

# TEMA 12. Introducción a TypeScript.

Desarrollo Web en Entorno Cliente.

Profesor: Juan José Gallego García

# Índice :

- Introducción.
- Instalación del compilador de TypeScript
- Ejemplo “Hola mundo”
- Principales características de TS
- TS y jQuery
- Bibliografía.

# Introducción

**TypeScript** es un superconjunto de JavaScript que una vez compilado/transpilado, se obtiene un archivo javascript original. Fue lanzado en 2012, desarrollado y mantenido por Microsoft es de código abierto, y básicamente, añade tipos estáticos y objetos basados en clases (reales).

Esa diferencia con JS, que ampliaremos más adelante, hace que sea más fácil la detección de errores en tiempo de desarrollo, y que el código sea más mantenible y escalable, sobre todo en aplicaciones con un gran número de líneas de código.

## Instalación del compilador de TypeScript

Para la instalación seguiremos los siguientes pasos :

- Necesitamos tener instalado previamente, **Node.js** y su gestor de paquetes **npm**. Enlace para descarga : [Node.js](#) y lo instalamos.  
Podemos ver la versión de **Node.js** (puede que sea necesario lanzar el servicio, lanzando la aplicación) y **npm** en la consola (cmd) con :

`node -v`

`npm -v`

- Para instalar el compilador de **TS** ejecutamos en consola : **npm install -g tsc**
- Comprobamos en consola la versión con: **tsc -v**  
(Si no muestra la versión, habilitar script en Powershell como administrador, comando: **Set-ExecutionPolicy RemoteSigned -Force**)
- Para actualizar a la última versión: **npm install -g typescript@latest - --force**

## Ejemplo “Hola mundo”

1. Abrimos VSCode y activamos la consola **ctrl+ñ**
2. Creamos desde consola el archivo de configuración por defecto para la compilación con **tsc --init** , se creará el archivo **tsconfig.json**
3. Creamos un nuevo archivo con extensión **.ts** y copiamos el siguiente código:

```
let x : string;
x='Hola mundo';
console.log(x);
```

4. Compilamos el archivo desde consola con **tsc nombre.ts** y obtenemos un archivo equivalente en JS : **nombre.js**

5. Para no tener que estar compilando continuamente al hacer cambios en el archivo **.ts** , tecleamos en consola **tsc -w** , y automáticamente, cada vez que se guarde el archivo fuente se obtiene la nueva compilación para **.js**
6. Creamos un archivo base HTML , e insertamos el archivo **.js** para probarlo. Para usar compatibilidad de módulos insertados **import/export** cambiamos en el archivo **tsconfig.json** **"module": "ES2020"**, **"target": "ES2020"**
- 7.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script type="module" src="nombre.js"></script>
</head>
<body>
</body>
</html>
```

# Principales características de TS

En este apartado vamos a ver las principales características de TS, destacando las diferencias que hacen que este lenguaje aporte una mejora sobre JS. No se pretende estudiar en profundidad el lenguaje, ya que como se mencionó al principio, TS es un superconjunto de JS.

## Tipificado de datos.

Es, quizás, una de las características más importante de TS, ya que permite declarar el tipo de dato que se va a usar, aunque también puede inferirse, estableciendo el tipo automáticamente.

A continuación, vemos un resumen de los tipos que admite TS, se muestran las declaraciones y asignaciones del valor en la misma instrucción. La principal ventaja que adquiere TS con el tipificado estático, es tener un mejor control de los errores que se puedan dar, se detectan en tiempo de compilación con VSCode no es necesario esperar a ejecutar el código para capturar el error.

```
- let b: boolean= false;
- let numero: number= 3;
- let cadena: string = 'hola';
- let ambos: number|string; // Permite ambos tipos, se denomina UNIÓN
ambos=3;
ambos='cadena';
- let varArray: number[]=[2,5,1]; // Array numérico.
// Tupla es un array con un número fijo de elementos de distinto tipo.
- let tupla: [string,number]= ['hola',3];
- enum Color {blanco,azul,verde}; // Se declara Enum,para usar nombres (más manejables) en lugar de números
console.log(Color.blanco) // Imprime 0
- let cualquier: any ; // ANY permite declarar un dato de cualquier tipo, en la asignación tomará el tipo.
cualquier=[1,2,3]; // Adquiere tipo Array
cualquier='hola'; // Adquiere tipo string
// Void para funciones que no devuelven un valor.
- function f1 ():void { console.log('ok') }
// Nulo e indefinido
- let u: undefined = undefined;
- let n: null = null;
// Objeto, con INTERFACES también es posible, lo veremos más adelante.
- let objeto:{a:number,b:string}={a:1,b:'hola'};
console.log(objeto.b);
```

# Tipos Genéricos

Aunque hemos visto como con `any` podemos declarar una variable de cualquier tipo sin tener que conocerlo, tiene el problema que permite cambiar dinámicamente los tipos, y por tanto perdemos el control de saber en todo momento de qué tipo es una variable, característica muy importante que nos ofrece TS con la tipificación.

Aquí es donde entran los **genéricos**, se declaran tras el nombre de una función con `<varTipo>`, nos permite usar un tipo genérico en una función que en el momento de la llamada toma el tipo. En el siguiente ejemplo se define una función que recibe un parámetro, lo muestra y lo devuelve, usando tipo numérico :

```
function muestra (valor: number): number
{
    console.log(valor);
    return valor;
}
```

Si nos pidieran que el dato fuese un `string` habría que , o definir de nuevo la función con ese tipo, usar `any` con los inconvenientes que hemos señalado, o usar los genéricos.

Ver sintaxis y ejemplo completo  
a continuación. →

```
// Genéricos

function muestra<elTipo>(valor: elTipo): elTipo {
    console.log(valor);
    return valor;
}

function muestraAny (valor: any): any {
    console.log(valor);
    return valor;
}
```

```
// No hay problemas, al modifica su tipo
// en la asignación de retorno.

let numerico:number=5;
numerico=muestraAny(numerico);
let a1:number=muestraAny('hola');

// Nos da error de compilación, ya que a2 se
// declara número y muestra() devuelve un string.

let numerico1:number=5;
numerico1=muestra(numerico1);
let a2:number=muestra('hola');
```

## Alias con 'Type'

Anteriormente hemos visto los distintos tipos de declaraciones que permite TS, pero también podemos crear nuestros propios tipos usando un alias con la palabra clave `type`.

Por ejemplo, si queremos crear un alias para declarar tipos de variables que permitan números y cadenas, lo usamos de la siguiente forma:

```
type strNum = string|number;  
let tip:strNum;  
tip='gg';  
tip=3;
```

`strNum` es un alias para poder declarar variables que admiten `string` y `number`, posteriormente se declara `tip` del tipo creado y se le puede asignar ambos tipos de datos.

## Funciones

Al igual que en JS, para definir funciones usamos la palabra clave **function**, o las demás formas que conocemos por ejemplo como función **flecha**, con la diferencia que usaremos el tipificado de los parámetros y el tipo de valor retornado.

```
let fun1=function (a:number,b:string):number{return a}; // retorna el valor..  
console.log(fun1(3,'tres')); // Escribe en consola el valor retornado ..  
// Definida como función flecha ..  
let fun2=(a:number,b:string):string=>`El número ${a}, ${b}`; // retorna la expresión  
console.log(fun2(3,'tres')); // Escribe en consola la expresión retornada
```

## Clases

Las clases se definen igual que en JS con la palabra clave **class**, pero teniendo en cuenta de nuevo el tipificado, y mejora (entre otras) poder definir los atributos o métodos como solo-lectura, privados, públicos o protegidos. Si no se indica el modificador se considera por defecto como público.

Sobre herencia entre clases, igual que en JS con **extends**

```
class Persona {  
    private nombre:string  
    public apellido:string  
    constructor (nombre:string, apellido:string) {  
        this.nombre=nombre;  
        this.apellido=apellido;  
    }  
    get ():void {  
        console.log(`Mi nombre es ${this.nombre}`);  
    }  
}  
  
let personal= new Persona('Juan','Pérez');  
personal.get();  
// console.log(personal.nombre) ---- No compila ya que es propiedad privada.  
console.log(personal.apellido);
```

## Interfaces

Son utilizados por TS para definir tipos en las clases u otros elementos. Se encargan de definir qué atributos o métodos (públicos, no privados) van a poder implementar una o varias clases (posteriormente) y sus tipos, sin saber el cómo. Esto va a permitir crear una especie de contrato o firma de estructura de miembros y sus tipos para que las clases lo cumplan, y que va a servir para :

- Dar consistencia a las clases.
- Clarificar las funciones y los parámetros, así como sus tipos (tanto de entrada como de salida).
- Definir archivos de definición de tipos para librerías y frameworks.

Por otro lado, también permiten crear objetos con sus propiedades tipificadas.

```
interface Figura {  
    a: number;  
    b: number;  
    c?:number; // Opcional  
    area (): number;  
}  
  
class Cuadrado implements Figura {  
    readonly a: number; // De solo lectura  
    b: number;  
    constructor (a: number,b:number){  
        this.a=a;  
        this.b=b;  
    }  
    area (): number{return this.a*this.b};  
}
```

```
class Rectangulo implements Figura {  
    a: number;  
    b: number;  
    constructor (a: number,b:number){  
        this.a=a;  
        this.b=b;  
    }  
    area (): number{return this.a*this.b};  
}  
  
let fig1= new Cuadrado(3,4);  
//fig1.a=5; --- No se permite es de sólo lectura  
console.log(fig1.area());
```

```
interface Obj {  
    nombre: string  
    apellido? : string  
    edad : number  
}
```

```
const p1 : Obj = {  
    nombre:"holo",  
    edad :3  
}
```

## Decoradores

Se genera a partir de una función que recibe en su caso más básico un único parámetro, otra función. El decorador recibe dicha función como argumento (aquella que se quiere decorar: e.j. una clase o alguno de sus miembros), y devuelve esa función con alguna funcionalidad adicional sin modificar la original, o registra algún tipo de metadatos (anotaciones sobre los propios datos).

La sintaxis se crea a partir de [@nombreDecorador](#) (que es el nombre de la función que realiza los cambios) anteponiéndose al elemento a decorar (Clase, método, propiedad, parámetro, etc..).

En el ejemplo que viene a continuación se muestra cómo modificar el comportamiento del constructor de una clase y de un método, comentando y descomentando la acción de los decoradores podremos ver cómo cambia el resultado en la consola.

Para que funcione debemos descomentar o añadir las siguientes líneas en el archivo: [tsconfig.json](#)

```
"experimentalDecorators": true  
"emitDecoratorMetadata": true
```

```

// Decoradores

// Modifica constructor de la clase
function decorClase (target:Function):any {
    return function(){console.log('Constructor modificado')};
}

// Modifica método de la clase
function decorMetodo(target:any, propName: string, descriptor: PropertyDescriptor = {}){
    descriptor.value=function(){ console.log('Método modificado')}
    return descriptor;
}

//@decorClase
class MiClase {
    constructor() {
        console.log('Constructor de clase')
    }
    //@decorMetodo
    get() {console.log('Método de clase')};
}
var dat=new MiClase();
dat.get();

```

- En este primer estado, no intervienen los decoradores por lo que se imprimen los correspondientes mensajes.
  - Si descomentamos los decoradores podemos ver cómo se modifican los mensajes en consola:
- Ej1: descomentamos **@decorClase** y comentamos **dat.get()**
- Ej2: comentamos **@decorClase** y descomentamos **@decorMetodo** y **dat.get()**

## Espacios de nombres

En las últimas versiones de TS, debemos diferenciar [módulos](#) de [espacios de nombres](#). Los primeros se declaran y usan como ya vimos también en JS, con ***export/import*** en distintos archivos.

Los espacios de nombres son usados principalmente para crear módulos internos en un mismo archivo **.ts**, es decir agrupaciones de código común para organizarlo y a la vez encapsularlo, y así no tener que usar tantos nombres de elementos. Para declarar un espacio de nombres se usa la palabra clave [namespace](#) y se puede exportar con [export](#) los elementos que quieran ser reconocidos fuera de ese espacio de nombres, de lo contrario el ámbito se extiende sólo dentro del espacio. Sintaxis :

```
namespace Nombre{  
    export elemento ...  
    // Código ....  
}
```

Ej :

```
namespace NombreEsp1{  
  
    let num1:number;  
    export function esp1 ():void {  
        console.log('Hola desde esp1')  
    }  
    function esp2():void {  
        console.log('Hola desde esp2')  
    }  
    esp2(); // Imprime 'Hola desde esp2'  
}  
  
// console.log(num1); // Error, no compila..  
// NombreEsp1.esp2(); // Error, no compila..  
NombreEsp1.esp1(); // Imprime 'Hola desde esp1'
```

Podemos comprobar en el ejemplo, cómo se puede hacer referencia a los elementos que están dentro del espacio de nombres directamente, pero para aquellos que son exportados es necesario el prefijo del espacio.

Por otro lado, daría error de compilación referenciar cualquiera de los que no sean exportados.

Para ampliar información ver **Bibliografía**

# TypeScript y jQuery

Podemos hacer uso de [jQuery](#) con [TS](#) en el editor de VSCode, para ello:

- Instalamos los paquetes necesarios en VSCode `npm install --save-dev @types/jquery`
- En el archivo .HTML añadimos la ruta del archivo .js o CDN de [jQuery](#)
- Y en el archivo **tsconfig.json** añadimos `"types": ["jquery"]`

Un ejemplo de HTML quedaría compuesto por la siguientes referencias :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
<script type="module" src="compiladoDesdeTS.js"></script>
</head>
<body>
    <div id="div1"></div>
</body>
</html>
```

Se reconoce cualquier instrucción [jQuery](#) en el archivo **TS**, por ejemplo :

```
$function() {
    // Código aquí ....
})
```

# Bibliografía

- <https://www.typescriptlang.org/>
- <https://rmolinamir.github.io/typescript-cheatsheet/>