

# Anotaciones sobre curso Typescript Udemy

Notes	1
Compilar un archivo TS	2
Modo Observador - Watch Mode	3
Inicializar el proyecto de TypeScript - tsconfig	4
Introducción a los tipos de datos	4
STRINGS	4
NÚMEROS	4
BOOLEANOS	4
NULL Y UNDEFINED	4
OBJETOS LITERALES	5
CLASES	5
FUNCIONES	5
ES POSIBLE	1
Booleans - Booleanos	6
Numbers - Números	6
Strings - Cadenas de caracteres	6
Tipo Any	7
Arrays - Arreglos	8
Tuples - Tuplas	1
Enum - Enumeraciones	9
Void - Vacío	10
Never - Nunca	11
Null - Undefined	11
Aserción de tipo	13
Funciones Básicas	14
Parámetros obligatorios de las Funciones	15
Parámetros Opcionales de las Funciones	15
Parámetros por Defecto	16
Parámetros REST	17
Tipo Función	19
Objetos básicos	20
¿Cómo crear objetos con tipos específicos?	20

<b>Métodos dentro de los objetos</b>	<b>20</b>
<b>Tipos personalizado</b>	<b>21</b>
<b>Múltiples tipos permitidos</b>	<b>22</b>
<b>Revisar el tipo de una objeto o variable</b>	<b>22</b>
<b>Depurar código de TypeScript</b>	<b>23</b>
<b>Remover los comentarios al compilar</b>	<b>23</b>
<b>Características de ES6</b>	<b>24</b>
Características:	24
Variables Let	25
Constantes - const	28
Templates Literales	29
Funciones Flecha	30
Desestructuración de Objetos	34
Nuevo Ciclo - For Of	35
Clases en ES6	35
Desestructuración de Arreglos	36
<b>Clases en TypeScript</b>	<b>37</b>
Definición de una clase básica en TypeScript	37
Constructores	37
Propiedades públicas, privadas y protegidas	38
Métodos públicos, privados y protegidos.	40
Herencia, super y definición de propiedades en el constructor	42
Gets y Sets	42
Métodos y propiedades estáticos	46
Clases Abstractas	47
Constructores privados	48
<b>Interfaces</b>	<b>48</b>
Interfaz Básica	48
Propiedades Opcionales	48
Métodos en la Interfaz	49
Interfaces en las Clases	51
Interfaces para las funciones	53
Notes	54
<b>Módulos</b>	<b>55</b>
<b>Genéricos - Generics</b>	<b>56</b>
Introducción a los Genéricos	56
Creando funciones genéricas	57

Ejemplo de función genérica en acción	60
Arreglos Genéricos	61
Clases genéricas	62
<b>Decoradores</b>	<b>63</b>
*Introducción a los decoradores	63
Decoradores de clases	64
Decoradores de fábrica - Factory decorators	64

# Notes

---

Las interfaces y los tipos son ignorados al momento de ser compilados a JavaScript

---

```
tsc -v
```

Para ver la versión de Typescript

---

## Instalar paquetes en Atom:

atom - preferences - install

En la caja de texto ponemos el nombre del paquete, y le damos al botón instalar

---

## Colonize

es una extensión de visual studio code

Agrega tres shortcuts para insertar puntos y comas con facilidad:

1. **shift + enter**: Insertar punto y coma al final de la línea y continuar en la misma línea
2. **alt + enter**: Insertar punto y coma al final de la línea y continuar en la nueva línea
3. **ctrl + alt + enter**: Insertar punto y coma y permanecer en la misma posición

# Compilar un archivo TS

oiu

Para entenderlo mejor miremos el siguiente ejemplo:

Creamos una carpeta con el nombre TypeScript, y la abrimos con visual studio code, dandole clic derecho OPEN WITH CODE.

creamos un archivo con el nombre index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Demo</title>
</head>

<body>

  <script src="app.js"></script>

</body>

</html>
```

creamos el archivo app.ts

```
console.log("hola mundo");
```

Para compilar un archivo ts ejecutamos el comando

```
tsc nombre_del_archivo
```

**note:** que no hay necesidad de poner al nombre del archivo el .ts

esto creará un archivo nuevo con el nombre: nombre\_del\_archivo.js

En resumen los navegadores web no pueden correr el código TypeScript directamente, necesita ser traducido a su versión JavaScript para que el navegador web lo pueda interpretar. Mediante el comando tsc lo convertimos de TypeScript a JavaScript.

## Modo Observador - Watch Mode

En TypeScript existe lo que se conoce como Modo Observador, quiere decir que TypeScript va a estar pendiente de cualquier cambio que suceda en los archivos con extensiones .ts y automáticamente lo va a compilar a su versión de JavaScript.

```
tsc nombre_del_archivo -w
```

y esto va a iniciar un servicio que va a estar escuchando cualquier cambio.

para cancelar el observador de TypeScript presiona CONTROL + C

## Inicializar el proyecto de TypeScript - tsconfig

Cómo hacemos para escuchar todos los cambios, para esto necesitamos crear nuestro proyecto de TypeScript

en la raíz del proyecto

```
tsc -init
```

esto nos va a crear el archivo tsconfig.json

eso nos va a permitir escribir el siguiente código:

```
tsc -w
```

es decir TypeScript escucha todos los archivos con extensión ts



# Introducción a los tipos de datos

## STRINGS

```
"Maria perez"  
'Mazda'  
`<h1>Hola mundo</h1>`
```

## NÚMEROS

```
pi = 3.14159265359  
salario = 1500.00  
entero = 1
```

## BOOLEANOS

Verdadero (true)

Falso (false)

## NULL Y UNDEFINED

```
numero = null  
persona = undefined
```

## OBJETOS LITERALES

```
var persona = {  
  nombre: "Stiven",  
  edad: 26  
}
```

Está entre llaves y dentro de esas llaves tiene propiedades y métodos.

propiedades como la que dice nombre y métodos que serían funciones dentro de esa persona.



## CLASES

```
class Persona {  
  nombre;  
  edad;  
}
```

## FUNCIONES

```
function saludar() {  
  return "hola!";  
}
```

a pesar de que pareciera raro decir que una función es un tipo de dato, mentalizarse que pueden definir una variable de tipo función.

## ES POSIBLE

Crear tipos nuevos, a los que ya trae TS por defecto.

También pueden crear interfaces.

Y usar Tipos genéricos.

Documentación: <http://www.typescriptlang.org/docs/handbook/basic-types.html>

## Booleans - Booleanos

```
let esSuperman: boolean = false;  
let esBatman: boolean;  
let esAcuaman = true;
```

TS en la última declaración sabe ya que esAcuaman es un booleano, pero esto no es recomendable de hacer por que no estaríamos aprovechando TS como es debido.

## Numbers - Números

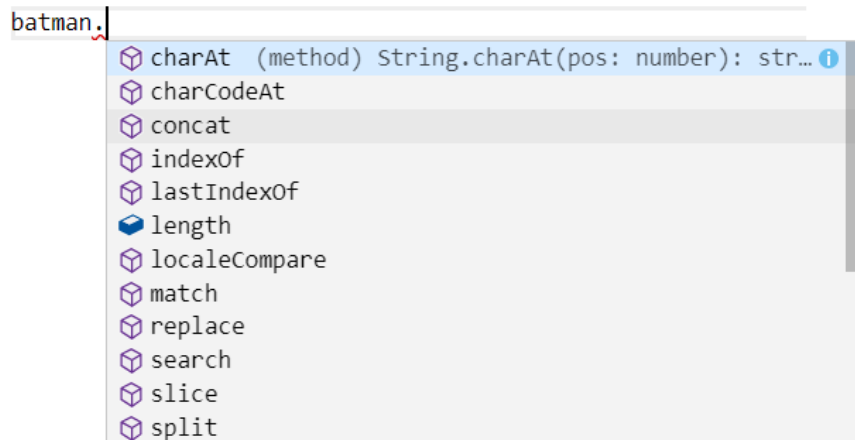
```
let avengers: number = 5;  
let villanos: number;  
let otros = 2;
```

```
if( avengers > villanos ) {  
    console.log("Estamos Salvados!");  
}else {  
    console.log("Estamos muertos!");  
}
```

**Note:** en Javascript toda variable que no está asignada, tienen el valor por defecto undefined. Esto se puede controlar, podemos decirle al TS que nos avise cuando las variables no tengan valores dentro.

## Strings - Cadenas de caracteres

```
let batman: string = "Batman";  
let linternaVerde: string = 'Linterna Verde';  
let superman: string = `Superman`
```



si yo escribo batman. me salen todos los métodos que tiene un string. esto es de gran ayuda a la hora de programar.

La mejor manera para hacer este tipo de uniones de strings es usar los templates literales del EMC6

```
let concatenar: string = "Los héroes: " + batman + ", " + linternaVerde;  
let concat: string = `Los héroes son: ${ batman }, ${ linternaVerde } `;  
  
console.log( concat );
```

**note:** el `` es utilizado para crear strings multilínea.

## Tipo Any

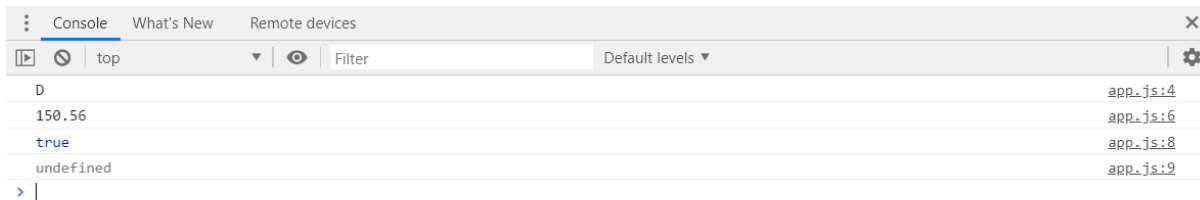
Una variable de tipo Any puede ser cualquier cosa

```
let vengador: any;  
let existe;  
  
vengador = "Dr. Strange";  
console.log( vengador.charAt(0) );  
  
vengador = 150.5555;  
console.log( vengador.toFixed(2));
```

```
vengador = true;
console.log( vengador );

console.log( existe );
```

imprime:



existe va a tener el valor undefined, pero es de tipo any.

## Arrays - Arreglos

los arreglos van a ser igual que en JS solo que le podemos definir que tipo de dato tiene.

```
let arreglo = [1,2,3,4,5,6,7];
```

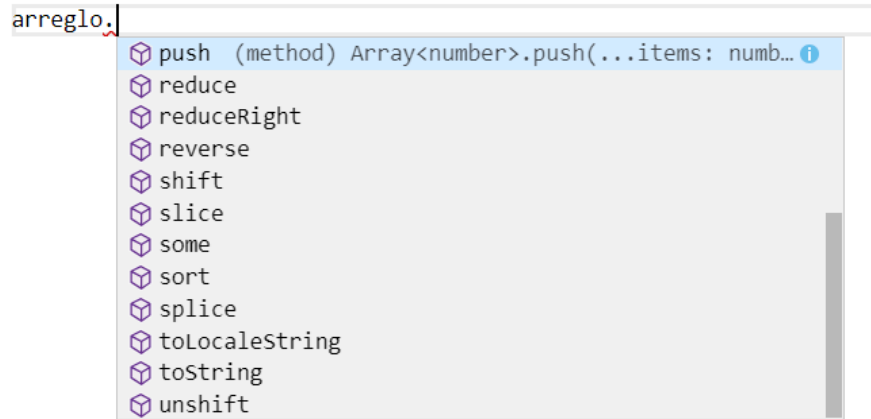
TS va a deducir que es un arreglo de número por el contenido, vemos que al hacer un push de un string TS nos arroja un error:

```
3 let arreglo = [1,2,3,4,5,6,7];
4
5 arreglo.push("123");
```

pero esto no es buena práctica, no tiene sentido que usemos TS para hacer todo de este calibre.

```
let arreglo: number[] = [1,2,3,4,5,6,7];
```

si yo quiero decir que esto es un arreglo de números, tengo que poner [] junto al tipo.



Al definir su tipo tengo como sugerencias todas las características que tiene un arreglo :D.

## Tuples - Tuplas

Las tuplas es un tipo de dato que no existe en JS , pero tienen una forma muy parecida a lo que es un arreglo.

```
3 let hero: [string, number, boolean] = ["Dr. Strange", 100, true];
4
5 hero[0] = 100;
6
7 hero[1] = "100";
8
9 hero.push("puedo hacerle un push");
```

Funciona también para poder hacer lectura de los datos no solo como tupla sino también como tercios, cuartos Etc.

## Enum - Enumeraciones

Pensemos en este tipo de dato para dar un sentido lógico a unos números. pensemos que estamos trabajando en un aplicación que reproduzca audio:

```
enum Volumen{
    min,
    medio,
    max
}
```

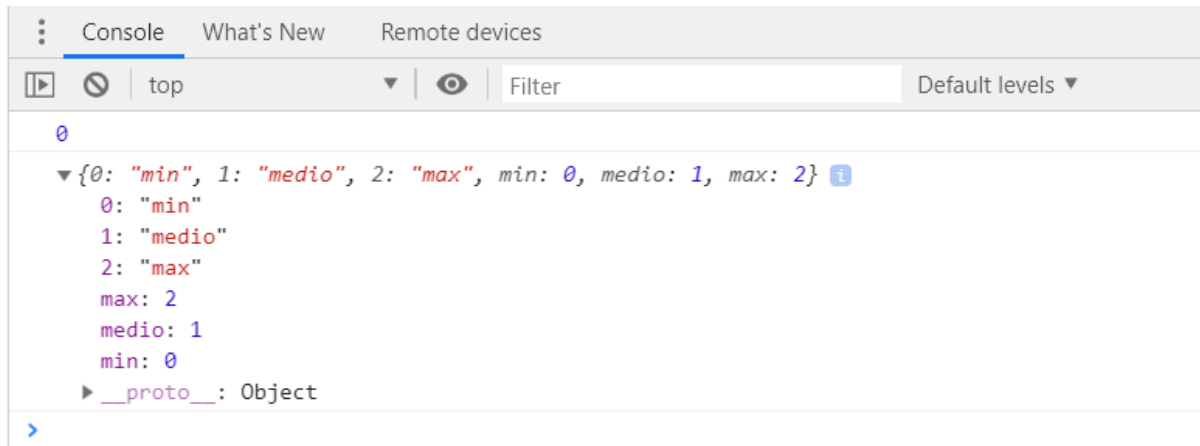
```

let audio: number = Volumen.min;

console.log( audio );

console.log( Volumen );

```



```

enum Volumen{
    min = 1,
    medio,
    max = 10
}

let audio: number = Volumen.medio;

console.log( audio );

console.log( Volumen );

console.log( Volumen[2] );

```

```
7_Enum.ts 7_Enum.js
1  (()=>{
2
3      enum AudioLevel{
4          min = 1,
5          medium,
6          max = 10,
7      }
8
9      let currentAudio:AudioLevel = AudioLevel.medium;
10
11      console.log(currentAudio);
12      console.log( AudioLevel)
13
14  })()
```

```
7_Enum.js
1  "use strict";
2  (() => {
3      let AudioLevel;
4      (function (AudioLevel) {
5          AudioLevel[AudioLevel["min"] = 1] = "min";
6          AudioLevel[AudioLevel["medium"] = 2] = "medium";
7          AudioLevel[AudioLevel["max"] = 10] = "max";
8      })(AudioLevel || (AudioLevel = {}));
9      let currentAudio = AudioLevel.medium;
10      console.log(currentAudio);
11      console.log(AudioLevel);
12  })();
13
```



note:



¡Buen trabajo!

Como "c" se iguala a 9, el siguiente valor es 10, no importa que se repita el valor de la enumeración.

Pregunta 10:

Dada la siguiente enumeración, ¿Qué valor tiene "d"?

```
1  enum enumeracion {
2      a=10,
3      b,
4      c=9,
5      d
6  }
```

## Void - Vacío

es el opuesto a any, void es ningún tipo, vacío.

está más relacionado a lo que es el retorno de una función.

sabemos que en JS una función que no regresa nada realmente si regresa algo, un Undefined.

si yo quiero que una función no regrese nada y solo ejecute algo.  
si yo pongo dos puntos, le estoy diciendo que va a retornar.

```
function llamar_batman(): void {  
    console.log("Mostrar la batiseñal");  
}
```

Note que me muestra error al tratar de retornar algo, cuando está ya definido que no retorna nada.

TS interpreta que Void que no hay un valor de retorno, si una función de tipo void le colocaremos un return, este presenta error ya que no está esperando nada

```
(()=>{  
  
    function callBatman():void{  
        return 1;  
    }  
  
    const a = callBatman();  
    console.log(a);  
  
})()
```

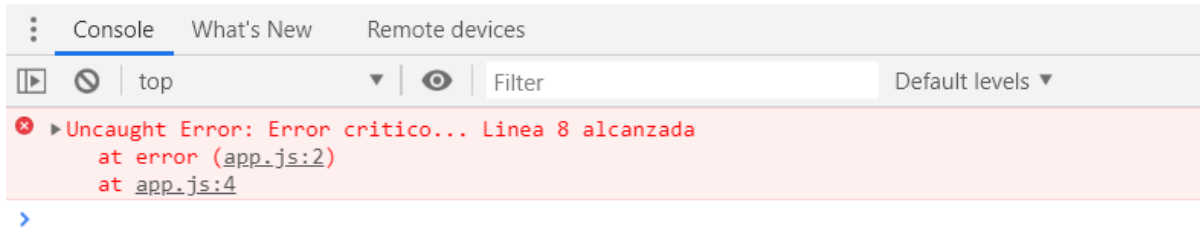
## Never - Nunca

representa un valor que nunca puede suceder.

es por decirlo así, algo que sucede en nuestro programa y si retorna un never, o si tenemos una función con un never, quiere decir que ya falló nuestra aplicación.

si llega a es punto, la aplicación ya no debe de salir de ahí, ahí termina el código.





## Null - Undefined

v

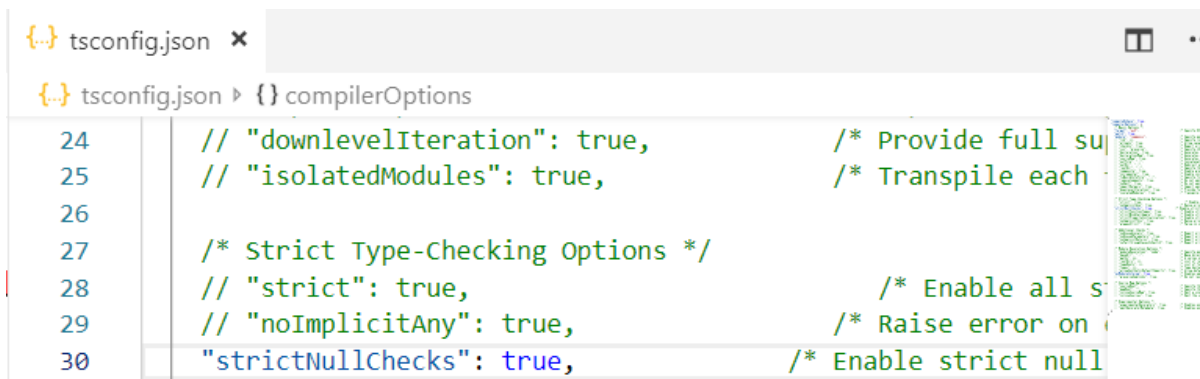
Note que genera error.

```
3 let nada: undefined = undefined;
4
5 nada = null;
```

Pero note que esto sí está permitido. El primer ejemplo ocurre lo mismo si definimos la variable de tipo null. funcionan parecido.

```
3 let ironman: string;
4
5 ironman = "Tony";
6
7 ironman = undefined;
```

En este ejemplo vemos que si acepta el cambio de tipo para este caso



Si en el tsconfig.json descomentamos la opción, strictNullChecks ahora si nos generará error:

```
3 let ironman: string;
4
5 ironman = "Tony";
6
7 ironman = undefined;
```

el compilador de TS me va a impedir que yo pueda igualar variables de cualquier tipo a undefined y null.

por que sin está restricción si pasa?, por que el null y el undefined son valores permitidos para estas variables.

## Aserción de tipo

```
// Aserciones de Tipo
var poder: any = "100";
var largoDelPoder: number = (<string>poder).length;
console.log( largoDelPoder );
```

note: con el <string> le estoy diciendo a TS que trabaje la variable como un string, así permitiéndonos tener todas las opciones del string (la ayuda de autocompletar).

Permitirme poder usar cualquier variable de otro tipo, y decirle a TS confía en mí, yo sé lo que estoy haciendo.

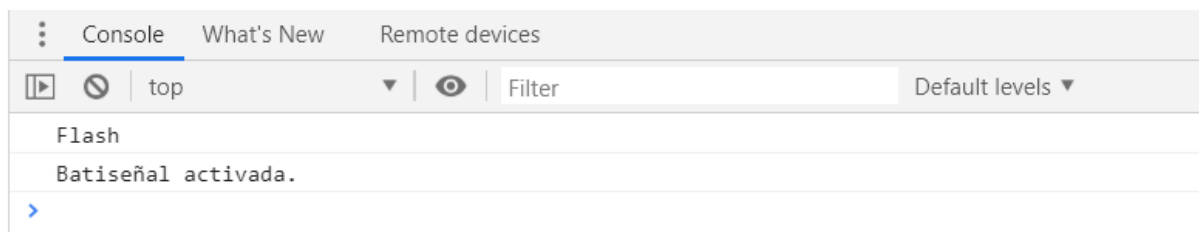
## Funciones Básicas

```
let hero:string = "Flash";

function imprime_heroe():string{
    return hero;
}

let activar_batiseñal = function():string{
    return "Batiseñal activada.";
}

console.log( imprime_heroe() );
console.log( activar_batiseñal() );
```

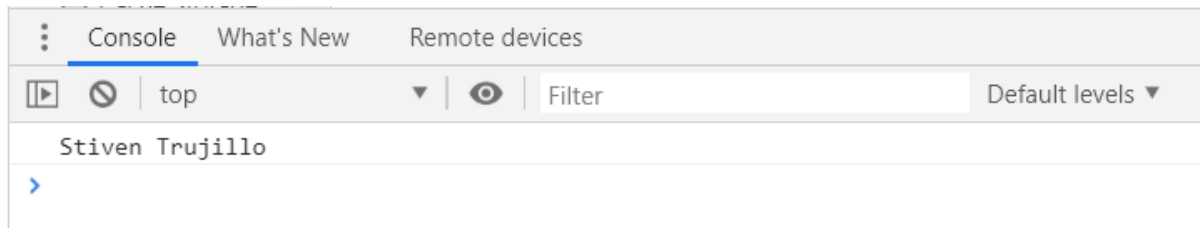


## Parámetros obligatorios de las Funciones

```
function nombreCompleto (nombre: string, apellido: string) {
    return nombre + ' ' + apellido;
}

let nombre = nombreCompleto("Stiven", "Trujillo");

console.log( nombre );
```

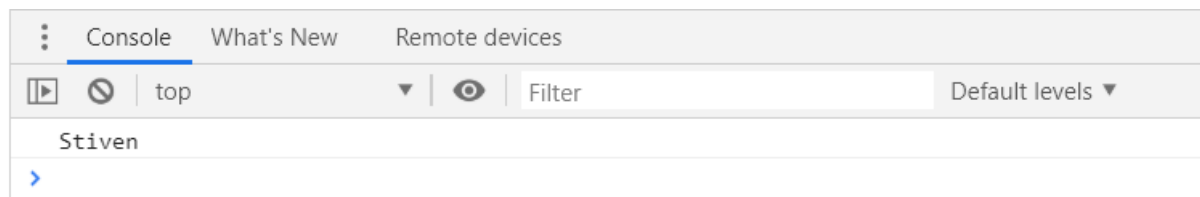


## Parámetros Opcionales de las Funciones

En JS todos los parámetros son opcionales. Pero en TS necesitamos ponerles un ?

```
function nombreCompleto (nombre: string, apellido?: string): string {  
  
    if ( apellido ) {  
        return nombre + ' ' + apellido;  
    } else {  
        return nombre;  
    }  
}  
  
let nombre = nombreCompleto("Stiven");  
  
console.log( nombre );
```

apellido es un parámetro opcional, y para evitar que nos retorne un undefined, hacemos un condicional que verifique si existe.



## Parámetros por Defecto

```
function nombreCompleto ( nombre: string,
                          apellido: string,
                          capitalizado: boolean = true): string {

  if ( capitalizado ) {
    return capitalizar(nombre) + ' ' + capitalizar(apellido);
  } else {
    return `${nombre} ${apellido}`;
  }
}

function capitalizar ( palabra: string ): string {
  return palabra.charAt(0).toUpperCase() + palabra.substr(1).toLowerCase();
}

let nombre = nombreCompleto("stiven", "trujillo");

console.log( nombre );
```

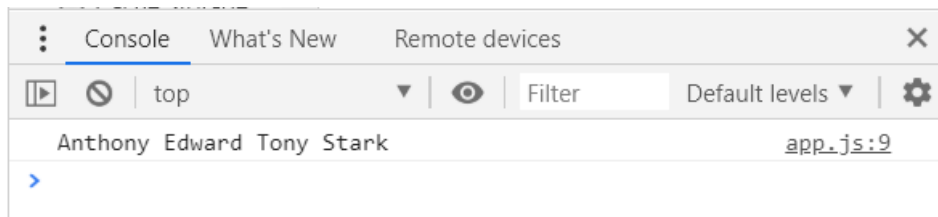
Vemos que solo le definimos un valor al parámetro para que este sea al valor por defecto al no ser ingresado en el llamado de la función.

## Parámetros REST

problema: Tenemos una función que debe ser capaz de recibir n cantidad de parámetros, los que sean necesarios

que es: Todo lo que viene por parámetro se junta en un arreglo.

```
function nombreCompleto( nombre:string,  
                        ...losDemasParametros: string[] ):string {  
    return `${nombre} ${losDemasParametros.join(" ")}`;  
}  
  
let iroman: string = nombreCompleto("Anthony", "Edward", "Tony", "Stark");  
  
console.log( iroman );
```



El parámetro nombre está definido como obligatorio para la función en este caso.

¿Cómo definimos ese tal parámetro REST? con la siguiente función: ...losDemasParametros

En el caso del ejemplo este es un arreglo de string.

join es una función de los arreglos de JS donde se le ingresa con que se quieren unir, en este caso con un espacio.

Note:

Pregunta 9:

¿Qué es un parámetro REST?

☒ Es un arreglo que contiene el resto de parametros enviados como argumentos a la función.

☐ Es un arreglo que contiene todos los parámetros enviados a la función.

## Tipo Función

```
() =>
```

donde lo que va dentro de los paréntesis son los parámetros, y lo que va después de => es el tipo de valor que retorna. ejemplo:

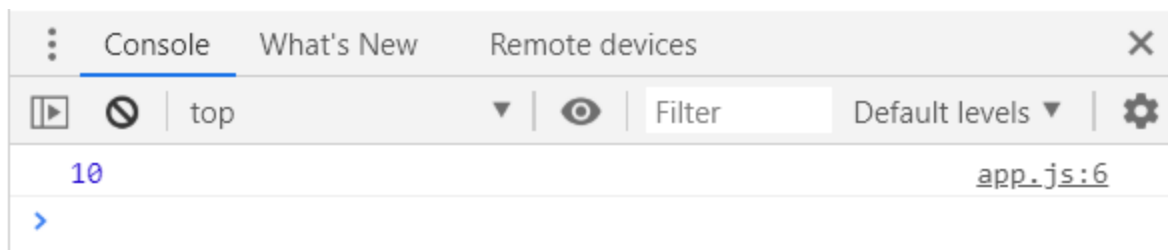
```
let miFuncion: (x: number, y: number) => number ;
```

lo que le estamos diciendo a TypeScript es que la variable miFuncion es de tipo función, y que tiene como parámetros X y Y que son de tipo number, y además retorna un valor de tipo number.

ejemplo de utilización:

```
function sumar(a:number, b:number):number {  
    return a + b;  
}  
  
let miFuncion: (x: number, y: number) => number ;  
  
miFuncion = sumar;  
  
console.log( miFuncion(5,5) );
```

note que la función a la que se iguala miFuncion, no tiene que tener el mismo nombre en los parámetros, pero sí deben ser la misma cantidad y el mismo tipo, al igual que tienen que tener el mismo tipo de valor de retorno.



## Objetos básicos

```
let flash = {  
  nombre: "Barry Allen",  
  edad: 24,  
  poderes: ["correr muy rápido", "viajar en el tiempo"]  
}
```

el definir este objeto de esta manera, y yo quiero re asignar los valores debo cumplir esta estructura.

el orden no afecta en los objetos.

## ¿Cómo crear objetos con tipos específicos?

```
let flash: { nombre:string, edad:number, poderes:string[] } = {  
  nombre: "Barry Allen",  
  edad: 24,  
  poderes: ["correr muy rápido", "viajar en el tiempo"]  
}
```

## Métodos dentro de los objetos

ES6 y TypeScript:

```
let flash: { nombre:string, edad:number, poderes:string[],  
getNombre():=>string } = {  
  nombre: "Barry Allen",  
  edad: 24,  
  poderes: ["correr muy rápido", "viajar en el tiempo"],  
  getNombre() {  
    return this.nombre;  
  }
```



```
    }  
}  
  
flash.getNombre();
```

ES5:

```
var flash = {  
  nombre: "Barry Allen",  
  edad: 24,  
  poderes: ["correr muy rápido", "viajar en el tiempo"],  
  getNombre: function () {  
    return this.nombre;  
  }  
};  
  
flash.getNombre();
```

## Tipos personalizado

Estoy creando una definición de un tipo.

la primera letra debe ser en mayúscula.

se debe utilizar type y no let.

```
type Heroe = {  
  nombre: string,  
  edad: number,  
  poderes: string[],  
  getNombre: () => string  
}  
  
let flash: Heroe = {  
  nombre: "Barry Allen",
```

```
edad: 24,  
poderes: ["correr muy rápido", "viajar en el tiempo"],  
getNombre() {  
    return this.nombre;  
}  
}
```



¡Buen trabajo!

No, los tipos sólo existen en TypeScript para brindarnos control sobre los objetos.

Pregunta 10:

¿Los tipos son traducidos a JavaScript?



Verdadero



Falso

## Múltiples tipos permitidos

la unión de tipos.

pueden hacer cuantas uniones ustedes deseen.

```
let loquesea: string | number | Heroe = "Fernando";  
  
loquesea = 10;
```

## Revisar el tipo de una objeto o variable

llegará un momento en el que estemos trabajando mucho con los múltiples tipos o bien las utilizaciones del tipo any, que ocupamos saber el tipo de dato, antes de hacer alguna operación.

Utilizaremos la instrucción de JS typeof, el cual nos regresa un string con el tipo de dato:

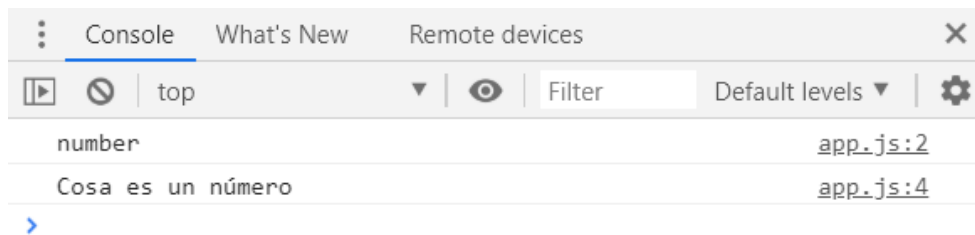
```

let cosa: any = 123;

console.log(typeof cosa);

if ( typeof cosa === "number" ) {
    console.log( "Cosa es un número" );
} else {
    console.log( "Cosa no es un número" );
}

```



## Depurar código de TypeScript

En el archivo tsconfig cambiar el valor de sourceMap

`{...}` tsconfig.json ▶ `{}` compilerOptions

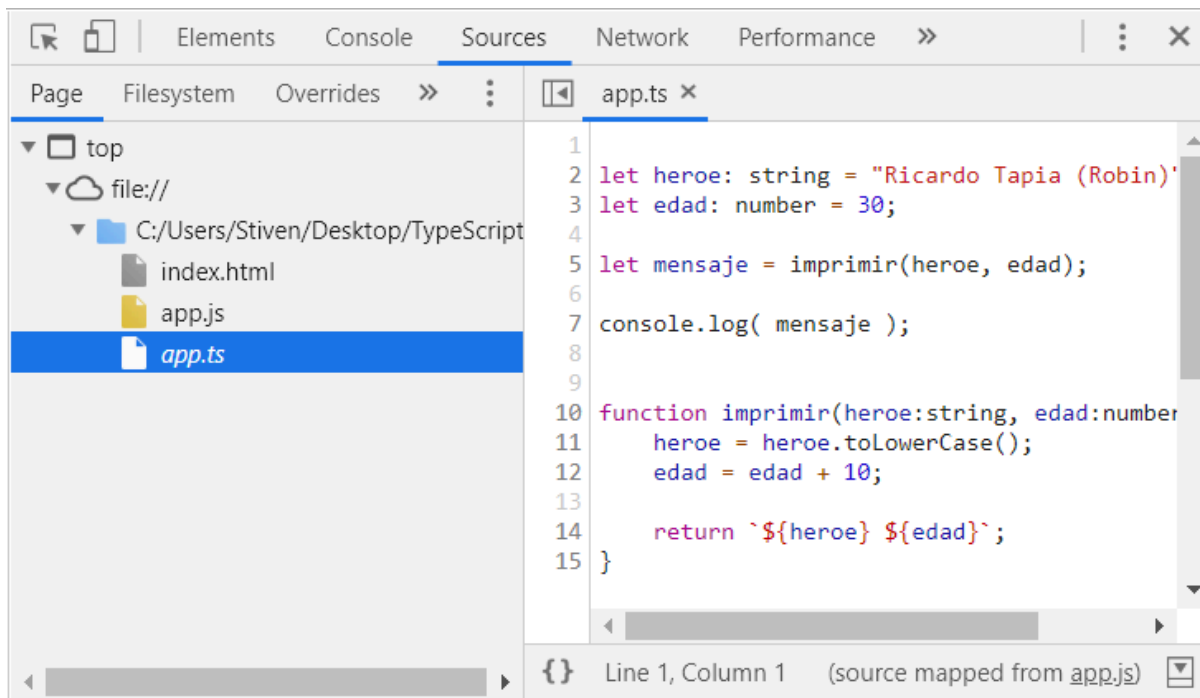
```

1  {
2      "compileOnSave": true,
3      "compilerOptions": {
4          /* Basic Options */
5          "target": "es5",                /* Specify ECMAScript
6          "watch": true,
7          "module": "commonjs",          /* Specify module co
8          // "lib": [],                  /* Specify library f
9          // "allowJs": true,           /* Allow javascript
10         // "checkJs": true,            /* Report errors in
11         // "jsx": "preserve",          /* Specify JSX code
12         // "declaration": true,        /* Generates corresp
13         // "declarationMap": true,     /* Generates a sourc
14         "sourceMap": true,             /* Generates correspond

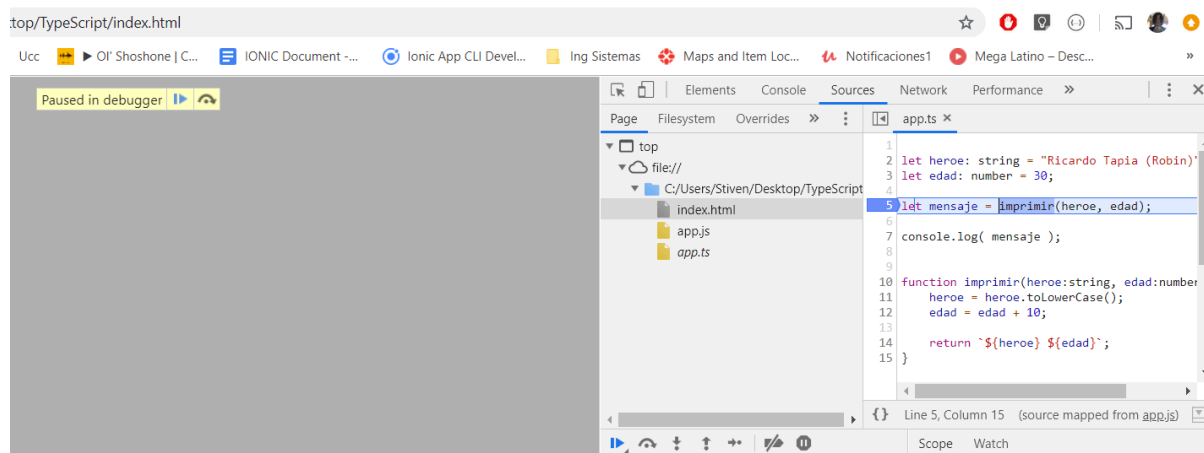
```

guardas el archivo.

ahora al compilar el archivo de TS, se creará un archivo .js.map y al ejecutarlo en el navegador podremos ver el archivo .ts en la opción de Source:



puedo hacer un breakpoint o punto de quiebre dando clic en el número de la línea, lo que hará que al ejecutarse se pause justo ahí.



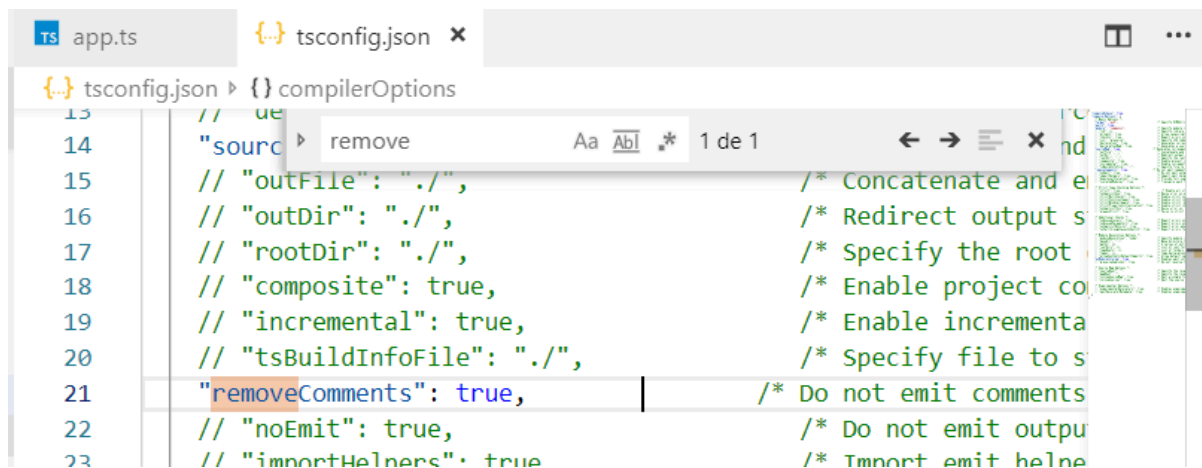
abajo veremos uno botones de acciones para la depuración:



## Remover los comentarios al compilar

En el archivo tsconfig, buscamos el comando removeComments, lo descomentamos y ponemos en true.

de esta manera, todos los comentarios que hayamos hecho en nuestro archivo de TS, no aparecerán o estarán en el archivo de JS al compilar, sería como un ignoreme los comentarios.



también podemos hacerlo por consola con el comando:

```
tsc archivo.ts --removeComments
```

**Note:** si tenemos el archivo tsconfig podemos omitir el nombre del archivo y solo poner tsc --removeComments.

Si al tener esta opción activada queremos que un comentario no sea ignorado lo ponemos con un /\*!

# Características de ES6

## Características:

- Mejora en el ámbito de las variables
- Adición de constantes
- Templates Literales
- Funciones de flecha (lambda)
- Desestructuración de objetos y arreglos
- Nuevos ciclos
- Clases
- Entre otras cosas

## Variables Let

Declarando variables con la palabra reservada let

podemos utilizar let en todos los lugares donde estaba anteriormente var

observemos este ejemplo:

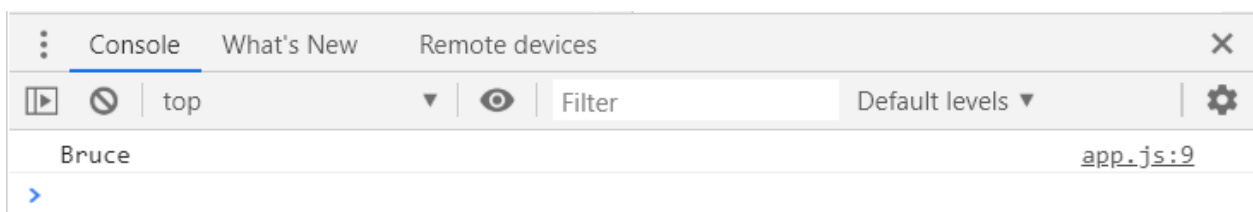
```
var nombre = "Tony";

if (true) {

    var nombre = "Bruce";

}

console.log(nombre);
```



Pero si utilizamos let:

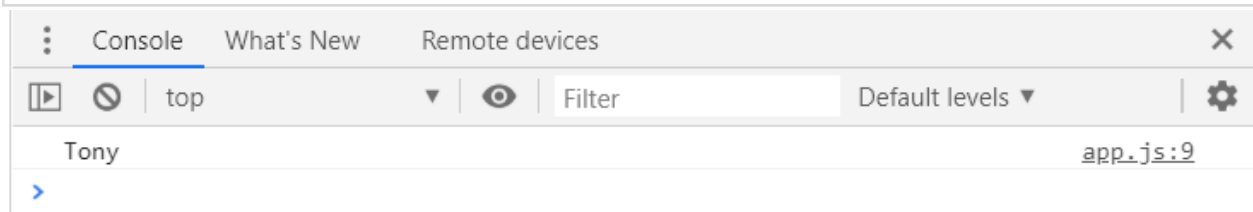
```
let nombre = "Tony";

if (true) {

    let nombre = "Bruce";

}

console.log(nombre);
```



¿por qué no salió Bruce?, al crear una variable let la crea en un ámbito específico, es decir, si yo creo la variable en el scope global y yo entro y creo una variable dentro del if, no va a poner la variable en el scope global de nuevo, simplemente va a crearse un ámbito especial. Estas llaves (del if) van a decir que hasta ahí van a vivir las variables que ustedes pongan allí adentro.

esto es particularmente útil porque podrían usar el mismo nombre de variable sin tener ningún error.

Otra característica de let, es que ustedes no pueden re declarar variables en el mismo ámbito

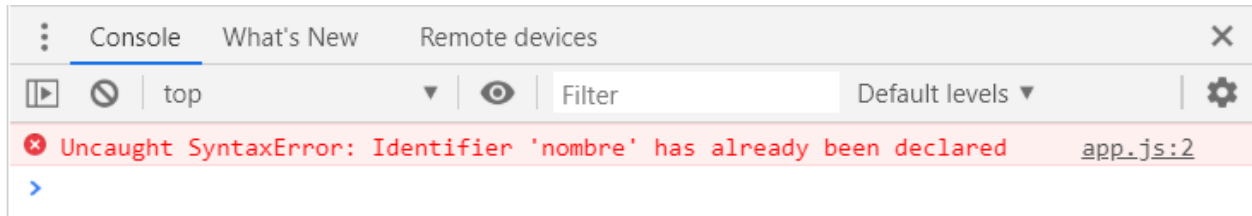
```
let nombre = "Tony";
let nombre = "Stiven";

if (true) {

    let nombre = "Bruce";

}
```

```
console.log(nombre);
```



Pero ustedes si pueden utilizar el mismo nombre si están en diferentes scopes.

## Constantes - const

El ES6 incorpora lo que son las constantes, que es un dato que no puede mutar una vez definido.

Es una convención de que las constantes todas van en mayúsculas.

```
const OPCIONES: string = "Activo";
```

**Note los siguientes casos:**

```
3  const OPCIONES: string = "Activo";  
4  
5  OPCIONES = "Desactivado";
```

vemos como no es permitido asignar otro valor a la constante

```
3  const OPCIONES: string = "Activo";  
4  
5  if( true ){  
6    |    const OPCIONES: string = "Desactivado";  
7  }
```

pero... ¿por qué en el caso anterior si es posible?, porque estamos en un diferente scope, aquí es otro ámbito totalmente de la variable, ahí si lo permite.



```
9  for ( const I of [1,2,3,4,5,6] ){  
10  |   console.log(I);  
11  }
```

vemos que este caso también es permitido a pesar de que la constante en teoría está cambiando de valor, realmente cuando se hace un ciclo for en el ES6 siempre va creando un nuevo ámbito por cada scope, por eso es que lo permite.

otra cosa que deben tener en cuenta es con los objetos:

```
const OPCIONES = {  
  estado: true,  
  audio: 10,  
  ultima: "main"  
}  
  
OPCIONES.estado = false;  
OPCIONES.audio = 1;  
  
console.log( OPCIONES );
```

veremos que si me permite cambiar las propiedades de los objetos.

Note:



¡Buen trabajo!

El IF, crea un nuevo scope o ámbito de la variable, por lo que si es válido.

Pregunta 2:

¿El siguiente código es válido?

```
1  const numero:number = 10;  
2  
3  if( numero >0 ){  
4  
5    const numero:number = 10;  
6  
7  }
```

☒ Verdadero

## Templates Literales

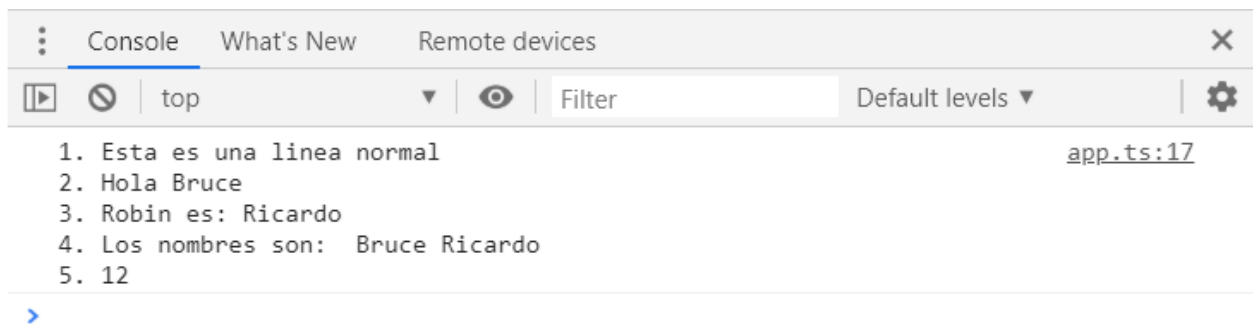
Son Strings que soportan multi línea, y permiten incrustar variables o el producto de funciones dentro del mismo String.

```
let nombre1:string = "Bruce";
let nombre2:string = "Ricardo";

function getNombres():string{
    return `${nombre1} ${nombre2}`;
}

let mensaje:string = `1. Esta es una línea normal
2. Hola ${ nombre1 }
3. Robin es: ${nombre2}
4. Los nombres son:  ${ getNombres() }
5. ${ 5 + 7 }
`;

console.log( mensaje );
```



## Funciones Flecha

cómo representamos una función normal a una de flecha:

función normal:

```
function sumar(a,b){  
    return a + b;  
}  
  
console.log( sumar(2,2) );
```

**Función flecha:**

```
let sumar = (a,b) => a+b;  
  
console.log( sumar(2,2) );
```

si tuviéramos más líneas, podríamos abrir llaves, y dentro lo que equivale al cuerpo de la función:

```
let sumar = (a,b) => {  
    console.log("vamos a sumar!");  
    return a + b;  
};
```

pero cuando es solo una instrucción basta con hacerlo de la primera forma.

**Otro ejemplo:**

```
function darOrden_hulk( orden ){  
    return `Hulk ${orden}`;  
}  
  
console.log( darOrden_hulk("smash!!!") );
```

**convirtamos esta función a una función de flecha:**

```
let darOrden_hulk = ( orden ) => `Hulk ${orden}`;  
  
console.log( darOrden_hulk("smash!!!") );
```

**¿Cual es la ventaja de utilizar las funciones de flecha?**

miremos este ejemplo donde las funciones de flecha si son útiles.

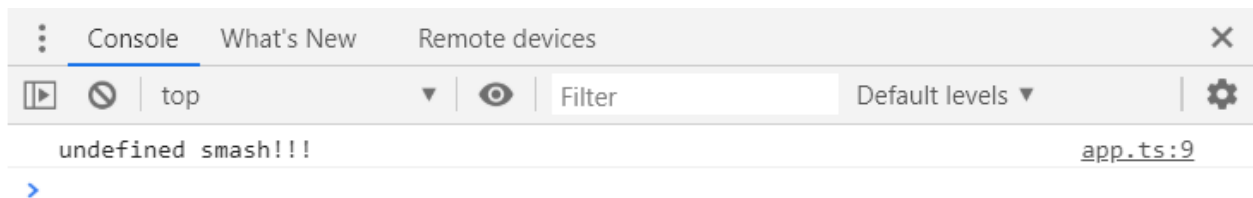
```
let capitan_america = {
  nombre: "Hulk",
  darOrden_hulk: function() {

    setTimeout(function() {

      console.log( this.nombre + " smash!!!" );
    }, 1000);
  }
}

capitan_america.darOrden_hulk();
```

supongamos que hulk es un poco lento de la cabeza y la orden que recibe la ejecuta un segundo después (setTimeout).



al segundo aparece nuestro mensaje, pero fíjense que es undefined smash!!!, ¿por qué undefined?

cuando usemos funciones asíncronas o trabajamos con cosas como el ejemplo que crea un nuevo contexto de una función dentro de una función, el puntero del this (en this.nombre) se pierde o apunta a otra cosa, en este caso está apuntando al objeto global. Este código ya no funciona.

entonces utilicemos la función de flecha para evitar que this mute, las funciones de flecha no cambian el objeto this, en otras palabras el this dentro de la función de flecha va a ser el mismo this del contexto anterior.

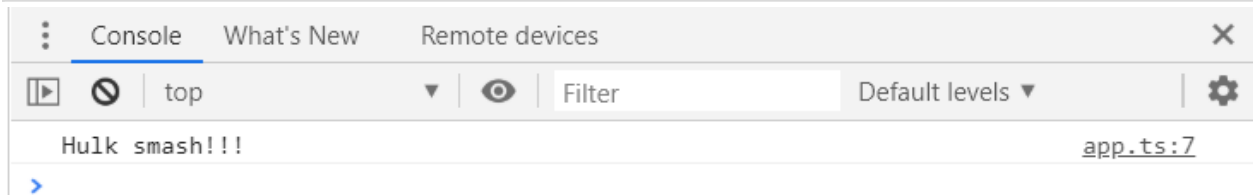
```
let capitan_america_ = {
  nombre: "Hulk",
  darOrden_hulk: function() {
```

```

        setTimeout( ()=>console.log( this.nombre + " smash!!!" ), 1000);
    }
}

capitan_america.darOrden_hulk();

```



**note:** como solo realiza una sola instrucción no hace falta ponerle llaves.

## Desestructuración de Objetos

suponga que necesita extraer los avenger dentro de ese objeto:

```

let avengers = {
    nick: "Samuel Jackson",
    ironman: "RObert Downey Jr",
    vision: "Paul Bettany"
}

let nick = avengers.nick;
let ironman = avengers.ironman;
let vision = avengers.vision;

```

miren todo ese código para extraer las variables de mi objeto

la desestructuración de objetos me permite hacer todo en una sola línea.

```

let avengers = {
    nick: "Samuel Jackson",
    ironman: "RObert Downey Jr",
    vision: "Paul Bettany"
}

```

```
}  
  
let { nick, ironman, vision } = avengers;
```

primero utilizo la palabra reservada let para hacer el espacio de memoria. Luego llaves, las llaves es para decir que quiero crear, en este caso quiero crear las variables nick, ironman y vision. ¿Pero de dónde voy a extraer esa data?, ponemos el igual y lo extraemos del objeto avengers. Eso es todo.

## Nuevo Ciclo - For Of

Básicamente nos permite barrer algún arreglo, por decirlo así, y nos ayuda a que el ciclo sea más corto y resumido.

```
let thor = {  
  nombre: "Thor",  
  arma: "Mjolnir"  
};  
  
let ironman = {  
  nombre: "Ironman",  
  arma: "Armorsuit"  
};  
  
let capitan = {  
  nombre: "Capitán América",  
  arma: "Escudo"  
};  
  
let avengers = [thor, ironman, capitan];
```

si yo quisiera barrer todas las personas que están dentro de ese arreglo

en ES5:

```
for ( let i in avengers ){
```

```
    let avenger = avengers[i];
    console.log( avenger.nombre, avenger.arma );
}

// tradicional
for ( let i = 0; i <= avengers.length - 1; i++ ){
    let avenger = avengers[i];
    console.log( avenger.nombre, avenger.arma );
}
```

en ES6 incorporaron el nuevo ciclo, el for of:

```
for ( let avenger of avengers ){
    console.log( avenger.nombre, avenger.arma );
}
```

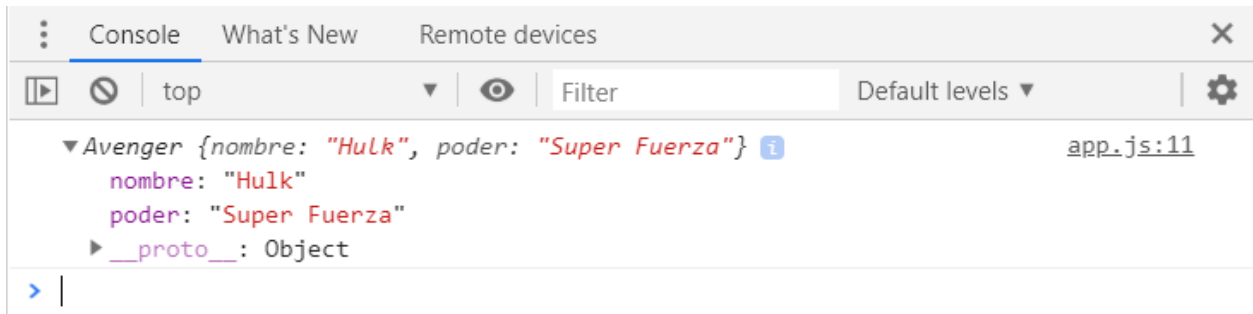
## Clases en ES6

```
class Avenger {

    constructor(nombre, poder) {
        this.nombre = nombre;
        this.poder = poder;
    }
}

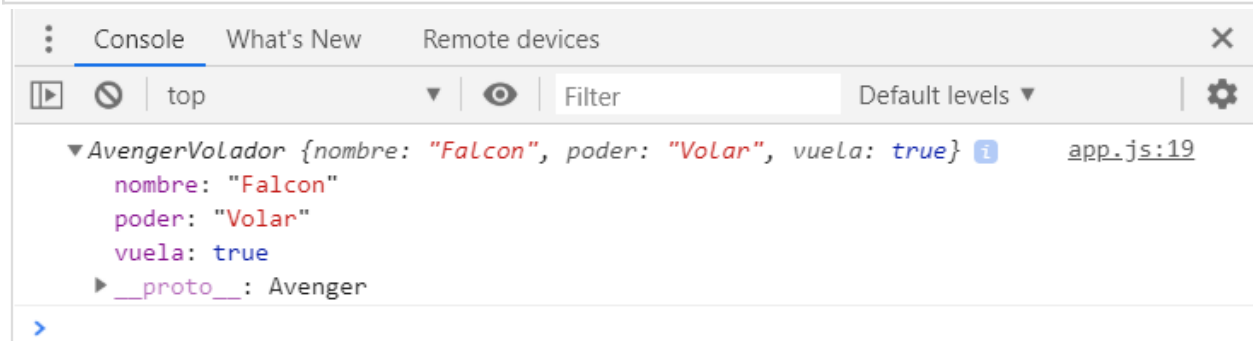
let hulk = new Avenger("Hulk", "Super Fuerza");

console.log(hulk);
```



Es posible en las clases heredar las características a otra clase hija.

```
class Avenger {  
  
  constructor(nombre, poder) {  
    this.nombre = nombre;  
    this.poder = poder;  
  }  
}  
  
class AvengerVolador extends Avenger{  
  
  constructor(nombre, poder){  
    super( nombre, poder );  
    this.vuela = true;  
  }  
}  
  
let falcon = new AvengerVolador( "Falcon", "Volar" );  
  
console.log( falcon );
```





## Desestructuración de Arreglos

```
// Desestructuración de arreglos?  
let versiones = ["Spider-Man 2099", "Spider-Girl", "Ultimate Spider-Man"];  
  
let spiderman2099 = versiones[0];  
let spidergirl = versiones[1];  
let ultimate = versiones[2];
```

de la siguiente forma:

```
let versiones = ["Spider-Man 2099", "Spider-Girl", "Ultimate Spider-Man"];  
  
let [ spiderman2099, spidergirl, ultimate ] = versiones;
```

# Clases en TypeScript

## Definición de una clase básica en TypeScript

```
class Avenger {  
  
    nombre:string = "sin nombre";  
    equipo: string;  
    nombreReal:string;  
  
    puedePelear:boolean;  
    peleasGanadas:number;  
  
}  
  
let antman:Avenger = new Avenger();
```

## Constructores

Un constructor es una simple función que es ejecutada cuando se crea una nueva instancia de ese objeto.

```
class Avenger {  
  
    nombre:string = "sin nombre";  
    equipo: string = undefined;  
    nombreReal:string = undefined;  
  
    puedePelear:boolean = false;  
    peleasGanadas:number = 0;  
  
    constructor( nombre:string, equipo:string, nombreReal:string ){  
        this.nombre = nombre;  
    }  
}
```

```

        this.equipo = equipo;
        this.nombreReal = nombreReal;
    }

}

let antman:Avenger = new Avenger("Antman", "cap team", "Scott Lang");

console.log(antman);

```

el constructor del TypeScript es mucho más poderoso que el constructor del ES6.

inicializar las propiedades con un valor por defecto puede hacerse perfectamente de esta forma.

el this va a hacer referencia siempre a la clase.

## Propiedades públicas, privadas y protegidas

En JavaScript todas las propiedades son públicas, pero en TypeScript podemos dar ese nivel de control.

por defecto si yo no especificó el tipo, es público.

```

class Avenger {

    public nombre:string = "sin nombre";
    protected equipo: string = undefined;
    private nombreReal:string = undefined;

    private puedePelear:boolean = false;
    private peleasGanadas:number = 0;

    constructor( nombre:string, equipo:string, nombreReal:string ){
        this.nombre = nombre;
        this.equipo = equipo;
        this.nombreReal = nombreReal;
    }
}

```

```

    }

}

20 let antman:Avenger = new Avenger("Antman", "cap team", "Scott Lang");
21
22 antman.
23     nombre (property) Avenger.nombre: string
24

```

## Métodos públicos, privados y protegidos.

```

class Avenger {

    public nombre:string = "sin nombre";
    protected equipo: string = undefined;
    private nombreReal:string = undefined;

    private puedePelear:boolean = false;
    private peleasGanadas:number = 0;

    constructor( nombre:string, equipo:string, nombreReal:string ){
        this.nombre = nombre;
        this.equipo = equipo;
        this.nombreReal = nombreReal;
    }

    public bio():void {
        let mensaje:string = `${this.nombre} ${this.nombreReal} ${this.equipo}`;
        console.log(mensaje);
    }

    public cambiarEquipoPublico(nuevoEquipo:string):boolean{
        return this.cambiarEquipo(nuevoEquipo);
    }
}

```

```

    private cambiarEquipo(nuevoEquipo:string):boolean{

        if( nuevoEquipo === this.equipo ){
            return false;
        }else {
            this.equipo = nuevoEquipo;
            return true;
        }

    }

}

let antman:Avenger = new Avenger("Antman", "cap team", "Scott Lang");

console.log( antman.cambiarEquipoPublico("cap team") );

```

## Herencia, super y definición de propiedades en el constructor

```

class Avenger{

    constructor( public nombre:string, private nombreReal:string ){
        console.log("Constructor Avenger llamado");
    }

    protected getNombre():string{
        console.log("get nombre avenger (protegido)");
        return this.nombre;
    }

}

class Xmen extends Avenger{

```

```

    constructor( a:string, b:string ){
        console.log("Constructor Xmen llamado");
        super(a,b) ;
    }

    public getNombre():string{
        console.log("get nombre xmen (público)");
        return super.getNombre() ;
    }
}

let ciclope:Xmen = new Xmen("Ciclope", "Scott");

console.log( ciclope.getNombre() );

```

⋮	Console	What's New	Remote devices	✕
▶	🔇	top	▼   🔍   Filter	Default levels ▼   ⚙️
	Constructor Xmen llamado			<a href="#">app.ts:19</a>
	Constructor Avenger llamado			<a href="#">app.ts:6</a>
	get nombre xmen (publico)			<a href="#">app.ts:24</a>
	get nombre avenger (protegido)			<a href="#">app.ts:10</a>
	Ciclope			<a href="#">app.ts:31</a>
>				

## Gets y Sets

es una forma de acceder a las propiedades de una manera controlada.

todos los gets y los sets deberían de ser públicos.

el get por defecto tiene que regresar algo, el void no es permitido.

definir una variable con un guión bajo es una convención para definir que la variable (propiedad) es privada y sólo se usará dentro de la clase.

para hacer el llamado de esta función, no esperen hacerlo como un método:

```

class Avenger {

    constructor( private _nombre:string ) {

    }

    get nombre():string{
        return this._nombre
    }
}

let ciclope:Avenger = new Avenger("Ciclope");

console.log( ciclope.nombre );

```

vamos a verificar si la propiedad tiene un valor:

```

class Avenger {

    constructor( private _nombre?:string ) {

    }

    get nombre():string{
        if ( this._nombre ){
            return this._nombre
        }else {
            return "No tiene un nombre el avenger"
        }
    }
}

let ciclope:Avenger = new Avenger();

console.log( ciclope.nombre );

```

tengo una forma de controlar cuando alguien ejecute mi .nombre

con los sets es lo mismo, por defecto estos no regresan nada, aunque ustedes pueden regresar si lo hizo o no lo hizo.

por lo general los sets agarran un parámetro y se lo establecen a una propiedad.

```
class Avenger {  
  
    constructor( private _nombre?:string ) {  
    }  
  
    get nombre():string {  
        return this._nombre;  
    }  
  
    set nombre(nombre:string){  
        this._nombre = nombre;  
    }  
}  
  
let ciclope:Avenger = new Avenger();  
  
ciclope.nombre = "Ciclope";  
  
console.log( ciclope.nombre );
```

validar el dato antes de asignarlo a la variable.

```
class Avenger {  
  
    constructor( private _nombre?:string ) {  
    }  
  
    get nombre():string {  
        return this._nombre;  
    }  
}
```



```

    set nombre(nombre:string){

        if( nombre.length <= 3 ){
            console.error("El nombre debe tener al menos 4 caracteres.");
            return;
        }
        this._nombre = nombre;
    }
}

let ciclope:Avenger = new Avenger();

ciclope.nombre = "Lee";

console.log( ciclope.nombre );

```

si el tamaño del nombre es menor o igual a 3 entonces hago un return, osea me salgo.

también se puede hacer de esta forma:

```

class Avenger {

    private _nombre:string;

    constructor( nombre?:string ) {
        this._nombre = nombre;
    }

    get nombre():string {
        return this._nombre;
    }

    set nombre(nombre:string){

        if( nombre.length <= 3 ){

```

```

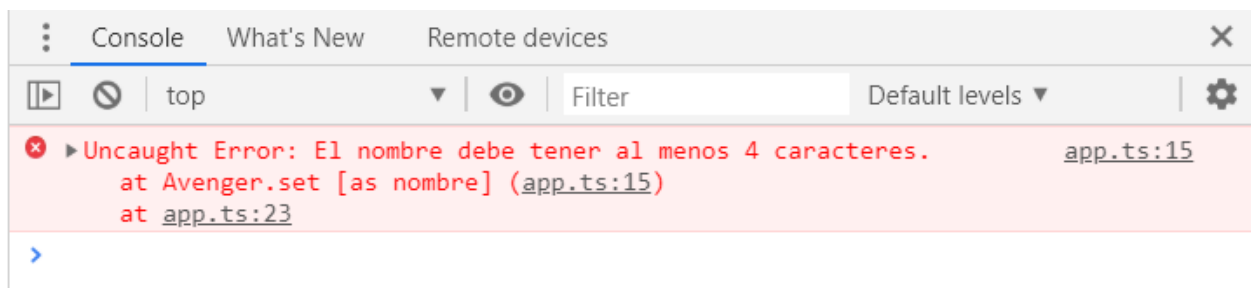
        throw new Error("El nombre debe tener al menos 4 caracteres.");
    }
    this._nombre = nombre;
}
}

let ciclope:Avenger = new Avenger();

ciclope.nombre = "Lee";

console.log( ciclope.nombre );

```



## Métodos y propiedades estáticos

```

class Xmen{

    public nombre:string = "Wolverine";

    constructor() {
    }

}

```

si yo quisiera acceder al nombre de Wolverine tendría que crear un instancia de la clase Xmen

```

let wolverine = new Xmen();

```

```
console.log( wolverine.nombre );
```

las propiedades o métodos estáticos se pueden llamar sin instanciar la clase :

```
class Xmen{

    static nombre:string = "Wolverine";

    constructor() {

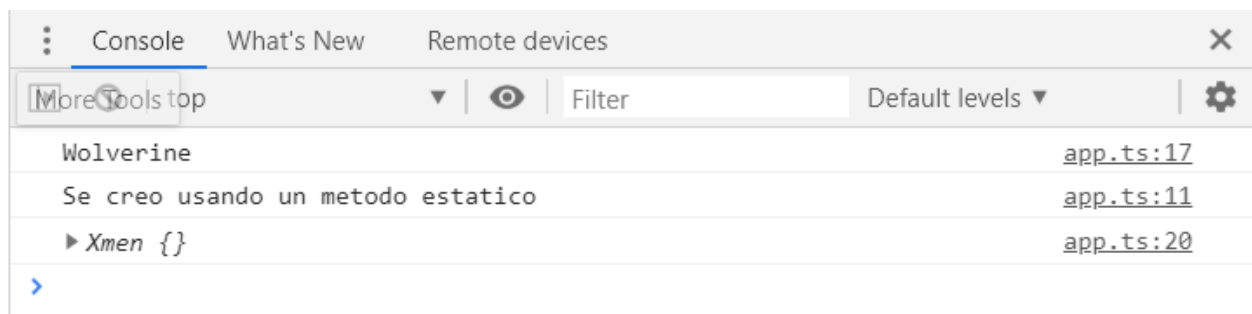
    }

    static crearXmen(){
        console.log("Se creó usando un método estático");
        return new Xmen();
    }

}

console.log( Xmen.nombre );

let wolverine = Xmen.crearXmen() ;
console.log( wolverine );
```



## Clases Abstractas

Cuando utilizo abstract no puedo crear instancias de la clase:

```

abstract class Mutantes{ // X-MEN

    constructor(public nombre:string, public nombreReal:string){}

}

```

```

6
7
8 }
9
10 let wolverine = new Mutantes("Wolverine", "Logan");

```

No se puede crear una instancia de una clase abstracta. ts(2511)

[Corrección Rápida](#) [Problema de pico](#)

Entonces esto para qué diablos sirve. Es utilizado para crear como un molde de las clases y heredarlo a otras clases.

```

abstract class Mutantes{ // X-MEN

    constructor(public nombre:string, public nombreReal:string){}

}

class Xmen extends Mutantes{

}

let wolverine = new Xmen("Wolverine", "Logan");

console.log( wolverine);

```

**note** como Xmen no tiene constructor definido utiliza el de Mutantes.

## Constructores privados



# Interfaces

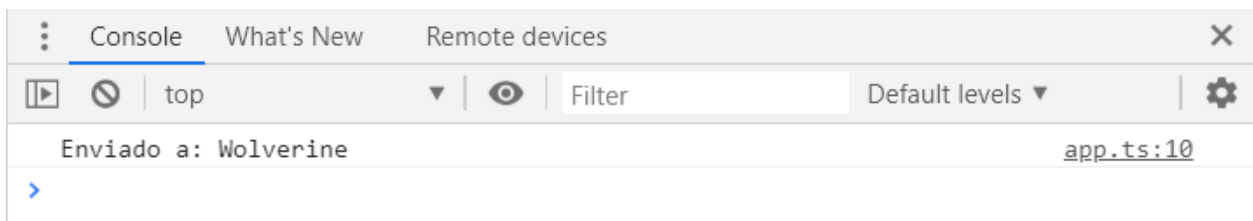
## Interfaz Básica

```
interface Xmen {  
    nombre: string,  
    poder: string  
}
```

### Con CamelCase

Y podremos utilizarla de la siguiente forma:

```
function enviarMision( xmen : Xmen ) {  
  
    console.log("Enviado a: " + xmen.nombre);  
  
}  
  
let wolverine:Xmen = {  
    nombre: "Wolverine",  
    poder: "Regeneración"  
};  
  
enviarMision( wolverine );
```

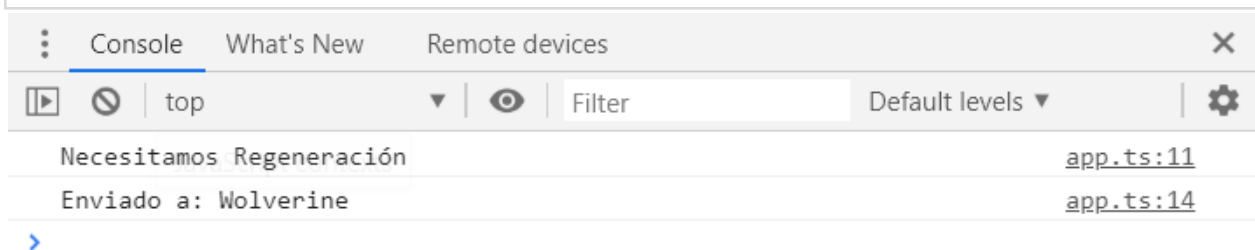


## Propiedades Opcionales

```
interface Xmen {  
    nombre: string,  
    poder?: string  
}
```

Poder es una propiedad opcional en este ejemplo.

```
function enviarMision( xmen : Xmen ) {  
  
    if(xmen.poder){  
        console.log(`Necesitamos ${xmen.poder}`);  
    }  
  
    console.log("Enviado a: " + xmen.nombre);  
  
}  
  
let wolverine:Xmen = {  
    nombre: "Wolverine",  
    poder: "Regeneración"  
};  
  
enviarMision( wolverine );
```



## Métodos en la Interfaz

Hay que especificar (si los tiene) los parámetros, y el retorno.

```
interface Xmen {  
    nombre: string,  
    regenerar( nombreReal:string ):void  
}
```

No estamos diciendo que es lo que tiene que hacer la función, puede ser lo que sea, pero recibe y regresa lo especificado.

El método también puede ser opcional agregando un signo de pregunta.

```
interface Xmen {  
    nombre: string,  
    regenerar?( nombreReal:string ):void  
}
```

### Ejemplo

```
function enviarMision( xmen : Xmen ) {  
  
    console.log("Enviado a: " + xmen.nombre);  
  
    xmen.regenerar("Logan") ;  
  
}  
  
let wolverine:Xmen = {  
    nombre: "Wolverine",  
    regenerar(x:string) {  
        console.log("Se ha regenerado " + x);  
    }  
};
```



```
enviarMision( wolverine );
```

## Interfaces en las Clases

Es posible utilizar las interfaces para definir los métodos y propiedades que va a tener una clase, es decir obligar a las clases a que tengan una cierta forma.

```
interface Xmen{
    nombre:string;
    nombreReal?:string;
    regenerar( nombreReal:string ):void;
}

class Mutante implements Xmen {
    nombre:string;
    poder:string;
    esBueno:boolean;
    regenerar( nombre ){
        console.log("hola " + nombre);
    }
}

let wolverine = new Mutante();
```

Note que podemos agregar nuevas propiedades y hasta métodos.

## Interfaces para las funciones

Es posible utilizar las interfaces para moldear funciones.

```
interface DosNumerosFunc{
    ( num1:number, num2:number ) : number
}
```

```
let sumar:DosNumerosFunc;

sumar = function(a:number,b:number) {
    return a + b;
}

let restar:DosNumerosFunc;

restar = function(numero1:number,numero2:number) {
    return numero1 - numero2;
}
```

Note que el código para definir la interfaz es muy parecido pero no es igual, a la forma vista en clases anteriores.

La diferencia es que aquí no vamos a definir propiedades, vamos a definir únicamente un método, solo uno.

## Notes

es posible heredar interfaces con la palabra `extends` :

```
interface Carro{
    llantas:number;
    modelo:string;
}

interface Volvo extends Carro{
    seguro:boolean;
}

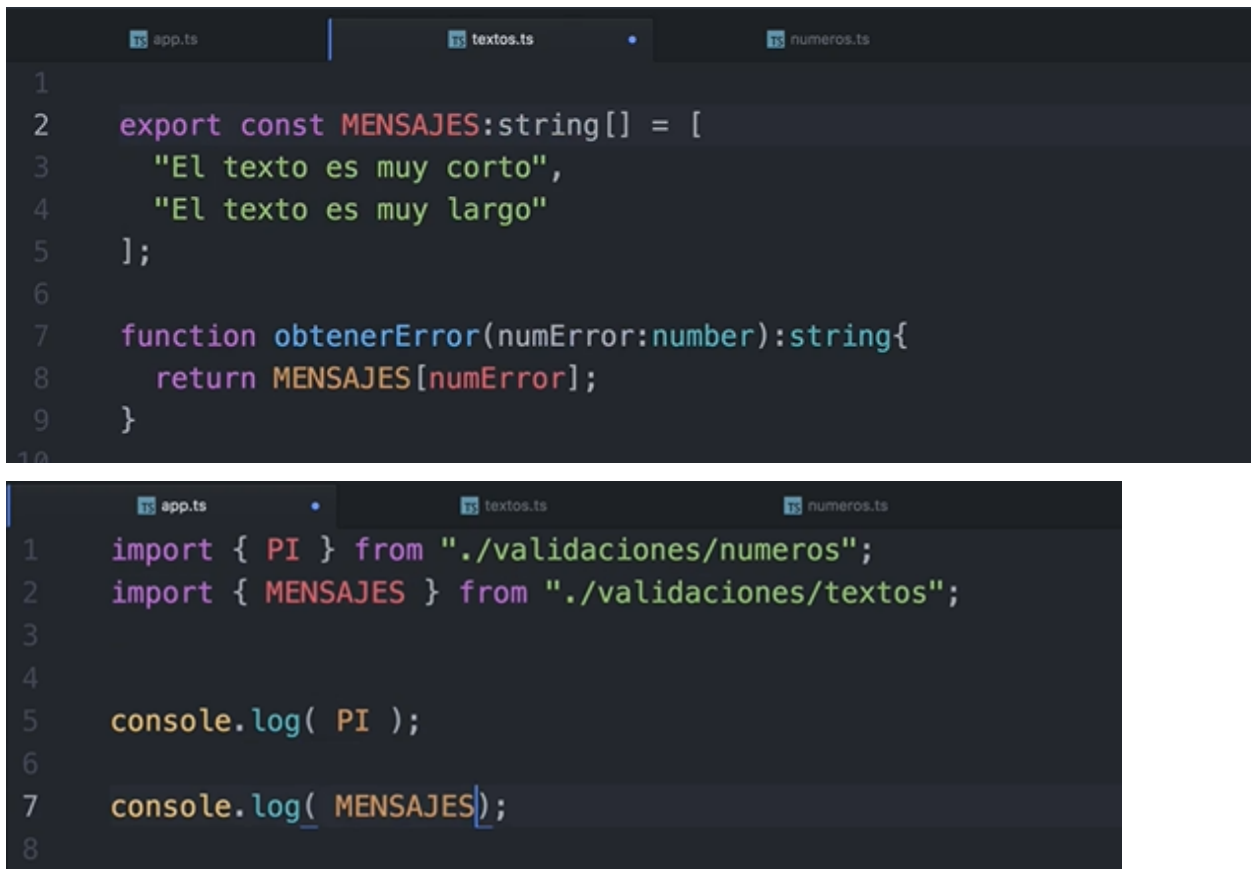
var volvo:Volvo = {
    llantas: 4,
    modelo:"sedan",
```

```
seguro:true  
}
```

## Módulos

TypeScript nos da la posibilidad de crear aplicaciones que permitan importar funciones, clases, propiedades, variables o constantes de otros archivos del proyecto sin necesidad de agregar las referencias en nuestras páginas HTML.

Los módulos es una forma de programar que brinda grandes funciones a la programación web.



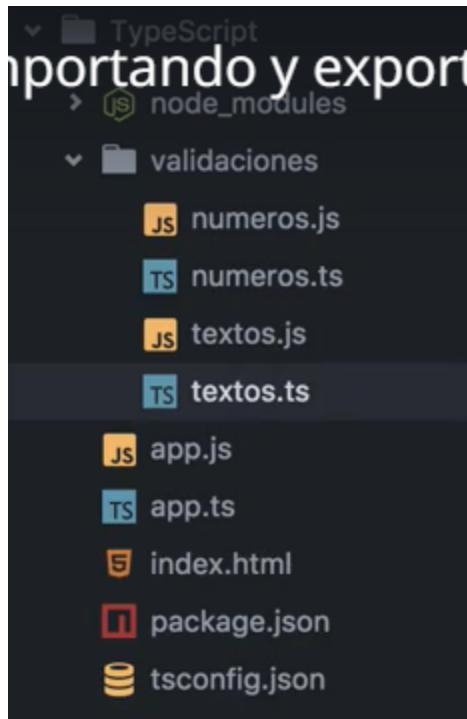
The image consists of two screenshots of a code editor. The top screenshot shows the 'textos.ts' file with the following code:

```
1
2 export const MENSAJES:string[] = [
3     "El texto es muy corto",
4     "El texto es muy largo"
5 ];
6
7 function obtenerError(numError:number):string{
8     return MENSAJES[numError];
9 }
10
```

The bottom screenshot shows the 'app.ts' file with the following code:

```
1 import { PI } from "../validaciones/numeros";
2 import { MENSAJES } from "../validaciones/textos";
3
4
5 console.log( PI );
6
7 console.log( MENSAJES);
8
```

de la raíz vayase a la carpeta validaciones y al archivo textos.



Para obtener todos los módulos (osea propiedades y métodos con la palabra export) del archivo textos:

```
1 import { PI } from "../validaciones/numeros";
2 import * as textos from "../validaciones/textos";
3
4
5 console.log( PI );
6
7 console.log( textos.obtenerError(0) );
8
```

import \* as textos from "../validaciones/textos";

para entender todo esto mejor, mirar los dos últimos videos de la sección de módulos.

# Genéricos - Generics

JavaScript por ser un lenguaje dinámico, conlleva a tener varios problemas por esa misma flexibilidad, pero a su vez, permite resolver problemas de una forma muy sencilla. Esta sección está destinada a comprender cómo mantener la programación estructurada del TypeScript con el dinamismo de JavaScript.

Puntualmente aprenderemos sobre:

1. Uso de los genéricos
2. Funciones genéricas
3. Ejemplos prácticos sobre los genéricos
4. Arreglos genéricos
5. Clases genéricas

## Introducción a los Genéricos

JavaScript no posee el concepto de genérico porque es totalmente dinámico, una función podría recibir cualquier argumento que ustedes deseen

los genéricos es algo que me va a permitir a mi trabajar con cualquier tipo de dato, y aprovechar las ventajas que tiene TypeScript para que me ayude con la autocompletación y posibles errores que yo pueda cometer a la hora de escribir o a la hora de manejar la data.

## Creando funciones genéricas

Teniendo la siguiente función:

```
function regresar( arg:any ){  
    return arg;  
}
```

voy a convertirla en una función genérica:


```
function regresar<T>( arg:T ){
    return arg;
}

console.log( regresar( 15.456789 ).toFixed(2) );
console.log( regresar("Stiven Trujillo").charAt(0) );
console.log( regresar( new Date() ).getTime() );
```

Es un estándar que nuestra función genérica tenga esa T, esta T puede ser cualquier cosa pero es una convención usar la T.

ya el argumento no va a ser de tipo any si no de tipo T, en otras palabras le estamos diciendo a TypeScript, que el tipo de retorno va a ser el mismo tipo que el argumento de entrada.

gracias a esto podremos obtener la ayuda de la autocompletación de TypeScript y podremos ver los posibles errores:



The screenshot shows a code editor with the following code:

```
function regresar<T>( arg:T ){
    return arg;
}

console.log( regresar( 15.456789 ). );
console.log( regresar("Stiven Trujillo"). );
console.log( regresar( new Date() ) );
```

Autocomplete suggestions for the first call site are shown:

- toExponential
- toFixed (method) Number.toFixed(fractionDigits)
- toLocaleString
- toPrecision
- toString
- valueOf

Below the code, two error messages are displayed:

- any
- La propiedad 'charAt' no existe en el tipo '15.456789'. ts(2339)

At the bottom, there are links for "Corrección Rápida" and "Problema de pico".

## Ejemplo de función genérica en acción

Tenemos una función genérica:

```
function functionGenerica<T>( arg:T ){  
    return arg  
}
```

y tenemos los siguientes tipos personalizados:

```
type Heroe = {  
    nombre:string;  
    nombreReal:string;  
}  
  
type Villano = {  
    nombre:string;  
    poder:string;  
}
```

Observe que el siguiente objeto puede ser de cualquiera de los dos tipos descritos arriba:

```
let deadpool = {  
    nombre: "Deadpool",  
    nombreReal: "Wade Winston Wilson",  
    poder: "regeneración",  
};
```

yo puedo especificarle al TypeScript que lo maneje como un Heroe o como un Villano.

para decirle a TypeScript que es un Heroe o Villano lo hacemos de la siguiente forma:

```
console.log(    functionGenerica<Heroe>( deadpool )    );
```

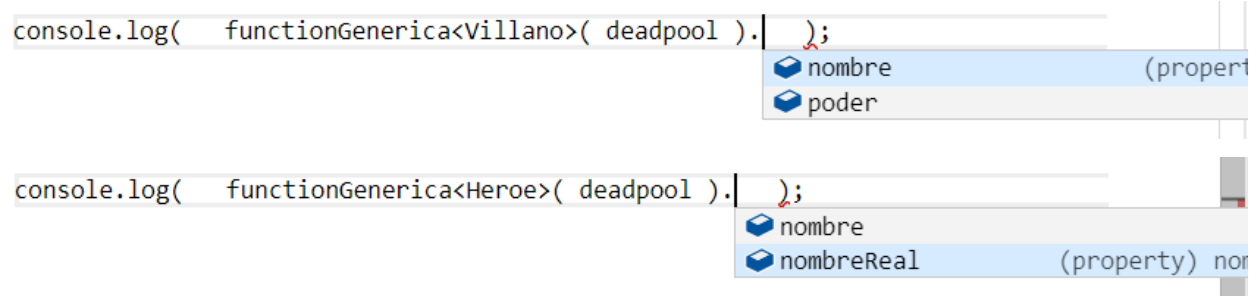
o

```
console.log(    functionGenerica<Villano>( deadpool )    );
```



mandando el tipo antes de los paréntesis.

note que solo me aparecerán las propiedades del tipo especificado:



```
console.log( functionGenerica<Villano>( deadpool ). );
console.log( functionGenerica<Heroe>( deadpool ). );
```

Con esto podemos tener o manejar una flexibilidad y todavía tener el control de TypeScript a la mano.

## Arreglos Genéricos

para crear un arreglo genérico basta con hacer el siguiente código:

```
let heroes: Array<string> = ["Flash", "Batman", "Superman"];
```

y este sería una array explícito de strings:

```
let villanos: string[] = ["Lex Luthor", "Flash Reverso"];
```

Básicamente es lo mismo, se quiere aclarar que se puede hacer de la manera genérica o de la manera explícita que hemos visto anteriormente.

## Clases genéricas

¿Cómo controlar el tipo que estoy enviando?

Mi función entre los tipos genéricos, solo puede recibir números o strings.

en la declaración también se puede definir cuál quieres que se acepte, en el ejemplo se define que acepte ambos tipos, mientras esté permitido en la definición de la clase.

por ejemplo si agrego un booleano, me generará un error por qué no está permitido en mi clase genérica.

ustedes pueden anticiparse que pueden ser números o pueden ser strings tanto en la clase como a la hora de instanciar la clase.

```
class Cuadrado<T extends number|string> {
  base:T;
  altura:T;
  area():number{
    return +this.base * +this.altura;
  }
}

let cuadrado = new Cuadrado<number|string>();

cuadrado.base = "10";
cuadrado.altura = 10;

console.log( cuadrado.area() );
```

el + actúa como casteo.

note los errores:

```
2  class Cuadrado<T extends number|string> {
3    base:T;
4    altura:T;
5    area():number{
6      return +this.base *
7    }
8  }
9
10 let cuadrado = new Cuadrado<number|string|boolean>();
```

El tipo 'string | number | boolean' no cumple la restricción 'string | number'.  
El tipo 'false' no se puede asignar al tipo 'string | number'. ts(2344)

[Corrección Rápida](#) [Problema de pico](#)

```
class Cuadrado<T extends number|string> {  
  base:T;  
  altura:T;  
  area():number{  
    return +this.base * +this.altura;  
  }  
}
```

(property) Cuadrado<string | number>.base: string | number

El tipo 'true' no se puede asignar al tipo 'string | number'. ts(2322)

[Corrección Rápida](#) [Problema de pico](#)

```
cuadrado.base = true;  
cuadrado.altura = 10;
```

# Decoradores

Los decoradores son una característica nueva en el TypeScript que cada vez es más utilizada por otros frameworks como Angular 2. Pero vamos a aprender a utilizar decoradores en nuestros proyectos.

## \*Introducción a los decoradores

Básicamente un decorador es una función.

Es una cosita que se pone antes de una función, antes de un método o una propiedad y nos va a permitir expandir su funcionalidad agregando cosas nuevas. Pueden hacer incluso mutar completamente la función.

## Decoradores de clases

Tenemos una clase sencilla:

```
class Villano {  
    constructor( public nombre:string ){  
  
    }  
}
```

Los decoradores recordemos que son una función. Los decoradores para las clases automáticamente al estar adjunto a una clase lo que hacen es enviar como parámetro el constructor de la clase:

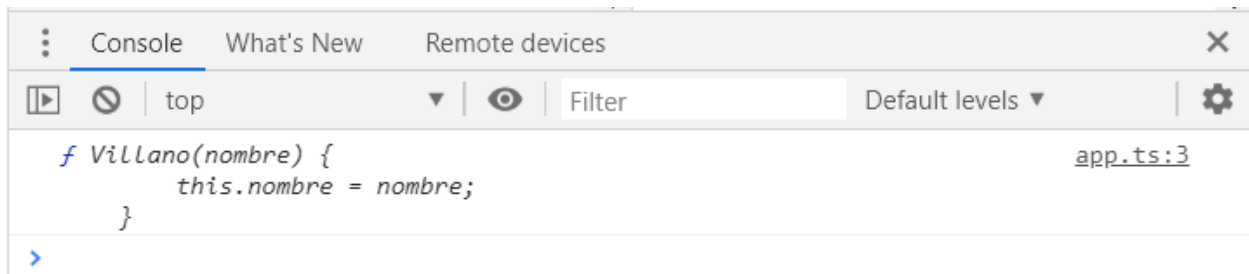
```
function consola( constructor:Function ){
    console.log( constructor );
}

@consola
class Villano {
    constructor( public nombre:string ){

    }
}
```

pero para que esto compile debemos activar la opción de nuestro tsconfig.json

```
"experimentalDecorators": true,
```



esto que vemos en consola es el constructor

## Decoradores de fábrica - Factory decorators

Un decorador de fábrica es una función común y corriente que puede recibir parámetros, pero debe regresar algo, que pueda ser utilizado como un decorador.

En este ejercicio sólo está imprimiendo la función constructor en la consola.

Qué pasaría si yo quisiera que mi función consola, imprima el constructor o no lo imprima, osea yo tengo que decirle si yo quiero imprimirlo o no quiero imprimirlo, aquí es donde viene el Factory Decorator.

mi decorador de fábrica será imprimirConsola, el cual recibe un booleano y regresa una función.

si imprimir es verdadero entonces quiero hacer el return de la función consola, pero sin los parentesis, no quiero ejecutar la función, solo quiero retornar la función que sirve como un decorador de clase o decorador de lo que quiera usar.

```
// Decorador de Clase
function consola( constructor:Function ){
    console.log( constructor );
}

// Decorador Factory
function imprimirConsola( imprimir:boolean ):Function {
    if ( imprimir ) {
        return consola;
    } else {
        return null;
    }
}

@imprimirConsola(true)
class Villano {
    constructor( public nombre:string ){

    }
}
```