





Angular 17

Tema 2. Introducción a TypeScript

Objetivos

-  Conceptos básicos de TypeScript
-  Sintaxis del lenguaje

Contenidos

- Conceptos básicos. Herramientas
- Clases, campos, métodos, interfaces, herencia, modificadores de acceso
- Tipos de datos , arrays y operadores especiales. Conversión de tipos
- Genéricos
- Expresiones lambda
- @Decoradores

Qué es TypeScript

Es un lenguaje de código abierto desarrollado por Microsoft. Es un superconjunto de JavaScript, diseñado para permitir la creación de código fuente más robusto, al estilo de Java o C#.

Superconjunto de JavaScript. Cualquier línea de JavaScript compila en TypeScript

Fuertemente tipado. Todo tiene un tipo de datos (por fin).

Los tipos son conjuntos de valores. Puedes asignar **varios** tipos de datos a un elemento o permitir sólo ciertos valores.

Los tipos son estructurales. Muy útil con JSON e interfaces.

Definiciones de clases, herencias e interfaces claras.

Módulos de TypeScript. Permite agrupar el código en unidades reutilizables bien organizadas.

Programación funcional. Las Lambdas son parte integral del lenguaje (similar a los delegados de C#)

Transpilación a JavaScript. Todo el código se puede convertir a JavaScript estándar.

Estructuras de control: if, switch

if

```
1  let valor=42;
2  if (valor>10 && valor <20) {
3    |    console.log('El valor es pequeño');
4  }
5  else {
6    |    console.log('El valor es grande');
7  }
```

switch

```
10  let direccion='oriente';
11  switch (direccion) {
12    |    case 'norte':
13    |        console.log('Iré hacia el norte');
14    |        break;
15    |
16    |    case 'sur':
17    |        console.log('Iré al sur');
18    |        break;
19    |
20    |    case 'oriente':
21    |    case 'este':
22    |        console.log('Me voy al este');
23    |        break;
24    |
25    |    case 'poniente':
26    |    case 'occidente':
27    |    case 'oeste':
28    |        console.log('Voy al oeste');
29    |        break;
30    |
31    |    default:
32    |        console.log('Me he perdido');
33  }
```

Estructuras de control: for, while

for tradicional

```
35 let suma=0;
36 for (let i=0; i < 10; i++) {
37     console.log(`voy a sumar ${i} a ${suma}`);
38     suma+=i;
39 }
40 console.log('Suma vale ' + suma);
```

for ... of

```
42 let lista=['rojo', 'verde', 'amarillo', 'añil', 'rosa'];
43 for (let texto of lista) {
44     console.log('El valor es ' + texto);
45 }
```

for ... in

```
47 for (let i in lista) {
48     console.log('El elemento ' + i + ' vale ' + lista[i]);
49 }
```

while

```
51 let total=0;
52 let i=0;
53 while (i < 10) {
54     console.log(`voy a sumar ${i} a ${total}`);
55     total+=i;
56     i++;
57 }
58 console.log('Total vale ' + total);
```

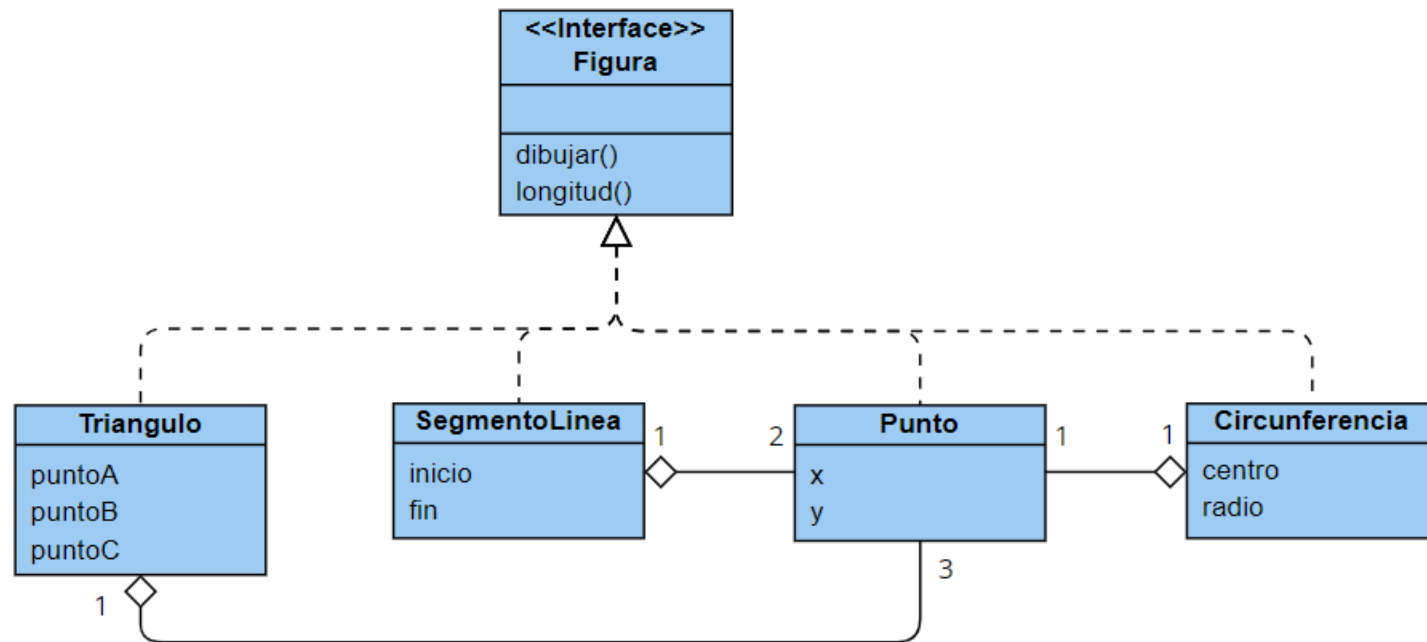
do ... while

```
60 let resultado=42;
61 do {
62     console.log('El bucle se hace al menos una vez');
63     resultado+=Math.random()*100;
64     console.log('Resultado vale ' + resultado);
65 } while(resultado<100);
```

Ejercicio 02 A

1. Elimina los ficheros de JavaScript y renombra los archivos de forma lógica: «punto.ts», «segmentoLinea.ts», «figura.ts» y «programa.ts». Muévelos a una carpeta llama «figuras», transpila y ejecuta de nuevo.
2. Crea la clase «Triangulo», con tres campos de tipo «Punto» y que implemente la interfaz «Figura» como consideres más adecuado. Úsala en el programa. Consejo: Evita el uso de «SegmentoLinea» dentro de esta clase, bázate únicamente en los puntos.
3. Crea la clase «Circunferencia», con los campos «centro» y «radio» y que implemente «Figura». Úsala en el programa.

El esquema UML y los archivos del proyecto quedarían así:



```
▼ SINTAXISTS
▼ figuras
  TS circunferencia.ts
  TS figura.ts
  TS programa.ts
  TS punto.ts
  TS segmentoLinea.ts
  TS triangulo.ts
```

Tipos de datos

Conjuntos de valores

Podemos definir variables de varios tipos a la vez, o restringir el conjunto.

```
5 let textosyNumeros:number | string;
6 textosyNumeros=42;
7 textosyNumeros='cuarenta y dos';
8
9 type deTodo=number | string | boolean;
10 let unValor:deTodo;
11 unValor=123;
12 unValor=false;
13 unValor='un simple texto';
14
15 let imparesYFalse:1|3|5|7|9|false;
16 imparesYFalse=5;
17 //imparesYFalse=12; No compila
18 imparesYFalse=false;
```

Tipos estructurales

Si la «estructura» es correcta, la asignación es válida.

```
20 let unaFigura:Figura;
21 unaFigura={
22     dibujar:()=> 'hola',
23     longitud:()=>42
24 };
25
26 let unaCircunferencia:Circunferencia;
27 unaCircunferencia={
28     centro:new Punto(12,23),
29     radio:23.4,
30     dibujar:function() {
31         return 'hola';
32     },
33     longitud:function() {
34         return 42;
35     }
36 };
```


Tipos de datos básicos

number	Cualquier número, con o sin decimales.
string	Una cadena de texto. Admite literales con comilla simple, doble o acento grave.
boolean	Valores booleanos,. Sólo admite «true» o «false».
array	Arrays de longitud variable pero fuertemente tipados.
«tuplas»	Arrays de longitud fija fuertemente tipados. El equivalente moderno de un «registro».
enum	Enumeraciones. Conjuntos de constantes numéricas agrupadas en un tipo.
void	No hay tipo de datos. Típico de los métodos que no devuelven nada.
any	Cualquier tipo de datos. Lo más parecido al comportamiento tradicional de JavaScript con las variables.
null	Valores nulos. Sólo admite «null». CUIDADO, en Angular no funciona como piensas.
undefined	Tipo de datos sin definir. Tampoco funciona como piensas. Poco usado, salvo en comprobaciones de parámetros.
unknown	Tipo de dato desconocido, y por algún motivo quieres que quede patente que no lo conoces. Poco usado.
never	Funciones que nunca acaban de forma correcta, o que nunca acaban. Poco usado.
object	Cualquier valor que no sea un tipo básico. Podrías asignar un JSON o un «Punto», pero no un «number». Poco usado.

Tipos de datos básicos: number, string, boolean

number

Sólo hay un tipo para representar números, de cualquier tamaño y precisión.

```
1 let cantidadUno:number=34;
2 let cantidadDos=34.56;
3 cantidadUno=4556.33;
```

string

Los textos se crean con comillas simples, dobles o acento grave.

No existe el tipo de datos «char».

No son objetos de JavaScript de clase String (en mayúsculas) , aunque el transpilador al final los convierta a objetos de esa clase.

```
5 let textoUno:string="un ejemplo de texto;"
6 let textoDos='un ejemplo adicional';
7 textoUno=`Uso de plantillas: ${textoUno}, ${cantidadDos}`;
8 textoDos=textoUno.charAt(3);
9 textoDos=textoUno.substring(4);
10
11 let textoTres=new String('es un ejemplo');
```

boolean

Los booleanos se comportan de la forma tradicional.

```
13 let valor:boolean;
14 let otroValor=true;
15 valor=false;
16 valor=cantidadUno==45;
```

Tipos de datos básicos: arrays, tuplas

Arrays

Se comportan igual que en JavaScript, excepto que están fuertemente tipados (luego veremos el tipo «any»).

```
1  let lista:number[];
2  lista=[10,20,30];
3  lista.push(40);
4  //No compila lista.push('12');
5
6  let otraLista=['norte','sur','este','oeste'];
7  otraLista[4]='noroeste';
8  otraLista[7]='suroeste';
9  otraLista['otra']='arriba';
10 //No compila otraLista.push(42);
11
12 //este bucle no funciona como debe
13 for (let texto of otraLista)
14 |   console.log(texto);
15 |
16 //este bucle sí
17 for (let i in otraLista)
18 |   console.log('El elemento ' + i + ' vale ' + otraLista[i]);
```

Tuplas

Arrays de longitud fija (más o menos). Una especie de registro.

```
21 let mateo:[string, string, number];
22 mateo=['Mateo','Aguilar',1200.56];
23
24 type persona=[string, string, number];
25 let marcos:persona=['Marcos','Rodríguez',1900];
26 let lucas:persona=['Lucas','Sánchez',3200];
27 lucas.push(45);
28
29 for (let i in lucas)
30 |   console.log('El elemento ' + i + ' vale ' + lucas[i]);
```

Tipos de datos básicos: any, unknown

any

La variable admite cualquier valor. Úsalo con precaución, no quieres escribir TypeScript como si fuera JavaScript. Muy usado en Angular cuando necesitas referencias a elementos estándares del DOM o de JavaScript.

```
21 let deTodo:any=12;  
22 deTodo='un texto';  
23 deTodo=null;  
24 deTodo=undefined;
```

unknown

También admite cualquier valor, pero «reconoce» que desconoce el tipo. Permite tantas asignaciones como «any», pero cuando quieres copiar el valor a otra variables es cuando aparecen las diferencias. Afortunadamente, el compilador es muy listo.

```
26 let niIdea:unknown;  
27 niIdea=false;  
28  
29 niIdea='es un texto';  
30 //No Compila let unTexto:string=niIdea;  
31  
32 niIdea=42;  
33 //No Compila let unValor:number=niIdea;  
34 let unValor:number;  
35 if (typeof niIdea==='number') unValor=niIdea;
```

Tipos de datos básicos: null, undefined

Por defecto son «subtipos» del resto de valores; cualquier variable de cualquier tipo puede valer «null» o «undefined». Pero cuando la opción de compilación **strictNullChecks** está activa (Angular la tiene activa por defecto) dejan de ser «subtipos» y se comportan como una especie de «boolean», que sólo admiten los valores «null» o «undefined».

null

Sólo admite el valor «null».

```
1  //No Compila let valor:number=null;
2  //No Compila let texto:string=null;
3  let valor:number | null=null;
4  valor=42;
5
6  let lista:string[] | null =['uno', 'dos'];
7  lista=null;
```

undefined

Sólo admite el valor «undefined».

```
9  //No Compila let valor:number=undefined;
10 //No Compila let texto:string=undefined;
11 let otroValor:number | undefined=undefined;
12 otroValor=42;
13
14 let otraLista:string[] | undefined =['uno', 'dos'];
15 otraLista=undefined;
```

Tipos de datos básicos: void, never

void

Casi siempre para métodos que no devuelven ningún valor. Eso no significa que no puedan usar «return».

```
3  unMetodoCualquiera():void {  
4      console.log('un ejemplo');  
5      if (1==1) return;  
6      console.log('Esta línea no se ejecuta');  
7  }
```

never

Métodos que ni devuelven nada ni acaban de forma correcta.

```
9  otroMetodoCualquiera():never {  
10     throw new Error('nunca acaba bien');  
11 }  
12  
13 unoAdicional():never {  
14     while (true) {  
15         console.log('Nunca acaba');  
16     }  
17 }
```

Tipos de datos básicos: enum, object

enum

Una lista de constantes numéricas (comienzan por defecto desde cero) que se comportan como un tipo.

```
3  export enum Altura {
4  |    Alto=1, Medio, Bajo
5  }
6
7  //Equivale a let tipo: 1 | 2 | 3;
8  let tipo=Altura.Medio;
9
10 tipo=3;
11 tipo=Altura.Bajo;
```

object

Un tipo «no básico».

```
13 let unObjeto:object;
14 //No Compila unObjeto=null;
15 //No Compila unObjeto='un texto';
16 unObjeto={nombre:'Javi', edad:89};
17 unObjeto=new Punto(10,20);
```

Sobrecarga de métodos y funciones

La firma de un método o función es la combinación de:

1. Nombre asignado
2. Número, tipo y orden de los parámetros
3. Tipo de retorno definido.

Para «sobrecargar» un método o función hay que declarar **las firmas de sobrecarga** que deberá admitir (sin cuerpo):

```
10 |   acumular(valor:number):void;  
11 |   acumular(texto:string):void;
```

y la **firma de implementación** que define el comportamiento del método o función:

```
12 |   acumular(dato:number | string) {  
13 |       if (typeof dato=='number') this.contador+=dato;  
14 |       else this.contador+=dato.length;  
15 |   }
```

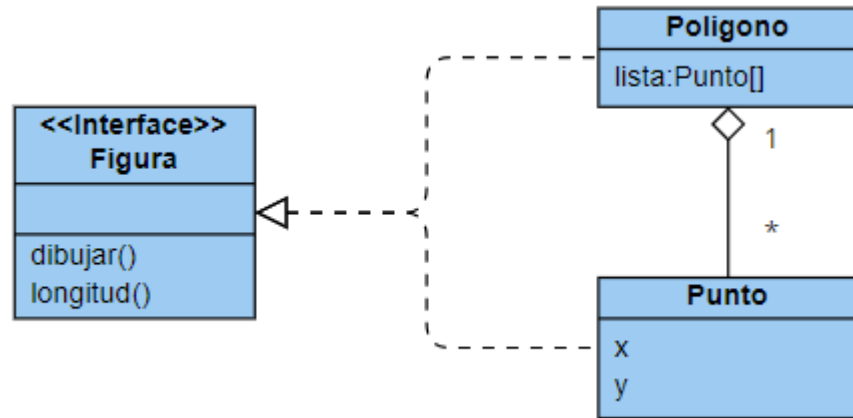
Las firmas de sobrecarga deben tener el mismo número de parámetros.

La firma de implementación debe ser «compatible» con todas las firmas de sobrecarga (número y tipo de parámetros, retorno)

¿Y qué diferencia hay en **definir directamente** la «firma de implementación», sin declarar esas «firmas de sobrecarga»? Pura cuestión de estilo, depende del caso concreto. No es lo mismo declarar «dato:number | string» que «dato:any».

Ejercicio 02 B

1. Usando como base el código del «ejercicio 02 A» (si no lo has completado lo tienes disponible en la sección de ficheros del tema actual) añade la clase «Poligono», de tal modo que en la función de pruebas puedas ejecutar el código de la imagen y obtener el resultado mostrado:



```
const pol1=new Poligono([
    new Punto(10,20),
    new Punto(40,20),
    new Punto(60,30),
    new Punto(40,50),
    new Punto(10,30)
]);
verFigura("POL1", pol1);
```

```
POL1 es (10, 20) --- (40, 20) --- (60, 30) --- (40, 50) --- (10, 30)
POL1 mide 126.70046377709969
```

2. Modifica «Poligono» para que su método «dibujar()» devuelva el texto «No hay puntos» si le pasas un array vacío y «No está definido» si le pasas un **null** al constructor. En ese caso, el método «longitud()» también devolverá null. Aviso: tendrás que modificar también la interfaz «Figura»:

```
const pol2=new Poligono(null);
verFigura("POL2", pol2);
```

```
POL2 es No está definido
POL2 mide null
```

RECUERDA:

```
...\ejercicio02B\figuras> tsc .\programa.ts
...\ejercicio02B\figuras> node .\programa.js
```

Ejercicio 02 C

Crea una clase llamada «Clasificador» con un método «clasificar()» que admita números, textos y caracteres nulos. La clase tiene que recordar los textos y números concretos y cuántos valores nulos ha procesado. Tendrás que definir además métodos o propiedades para poder recuperar la información desde el programa:

```
const c=new Clasificador();
c.clasificar(42);
c.clasificar('rojo');
c.clasificar(17);
c.clasificar(3934);
c.clasificar(null);
c.clasificar('verde');
c.clasificar('azul');
c.clasificar(null);
```

```
//Si te gustan los métodos get al estilo de Java:
c.getNulos()
c.getNumeros()
c.getTextos()

//Si te gustan las propiedades al estilo de C#:
c.cantidadNulos
c.listaNumeros
c.listaTextos
```

Usa los métodos «get» o las propiedades que hayas definido para que en la consola (con un par de bucles y varios «console.log()») aparezca el siguiente resultado:

```
-----
Valores nulos procesados:2
Valores numéricos procesados: 3
  42
  17
 3934
Textos procesados: 3
  rojo
  verde
  azul
```

Expresiones lambda

Son otra forma de definir funciones anónimas, aunque no interpretan el contexto (this) del mismo modo. Muy usadas.

```
1  let suma1=function(op1:number, op2:number) {
2    |   return op1 + op2;
3  }
4
5  let suma2=(op1:number, op2:number)=> {
6    |   return op1 + op2;
7  }
8
9  let suma3=(op1:number, op2:number) => op1 + op2;
```

En ocasiones es necesario definir variables de **tipo función**: (parámetros) => valor de retorno.

```
11 let suma4:(op1:number, op2:number) => number;
12 suma4=(op1,op2) => op1 + op2;
13
14 let contar:(string)=>number;
15 contar=texto=>texto.length;
```

No importa la sintaxis concreta, el resultado es el mismo:

```
17 console.log("S1: " + suma1(2,3));
18 console.log("S2: " + suma2(2,3));
19 console.log("S3: " + suma3(2,3));
20 console.log("S4: " + suma4(2,3));
21
22 console.log("Letras: " + contar("es una prueba"));
```

```
S1: 5
S2: 5
S3: 5
S4: 5
Letras: 13
```

¿Qué hemos aprendido?

- Qué es TypeScript. Sintaxis general
- Sintaxis POO
- Tipos de datos. Uso de genéricos
- Funciones lambda
- @Decoradores

Ejercicio 02 D (1/2)



Quiero que crees dos clases, «Vehiculo» y «Garaje», de tal modo que puedas ejecutar el siguiente «programa» de pruebas:

```
111 function prueba() {
112
113     let g=new Garaje(14);
114
115     let rojo=new Vehiculo('1122-ABC','coche rojo');
116     let verde=new Vehiculo('7689-XDC','coche verde');
117     let azul=new Vehiculo('9487-LLF','coche azul');
118     let blanco=new Vehiculo('4324-HGF','coche blanco');
119
120     console.log("Aparco ROJO: " + g.aparcar(rojo, 12));
121
122     console.log("Aparco VERDE: " + g.aparcar(verde, 12));
123     console.log("Aparco VERDE: " + g.aparcar(verde, 7));
124
125     console.log("Aparco AZUL: " + g.aparcar(azul, 1));
126
127     console.log("Aparco BLANCO: " + g.aparcar(blanco));
128
129     //desaparcado es "let desaparcado:false | Vehiculo"
130     let desaparcado=g.desaparcar(12);
131     //obligatorio preguntar, de lo contrario no te dejará compilar
132     if (desaparcado!=false) console.log("Aparco el DESAPARCADO: " + g.aparcar(desaparcado,14));
133
134     console.log();
135     g.verEnConsolaPlazasLibres(5);
136     console.log();
137     g.verEnConsolaCoches(5);
138 }
139
140 prueba();
```

Vehiculo es una clase muy simple, con dos campos, la matrícula y una descripción. Asegúrate de que la matrícula siempre sea de ocho caracteres, lanzando un «throw» si no es así o rellenándola, como prefieras.

Garaje es la complicada. Puedes aparcar y desaparcar vehículos, indicando la plaza que quieres ocupar. En el constructor indicas el número de plazas del garaje.

El método **aparcar** te pide el vehículo y la plaza. Si no indicas la plaza, usará la primera disponible. Devuelve un boolean indicando si ha podido aparcar (tal vez la plaza ya esté ocupada). Las plazas están numeradas de «1 a n». La plaza «cero» no existe.

El método **desaparcar** sólo pide la plaza. Devuelve el vehículo que ha desaparcado o bien un false si en esa plaza no había nadie.

Ejercicio 02 D (2/2)

El resultado de la ejecución debería ser aproximadamente el siguiente:

```
Aparco ROJO: true
Aparco VERDE: false
Aparco VERDE: true
Aparco AZUL: true
Aparco BLANCO: true
Aparco el DESAPARCADO: true

01 02 03 04 05
X  X
06 07 08 09 10
   X
11 12 13 14
      X

.....1 .....2 .....3 .....4 .....5
9487-LLF 4324-HGF
.....6 .....7 .....8 .....9 .....10
      7689-XDC
.....11 .....12 .....13 .....14
                        1122-ABC
```

El método **verEnConsolaPlazasLibres** te muestra en la consola el número de plaza y su estado, con una «X» si está ocupada. Te permite indicar el número de columnas que usarás para mostrar el resultado. Lo quiero bien formateado. Fíjate que he añadido «ceros» a los números de plaza cuando ha sido necesario para poder presentarlo bien.

verEnConsolaCoches es similar, sólo que muestra las matrículas. Como los ceros quedaban raros he añadido puntos para presentarlo mejor.