*Type Script

Ahmed Elashry
aelashry@outlook.com

* Typescript is nothing but JavaScript with some additional features

A brief history of typescript

* in 2010, anders hejlsberg (the creator of typescript) started working on typescript at Microsoft and in 2012 the first version of typescript was released to the public (typescript 0.8).
  Although the release of typescript was praised by many people around the world, due to the lack of support by major ides, it was not majorly adopted by the JavaScript community.
  The first version of typescript (typescript 0.8) released to the public October 2012.
  The latest version of typescript (typescript 5.x.x) was released to the public .

*Why typescript?

* Typescript is open source.
* Typescript simplifies JavaScript code, making it easier to read and debug.
* Typescript is a superset of ES3, ES5, and ES6.
* Typescript will save developers time.
* Typescript code can be compiled as per ES5 and ES6 standards to support the latest browser.
* Typescript can help us to avoid painful bugs that developers commonly run into when writing JavaScript by type checking the code.
* Typescript is nothing but JavaScript with some additional features.

*Typescript features

* Cross-platform: typescript runs on any platform that JavaScript runs on. The typescript compiler can be installed on any operating system such as windows, mac os and Linux.
* Object oriented language: typescript provides powerful features such as classes, interfaces, and modules. You can write pure object-oriented code for client-side as well as server-side development.
* Static type-checking: typescript uses static typing. This is done using type annotations. It helps type checking at compile time. Thus, you can find errors while typing the code without running your script each time.
* DOM manipulation: just like JavaScript, typescript can be used to manipulate the DOM for adding or removing elements.
* ES 6 features: typescript includes most features of planned ECMAScript 2015 (ES 6, 7) such as class, interface, arrow functions etc.

Setup Development Environment

- Install typescript using node.Js package manager (npm). Install the typescript plug-in in your IDE (integrated development environment).
  - npm install typescript
  - Or
  - npm install -g typescript

*Development Environment Setup

* Node.js is an open-source, cross-platform JavaScript run-time environment.
* **we use it to:**
* > Run a js Code.
* > Use the npm tool.
* npm is a Node.js Package Manager for JavaScript programming language.
* npm is automatically installed with Nodejs.

Install tsx module

* If you want to run TypeScript code directly on Node.js without precompiling, you can use the tsx module.

* To install the tsx module globally, run the following command from the Terminal on macOS and Linux or Command Prompt on Windows:

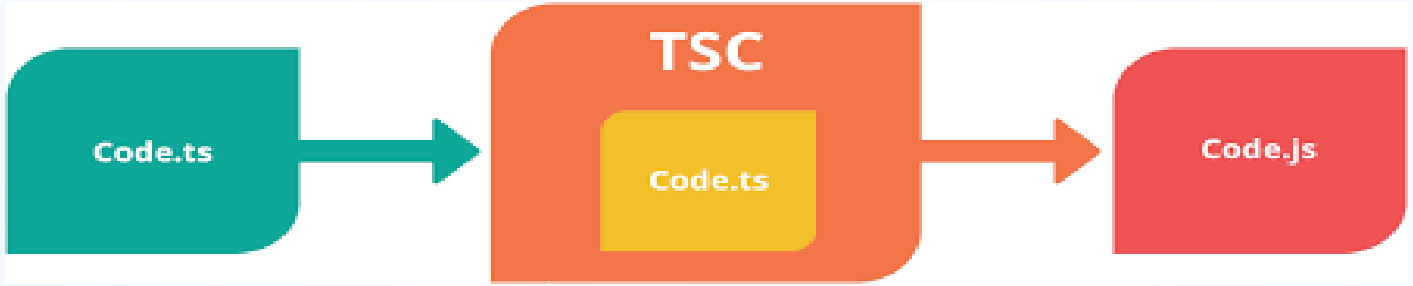
* npm install -g tsx

*Typescript Playground

* Typescript provides an online playground https://www.Typescriptlang.Org/play to write and test your code on the fly without the need to download or install anything.

*Browsers can't execute TypeScript directly. Typescript must be "transited" into JavaScript using the *tsc* compiler, which requires some configuration.

*npx tsc --init



*TSC

TypeScript Hello World program

- First, create a new directory to store the code, e.g., helloworld.
- Second, launch VS Code and open that directory.
- Third, create a new TypeScript file called app.ts. The extension of a TypeScript file is .ts.
- Fourth, type the following source code in the app.ts file:

```
let message: string = 'Hello, World!';
console.log(message);
```

- `tsc app.ts`
- `node app.js` or `tsx app.ts`

*TypeScript Data Type

- *Number
- *String
- *Boolean
- *Array
- *Tuple
- *Enum
- *Union
- *Any
- *Void

Variable Declaration

Variables can be declared using :

var, let, const

TypeScript Data Type - NUMBER

Example: TypeScript Number Type Variables

```
let first:number = 123; // number
let second: number = 0x37CF;  // hexadecimal
let third:number=0o377 ;      // octal
let fourth: number = 0b111001;// binary

console.log(first);  // 123
console.log(second); // 14287
console.log(third);  // 255
console.log(fourth); // 57
```

TypeScript Data Type - STRING

Example: TypeScript String Type Variable

```
let employeeName:string = 'John Smith';
//OR
let employeeName:string = "John Smith";
```

```
let isPresent:boolean = true;
```

Note that, boolean with an upper case B is different from boolean with a lower case b. Upper case boolean is an object type whereas lower case boolean is a primitive type. It is always recommended to use boolean, the primitive type in your programs..

*Data Type - BOOLEAN

*Array types can be written in:

*var list: number[] = [1, 2, 3];

*var list: Array<number> = [1, 2, 3];

*var list:any[] = [1, true, "free"]

Example: Array Declaration and Initialization

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

let ids: Array<number>;
ids = [23, 34, 100, 124, 44];
```

Example: Multi Type Array

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// or
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

*Data Type - Array

```typescript
let employee: object;
employee = {
  firstName: "John",
  lastName: "Doe",
  age: 25,
  jobTitle: "Web Developer",
};
console.log(employee);
//error TS2339: Property 'age' does not exist on type 'object'
employee.age = 30;
```

*TypeScript object Type

```
let employee: {
    firstName: string;
    lastName: string;
    age: number;
    jobTitle: string;
} = {
    firstName: "John",
    lastName: "Doe",
    age: 25,
    jobTitle: "Web Developer",
};
```

*TypeScript object Type

```
let emp: {};
//error TS2339: Property 'firstName' does not exist on type '{}'.
emp.firstName = "John";

let vacant: {} = {};
console.log(vacant.toString());
```

*TypeScript object Type

*Data Type - Tuple

* tuple is a typed array 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. To define a tuple, specify the type of each element in the array
* A good practice is to make your tuple readonly

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 - Enum

*Most object-oriented languages like java and C# use enums. This is now available in typescript too.

*There are three types of enums:

*Numeric enum

*String enum

```
enum Color { Red, Green, Blue };
var color = Color.Blue;
```

*Heterogeneous enum (not recommended) contain both string and numeric values

```
enum color {
    red = "5",
    green = 60,
    blue = "100",
    yellow = green * 3,
    anyColor = getColor(),
}
Tabnine | Edit | Test | Explain | Document
function getColor() {
    return (color.red as number) * color.green;
}
console.log(color["red"], color.anyColor);
```

*Heterogeneous enum

* Typescript allows us to use more than one data type for a variable or a function parameter. This is called 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
```

*DATA TYPE - ANY

*Typescript has type-checking and compile-time checks. However, we do 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

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' ]
```

*TypeScript Unknown Type

*In TypeScript, the unknown type can hold a value that is not known upfront but requires type checking.

*Unlike the any type, TypeScript checks the type before performing operations on it.

```
let result: unknown;
result = 1;
result = "hello";
result = false;
result = Symbol();
result = { name: "John" };
result = [1, 2, 3];
```

```
let var1:any="hello";
let var2:unknown="hello";
console.log(var1.toUpperCase());
//console.log(var2.toUpperCase()); //error
if(typeof var2=="string") //must be check data type is string
    console.log(var2.toUpperCase());
```

Unknown vs Any

*TypeScript - Never

*The never type in TypeScript represents the values that can never occur. For example, the return type of a function that throws an error is never.

```
//never
Tabnine | Edit | Test | Explain | Document
function error(message: string): never {
  throw new Error(message);
}
error("error");
```

*In TypeScript, a variable of void type can store undefined as a value. On the other hand, never can't store any value.

```
let val1: void = undefined;
let val2: never = undefined; // error: Type 'undefined' is not assignable to type 'never'.
```

*void vs. never

*TypeScript type aliases

* a type alias allows you to create a new name for an existing type.
* Type aliases can be useful for:
  * Simplifying complex types.
  * Making code more readable.
  * Creating reusable types that can be used in many places in the codebase.

```
type done = 200 | "ok";
let done0: done = 200;
let done1: done = "ok";
console.log(typeof done0, done1); //number ok
```

```
type Person = {
  name: string;
  age: number;
};
let person: Person = {
  name: "John",
  age: 25,
};
```

*type casting

* Using the angular bracket <> syntax.

```
let code: any = 123;
let employeeCode = <number> code;
```

*Using as keyword

```
let code: any = 123;
let employeeCode = code as number;
```

```
type done = 200 | "ok";
let done0: done = 200;
let done1: done = "ok";
console.log(typeof done0, done1); //number ok
let x1 = "ok";
let x2 = x1 as done;
console.log(typeof x2); //string
```

*null vs. undefined

* In TypeScript, 'undefined' denotes that a variable has been declared but has not been assigned any value.
* On the other hand, 'null' refers to a non-existent object which is basically 'empty' or 'nothing'.

*Functions

* Type Annotation for parameter and return type.
* Optional and Default Parameter.
* Function Overloads.
* Fat Arrow functions.
* Rest parameters.
  * denoted by '...argumentName' for the last argument allow you to quickly accept multiple arguments in your function and get them as an array.

```
function display(a:string, b:string):void
{
    console.log(a + b);
}
```

*OPTIONAL PARAMETERS IN FUNCTIONS

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

Example: Optional Parameter

```
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.
```

*Function Overloading

*TypeScript provides the concept of function 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.

*The last function should have the function implementation

```
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
```

* Able to create a component that can work over a variety of types rather than a single one.

```
function getArray<T>(items : T[] ) : T[] {
    return new Array<T>().concat(items);
}

let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // Compiler Error
myStrArr.push(500); // Compiler Error
```

* Generics