Tema 2 – TypeScript

- ¿Qué es TypeScript? ¿Por qué no JavaScript?
- Instalar compilador de TypeScript
- Tipos en TS vs JS
- Interfaces
- Clases

¿Qué es TypeScript?

- No es un lenguaje como tal, sino un superconjunto de JavaScript.
- Todo código JS es válido en TS, pero TS añade nuevas características:
  - Fuerte tipado: comprobación de tipos
  - Orientación a objetos: Clases, interfaces, constructores, elementos públicos y privados, propiedades.
  - Tipos genéricos: funciones, clases
- Se detectan errores en tiempo de compilación

Instalar compilador de TypeScript

```
> npm install -g typescript

> tsc --version
```

tsc es el compilador o transpilador de TypeScript a Javascript. Es decir, transforma un fichero .ts en su correspondiente en .js

Typescript “transpila” a javascript

micomponente.ts

```
export class MiComponente {

};

let comp1 = new MiComponente();
```

micomponente.js

```
var MiComponente = (function () {
    function MiComponente() {
    }
    return MiComponente;
}());
;
var comp1 = new MiComponente();
```

```
> tsc micomponente.ts
```

Cuando Angular compila el proyecto, usa el compilador de Typescript para transformar (o “transpilar”) los ficheros TS a JS, de forma que puedan ser ejecutados en un navegador. Recordar que un navegador solo ejecuta JS, no TS.

Nosotros mismos podemos compilar un fichero TS para ver el resultado en JS, usando “tsc” en la línea de comandos, pasándole como parámetro el nombre del fichero TS. Se generará otro con el mismo nombre y extensión JS.

Tipos en TS vs JS

- En JS el tipo es variable

JS

```
var cosa = "una cosa"; // cosa es "string"

cosa = 67; // ahora cosa es "number"
```

- En TS el tipo es FIJO

TS

```
let cosa = "una cosa"; // cosa es "string"

cosa = 67; // ERROR DE COMPILACION
```

En JS, podemos cambiar el tipo de valor de una variable en cualquier momento.

En TS, si asignamos un tipo a una variable al crearla, ya no se puede cambiar, y además el compilador comprueba que su uso es correcto y si no da un error.

Tipos en TS vs JS

- Asignar tipo a una variable:

TS

```
// Forma EXPLICITA:
let contador: number;

// Forma IMPLICITA
let contador = 67; // El tipo se infiere del valor inicial
```

- Tipos en TypeScript:
  - Básicos: *number*, *boolean*, *string*, y arrays de estos
  - *"any"* : admite cualquier valor

Podemos asignar un tipo de 2 formas:
- Explicitamente, indicando el tipo al declarar
- Implicitamente, dando un valor de un tipo concreto al declarar

Los tipos básicos son: number, boolean, string, y arrays de los mismos ([]).

Adicionalmente tenemos el tipo "any", que admite cualquier valor. Es el único caso en el que TS no va a comprobar el tipo de la variable. Es igual a como se comporta JS.

Tipos en TS vs JS

- Enumeraciones

JS

```
const ROJO = 0;
const VERDE = 1;
const AZUL = 2;

var miColor = ROJO;
```

TS

```
enum Color {ROJO, VERDE, AZUL};

let miColor = Color.ROJO;
```

Las enumeraciones sirven para declarar grupos de constantes relacionadas.

El poner los valores del enumerado en mayúsculas es una convención, no es obligatorio.

Interfaces

- Estructuras predefinidas

JS

```
function dibujarPunto(x, y) { ... };

dibujarPunto('cinco', 'seis'); // JS se lo traga
```

TS

```
interface Punto { x: number, y: number }

function dibujarPunto ( p: Punto ) { // Usa p.x, p.y }

dibujarPunto( { 5, 8 } ); // Compila bien

dibujarPunto({'cinco', 6}); // ERROR AL COMPILAR
```

En ocasiones, nos interesa manejar objetos con una estructura determinada, por ejemplo para entradas o salidas de una función, o para objetos con una estructura compleja de atributos.

En el ejemplo, le pasamos los datos de un punto a la función “dibujarpunto”, pero no hay forma de que en tiempo de compilación detecte si un objeto sigue una determinada estructura.

En Typescript, podemos usar una "interface", que precisamente define una estructura determinada, como si fuera un tipo de dato. De esa forma, el compilador validará si los objetos tienen la estructura marcada por la interface.

Como en la práctica las interfaces están definiendo tipos de datos, debemos usar, como buena práctica, la notación UpperCamelCase en los nombres de las interfaces: "Punto" en lugar de "punto"

Clases

- Encapsulación de datos y comportamiento

TS

```
class Punto {
    x: number;
    y: number;

    dibujar() {
        // .. Usaremos this.x, this.y
    }
}
------
let p = new Punto();
p.x = 5;
p.y = 8;
p.dibujar()
```

Como en cualquier otro lenguaje orientado a objetos, las clases son fundamentales para mantener la cohesión entre los datos y las funciones relacionadas.

En el caso anterior, existe una relación clara entre la estructura Punto y la función dibujarPunto. En OO, encapsulamos la estructura (propiedades) y las funciones (métodos) relacionadas en una clase.

Para crear objetos de la clase Punto, usamos el operador "new", de la misma forma que en Java.

Clases

- Constructores

TS

```
class Punto {
    x: number;
    y: number;

    constructor(x?: number, y?: number){
        this.x = x;
        this.y = y;
    }
}

let p = new Punto();

let p2 = new Punto(2,4);
```

En las clases usamos una función especial llamada "constructor()", que permite inicializar las propiedades del nuevo objeto.

OJO, a diferencia de Java, TS no se permite múltiples constructores. Solo puede haber uno como máximo. Podemos aliviar parcialmente esta limitación, haciendo alguno o todos los parámetros opcionales, con el operador “?”.

Clases

- Modificadores de acceso:
  - *public (acceso externo), private (acceso interno), protected (acceso subclases)*
- Generar campos a partir de parametros del constructor:

TS

```
class Punto {

  constructor(public x?: number, private y?: number){  }
   ...
}

let p = new Punto(5, 8);
let x = p.x // válido, 'x' es publico
let y = p.y // ERROR, 'y' es privado
```

Como en java, podemos limitar la visibilidad de los atributos de una clase, mediante los modificadores de acceso:

- public: por defecto, el atributo es visible y modificable desde fuera de la clase
- private: el atributo no es visible desde fuera de la clase
- protected: el atributo es visible solo para subclases

Modificadores de acceso en los parámetros de un constructor:
Para reducir el código de los constructores, y hacerlo menos repetitivo, TS crea atributos a partir de los parametros del constructor, siempre que estos incluyan un modificador de acceso expresamente.

Enlaces de interés

https://www.typescriptlang.org/docs

https://www.javatpoint.com/typescript-tutorial