify ECMAScript target version.
outDir	The location in which the transpiled files should be kept.
noEmitOnError	Disable emitting files if any type checking errors are reported.
noUnusedParameters	Raise an error when a function parameter isn't read.
removeComments	Disable emitting comments.
noImplicitAny	Enable error reporting for expressions and declarations with an implied 'any' type.
strictNullChecks	When type checking, take into account 'null' and 'undefined'.

3

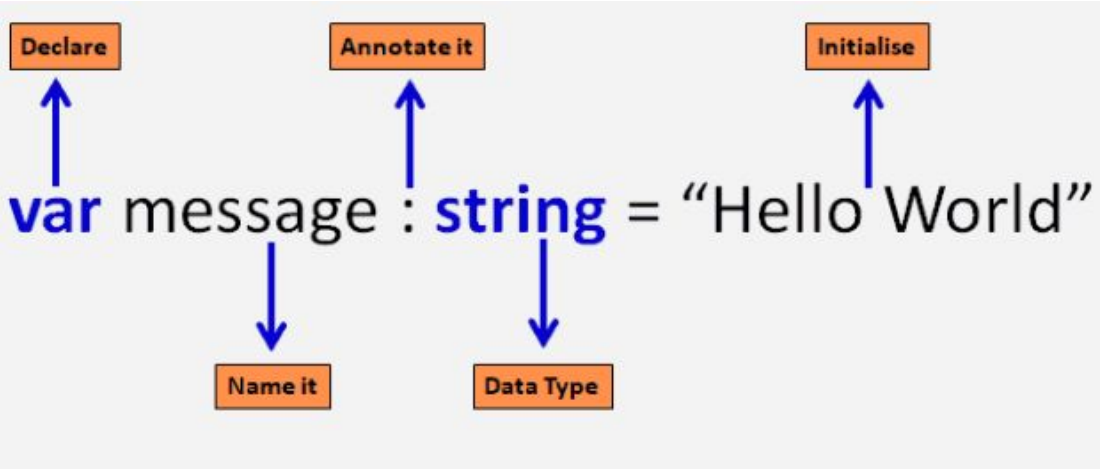
Type Annotation

Type Annotation

- Since TypeScript is a typed language , we can specify the type of variables , function parameters , and object properties.
- Type Annotation is declaration of type.
- Type annotation is not mandatory in TypeScript .
  Compiler can infer types of variable and help preventing errors.

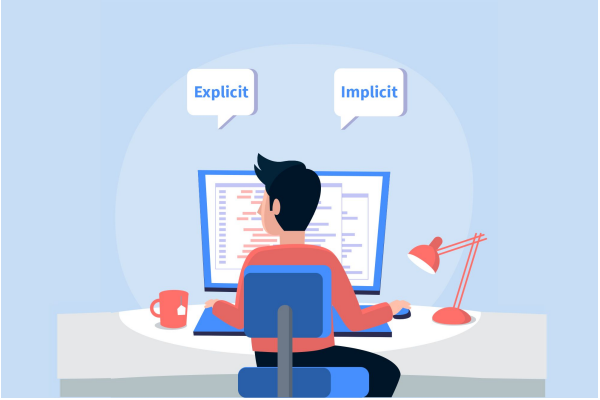
Type Annotation

- We annotate a variable by using a colon (:) followed by its type .



4

Types in TypeScript



Data Type - any

- We can not always have prior knowledge about the type of some variables, especially when there are user-entered values from third party libraries.
- In such cases, we need a provision that can deal with dynamic content.
- To do so, we label these values with the any type:

```
let looselyTyped: any = 4;

let arr: any[] = ['John', 212, true];
```

Data Type - any

Any means any of javaScript types. It means not using typeScript at all.

Example: Any

```
let something: any = "Hello World!";
something = 23;
something = true;
```

Example: Any type Array

```
let arr: any[] = ["John", 212, true];
arr.push("Smith");
console.log(arr); //Output: [ 'John', 212, true, 'Smith' ]
```

Data Type - unknown

- We may need to describe the type of variables that we do not know when we are writing an application.
- These values may come from dynamic content – e.g. from the user – or we may want to intentionally accept all values in our API.
- In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the unknown type.

```
let notSure: unknown = 4;

notSure = "maybe a string instead";
```

Data Type - unknown

Just like all types are assignable to any, all types are assignable to unknown. It is similar to any type.

You can assign anything to an unknown type:

```
let unknownVar: unknown;
unknownVar = false; // boolean
unknownVar = 15; // number
unknownVar = "Hello World"; // String
unknownVar = ["1" , "2" , "3" , "4" , "5"] // Array
unknownVar = { userName: 'admin' , password: '123x' }; // Object
unknownVar = null; // null
unknownVar = undefined; // undefined
```

But cannot assign unknown to any other types:

```
let value: unknown;

let newValue1: boolean = value; // Error
let newValue2: number = value; // Error
let newValue3: string value; // Error
let newValue4: object = value; // Error
let newValue5: any[] = value; // Error
let newValue6: Function = value; // Error
```

Data Type - void

- void is a little like the opposite of any: the absence of having any type at all.
- You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {

 console.log("This is my warning message");

}
```

Data Type - void

Similar to languages like java, void is used where there is no data type.

Example: void

```
function sayHi(): void {
    console.log('Hi!')
}

let speech: void = sayHi();
console.log(speech); //Output: undefined
```

Return type of functions that do not return any value.

Data Type - never

- Typescript introduced a new type never, which indicates the values that will never occur.

- The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

```
// Function returning never must not have a reachable end point

function error(message: string): never {

 throw new Error(message);

}
```

Data Type - never

Typescript introduced a new type never, which means the function will not be able to reach 'return' statement.

Example: never

```
function throwError(errorMsg: string): never {
            throw new Error(errorMsg);
}

function keepProcessing(): never {
            while (true) {
         console.log('I always does something and never ends.')
     }
}
```

DATA TYPE - Arrays

1. Square brackets. Similar to arrays in JavaScript.

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
```

2. Generic array type

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

3. initialize an array like shown below, but you will not get the advantage of typeScript.

```
let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];
```

DATA TYPE - Tuples

TypeScript introduced a new data type called tuple. Tuples are customized array.

Tuple is a new data type which stores pair of values of different data types.

Example: Tuple vs Other Data Types

```
var empId: number = 1;
var empName: string = "Steve";

// Tuple type variable
var employee: [number, string] = [1, "Steve"];
```

Example: Tuple

```
var employee: [number, string] = [1, "Steve"];
var person: [number, string, boolean] = [1, "Steve", true];

var user: [number, string, boolean, number, string];// declare tuple variable
user = [1, "Steve", true, 20, "Admin"];// initialize tuple variable
```

Data Type - Tuples

You can also declare an array of tuple.

Example: Tuple Array

```
var employee: [number, string][];
employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];
```

Add Elements into Tuple

Example: push()

```
var employee: [number, string] = [1, "Steve"];
employee.push(2, "Bill");
console.log(employee); //Output: [1, 'Steve', 2, 'Bill']
```

Data Type - Enum

- A helpful addition to the standard set of data types from JavaScript is the enum. An enum is a way of giving more friendly names to sets of numeric values.

- By default, enums begin numbering their members starting at 0. You can change this by manually setting the value that may contain both string and numeric values.

```
enum Color {
 Red,
 Green,
 Blue,
}
let selectedColor : Color = Color.Green;
console.log(selectedColor) // output: 1
```

Example: String Enum

```
enum PrintMedia {
    Newspaper = "NEWSPAPER",
    Newsletter = "NEWSLETTER",
    Magazine = "MAGAZINE",
    Book = "BOOK"
}
// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

Data Type - Union

Typescript allows us to use more than one data type for a variable or a function parameter. The **'|'** operator is used to create the union type.

Example: Union

```
let code: (string | number);
code = 123;   // OK
code = "ABC"; // OK
code = false; // Compiler Error

let empId: string | number;
empId = 111; // OK
empId = "E111"; // OK
empId = true; // Compiler Error
```

Example: Function Parameter as Union Type

```
function displayType(code: (string | number))
{
    if(typeof(code) === "number")
        console.log('Code is number.')
    else if(typeof(code) === "string")
        console.log('Code is string.')
}

displayType(123); // Output: Code is number.
displayType("ABC"); // Output: Code is string.
displayType(true); //Compiler Error: Argument of type 'true' is not assignable to a parameter of type string | number
```

Type Aliases

We use **type** keyword to define new type aliases.

```
type Point = {
 x: number;
 y: number;
}; // Point is a type now and we can use it

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " +
pt.x);
  console.log("The coordinate's y value is " +
pt.y);
}

printCoord({ x: 100, y: 100 });
```

```
type Combine = number | string;

function addOrConcat(input1: Combine , input2: Combine ) {
    let result;
    if (typeof input1 === 'number' && typeof input2 === 'number') {
        result = input1 + input2
    } else {
        result = input1.toString() + input2.toString()
    }
    return result;
}

const add = addOrConcat(3,25);
console.log(add) //28

const concatString = addOrConcat('type','Script');
console.log(concatString) //typeScript
```

String Literals

String literals allow us to use a string as a type.

```
type pet = 'cat' | 'dog';

let pet1: pet = 'cat';
let pet2: pet = 'dog';
let gator: pet = "horse"; // error
```

Intersection

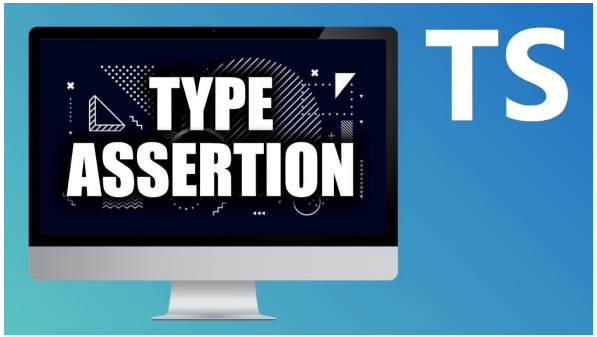
Intersection type allows you to take a cross section of many types

An object of this type will have members from all of the types given. The **'&'** operator is used to create the intersection type.

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

type Admin= {
  privileges: string[];
};


type SuperUser = User & Admin;

const elevatedUser: SuperUser = {
  id: 1,
  name: 'Mark',
  privileges: ['start-database'],
};
```

5

Type Assertions



Type Assertions

Type assertion is a technique that informs the compiler about the type of a variable. Type assertion is similar to typecasting but it doesn't reconstruct code.

You can use type assertion to specify a value's type and tell the compiler not to deduce it. When we want to change a variable from one type to another such as any to number etc, we use Type assertion.

```
let someValue: unknown = "this is a string";
console.log(someValue.length) //Object is of type 'unknown'.
```

```
let someValue: unknown = "this is a string";
console.log((someValue as string).length) // 16
```

```
let someValue: unknown = "this is a string";
console.log((<string>someValue).length) // 16
```

6

Functions

fn

CLARUSWAY
WAY TO REINVENT YOURSELF

Functions

Functions can also include parameter types and return type.

In Typescript, the compiler checks function signature against exact match of parameter order and types.

```
function Greet(greeting: string, name: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

Functions - Optional Parameters

Use the **?** symbol after the parameter name to make a function argument optional.

All optional parameters must follow required parameters and should be at the end.

```
function Greet(greeting: string, name?: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // OK, returns "Hi undefined!".
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

In this example, first parameter is mandatory , second is optional. We can omit name parameter and it's value will be undefined.

Functions - Overloading

You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.

```
function add(a:string, b:string):string;

function add(a:number, b:number): number;

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

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

Different number of parameters and types with same name is not supported.

Functions - Rest Parameters

When the number of parameters that a function will receive is not known or can vary, we can use rest parameters.

In JavaScript (ES6), this is achieved with the "arguments" variable. However, with typescript, we can use the rest parameter denoted by ellipsis.

```
function Greet(greeting: string, ...names: string[]) {
    return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"
```

Rest parameters should be at the end in the function definition. Otherwise the Typescript compiler will show an error.

THANKS!

Any questions?