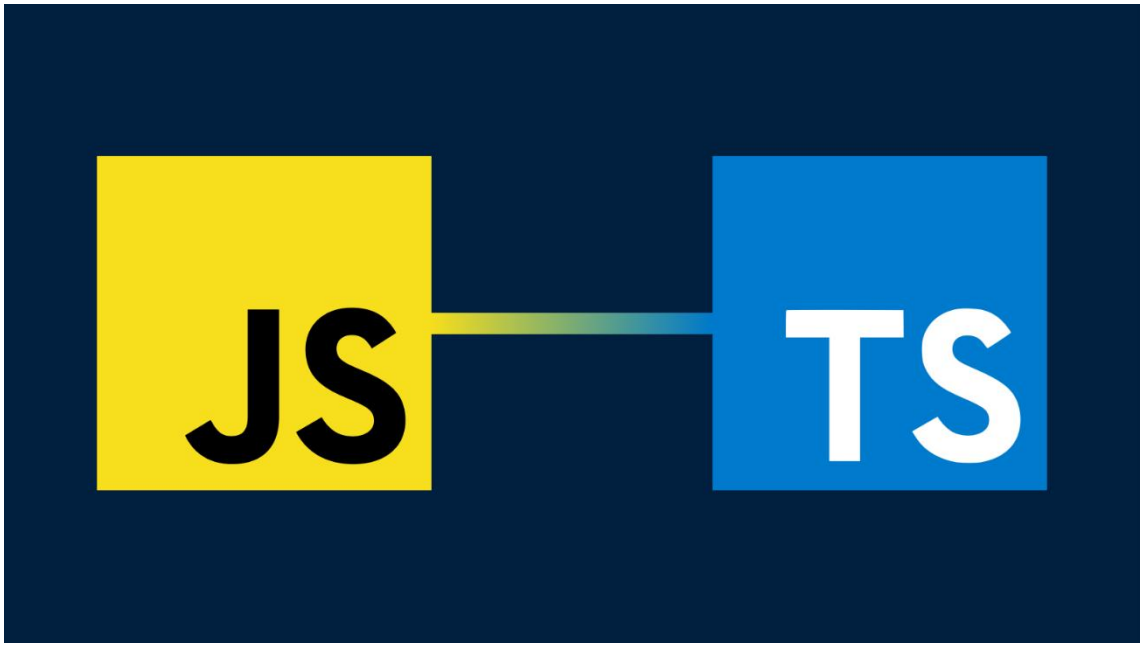
Introducción TypeScript



Jesús Rodríguez – Desarrollador Aplicaciones

INDICE

JAVASCRIPT CARECE DE MUCHAS COSAS

CARACTERÍSTICAS FALTANTES DE JAVASCRIPT

1. Tipos de variables.

   JavaScript maneja todo tipo de variables númericos, string, etc, pero no hay manera de saber que tipo de dato contiene una variable hasta que se le asigna el valor.

2. Errores en tiempo de escritura/compilación.

   No se puede detectar errores en tiempo de compilación/escritura ya que no lo tiene a pesar de ser una herramienta muy útil.

3. Auto completación dependiendo de la variable.

   Auto completación incompleta, debido a que según que variable sea tiene auto completación o no.

4. Método estático de programación.
5. Clases y módulos (antes de ES6).
6. Hay IDE en los que se puede manejar JavaScript, como por ejemplo NetBeans, eclipse, WS, etc, pero lo ideal sería un IDE especializado en JavaScript que nos indicará el control total indicando que un objeto tiene X propiedades o una clase x objetos y formado por x métodos, etc.

¿Qué problemas suceden en JavaScript?

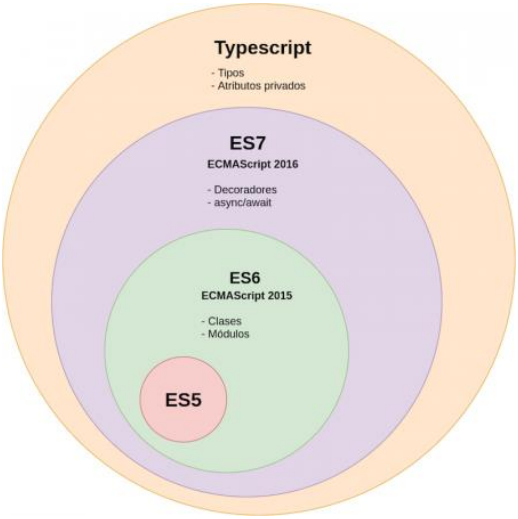
- Errores porque una variable no estaba definida.
- Errores porque el objeto no tiene la propiedad esperada.
- Errores porque no se tiene idea de como se trabajan las funciones de otros compañeros.
- Errores porque se sobre escriben variables, clases, funciones o constantes…
- Errores porque el código colisiona con el de otro.

- Errores porque el cache del navegador mantiene los archivos de JavaScript viejos.
- Errores porque colocamos una mayúscula o minúscula en el lugar incorrecto.
- Errores porque simplemente no sabemos como funciona el código de los demás.
- Errores porque el IDE no me dijo que eso no podía o debía hacerlo.

¡Lo peor de todos estos errores, es que no podemos detectarlos hasta que ejecutamos el proyecto!

TypeScript

Es un super set de JavaScript, para comprender dicha magnitud centrémonos en lo siguiente



ES → Es lo que conocemos como JavaScript (Declaración de variables, funciones, métodos, etc).

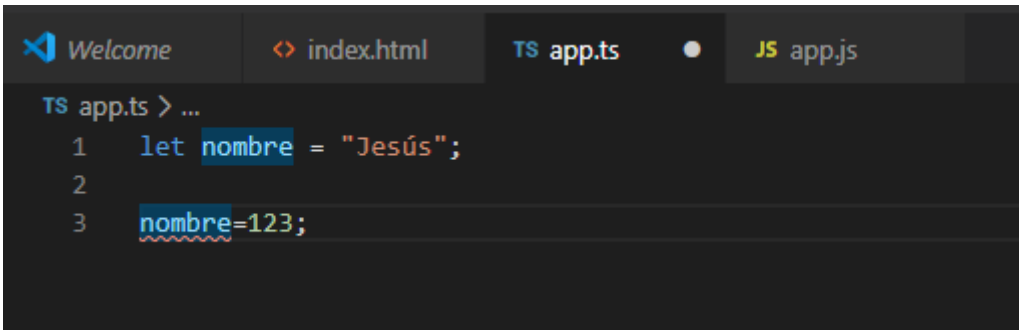
ES6, ES7 → Incorporó funciones de flechas, promesas, un nuevo tipo de declaración de variables, etc.

TypeScript (Creado por Microsoft) → TypeScript engloba todo lo anterior, y además le añade funcionalidades, métodos, propiedad, formas estáticas de programación, etc. Aunque hay algunas propiedades de ES6 que aún no son soportadas por TypeScript, por ejemplo:

- Maps y Set.

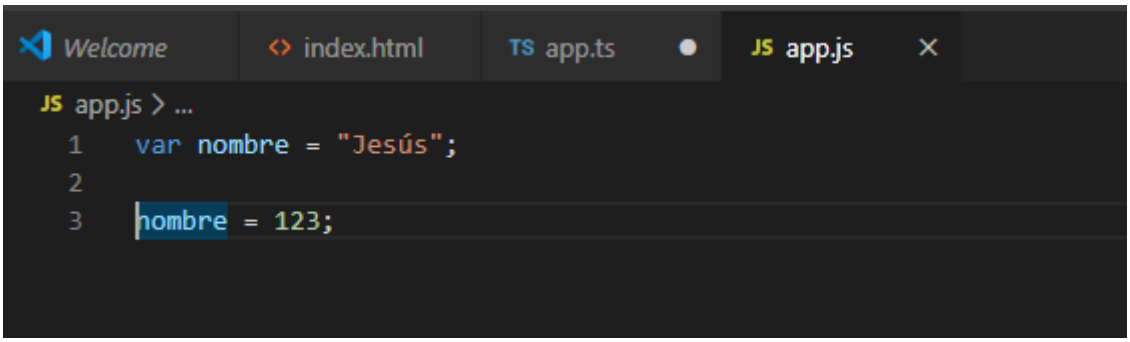
A continuación, podemos ver unos ejemplos que diferencian TypeScript de JavaScript:

1. Declaración de variables

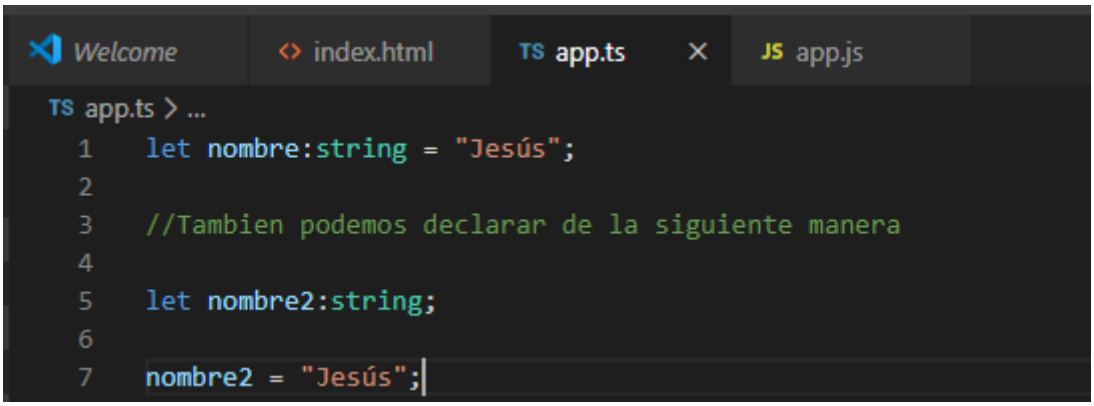


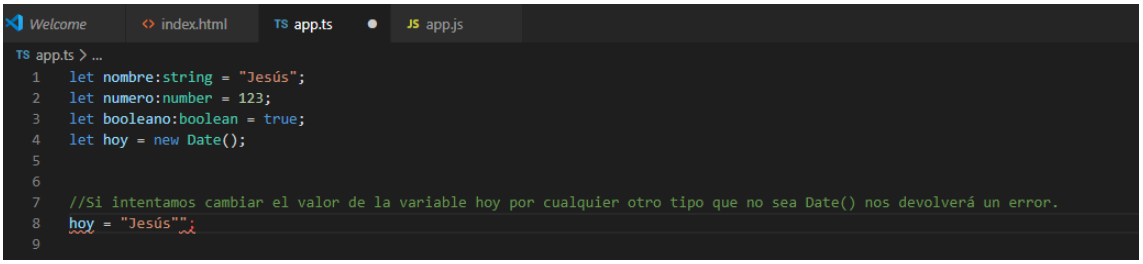
Como podemos ver hemos declarado una variable de tipo String, por lo que si intentamos insertar un valor numérico posteriormente nos va a dar un error.

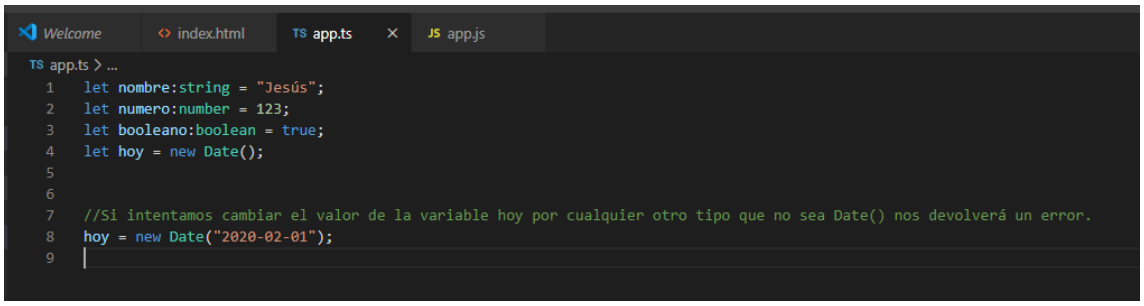
Sin embargo, si codificamos en JavaScript esto no pasa.



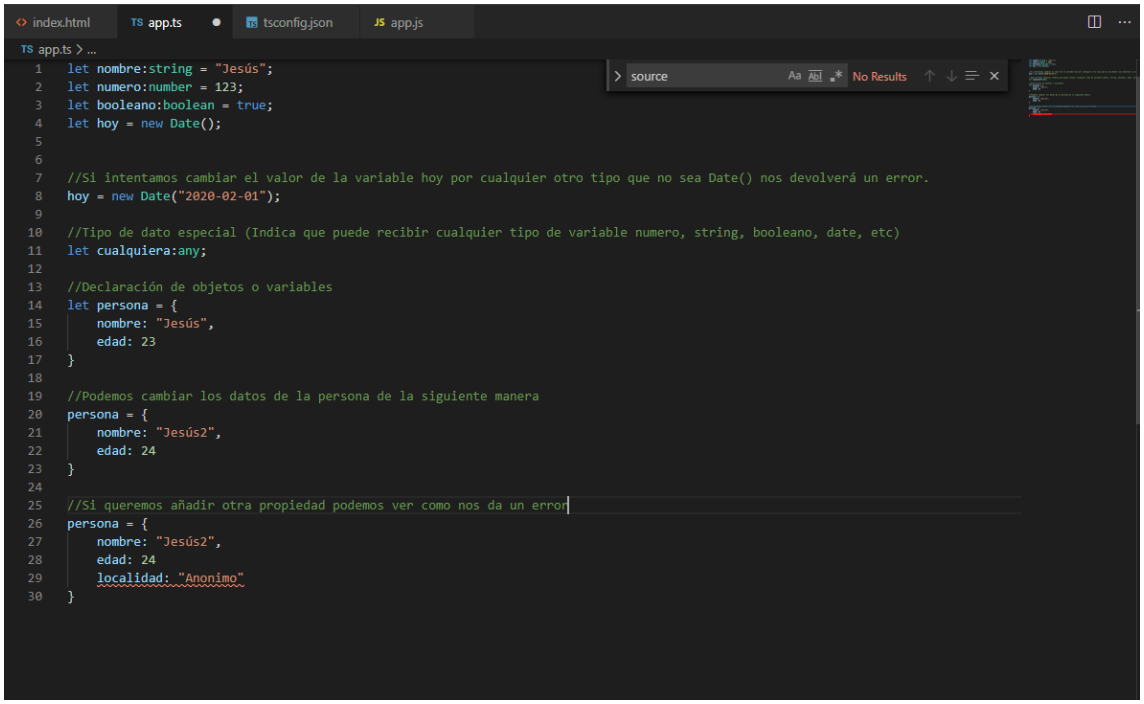
Aún así, TypeScript nos dirá que hay un error, pero nos dejará continuar, la manera correcta de declarar una variable de tipo string u otro tipo, es indicándole que tipo es como podemos ver a continuación:







```ts
let nombre:string = "Jesús";
let numero:number = 123;
let booleano:boolean = true;
let hoy = new Date();


//Si intentamos cambiar el valor de la variable hoy por cualquier otro tipo que no sea Date() nos devolverá un error.
hoy = new Date("2020-02-01");
```



```ts
let nombre:string = "Jesús";
let numero:number = 123;
let booleano:boolean = true;
let hoy = new Date();


//Si intentamos cambiar el valor de la variable hoy por cualquier otro tipo que no sea Date() nos devolverá un error.
hoy = new Date("2020-02-01");

//Tipo de dato especial (Indica que puede recibir cualquier tipo de variable numero, string, booleano, date, etc)
let cualquiera:any;

//Declaración de objetos o variables
let persona = {
    nombre: "Jesús",
    edad: 23
}

//Podemos cambiar los datos de la persona de la siguiente manera
persona = {
    nombre: "Jesús2",
    edad: 24
}

//Si queremos añadir otra propiedad podemos ver como nos da un error
persona = {
    nombre: "Jesús2",
    edad: 24
    localidad: "Anonimo"
}
```

PLANTILLAS/TEMPLATE LITERALES TYPESCRIPT

El siguiente código es totalmente válido, pero si que nos podemos dar cuenta de que nos alarga mucho el código y aumenta en cantidad de líneas:

```ts
/* PLANTILLAS O TEMPLATES */
let nombre2:string = "Jesús";
let apellido:string = "Rodríguez";
let edad1:number = 23;

let texto = "Hola " + nombre2 + " " + apellido + ", edad: " + edad1;

console.log(nombre2);
```

Por lo tanto, una manera correcta es utilizar un template literal que podemos ver a continuación:

```
32  /* PLANTILLAS O TEMPLATES */
33  let nombre2:string = "Jesús";
34  let apellido:string = "Rodríguez";
35  let edad1:number = 23;
36
37  //let texto = "Hola " + nombre2 + " " + apellido + ", edad: " + edad1;
38  let texto = `Hola, ${nombre} ${apellido}, edad: ${edad1}`;
39
40  console.log(texto);
```

Si queremos realizar una multilínea, podemos realizarlo de la siguiente manera:

```
32  /* PLANTILLAS O TEMPLATES */
33  let nombre2:string = "Jesús";
34  let apellido:string = "Rodríguez";
35  let edad1:number = 23;
36
37  //let texto = "Hola " + nombre2 + " " + apellido + ", edad: " + edad1;
38  let texto = `Hola,
39      ${nombre} ${apellido}, edad: ${edad1}`;
40
41  console.log(texto);
```

```
Hola,                                                    app.js:31
    Jesús Rodríguez, edad: 23
>
```

Podemos ver que el espacio que hemos dejado se contempla. Simplemente lo que realiza es una multilínea equivalente a “\n”.

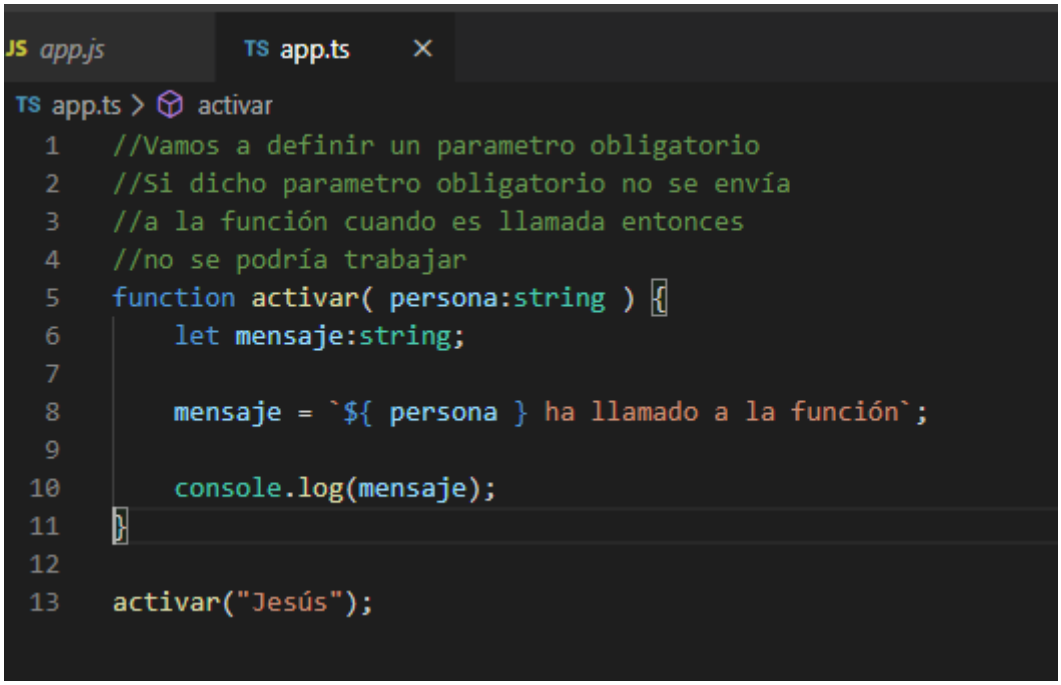
También se puede llamada a funciones que devuelven un resultado para asignarlas a una variable, como podemos ver a continuación:

Para ello utilizamos el template cuando llamamos a la función getNombre().

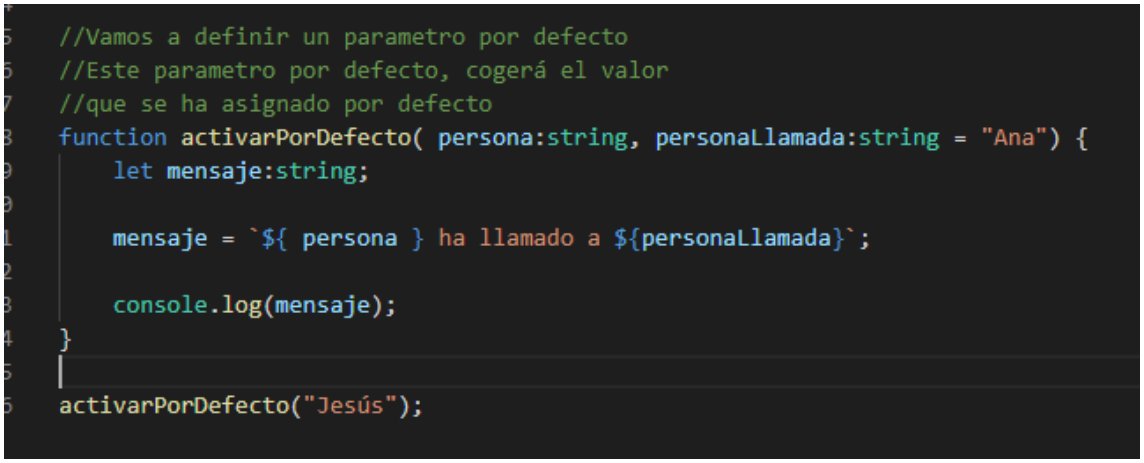
```
41  function getNombre() {
42      return "Jesús";
43  }
44
45  let nombre3:string = `${getNombre()}`
46
```

FUNCIONES: PARÁMETROS OPCIONALES, OBLIGATORIOS Y POR DEFECTO

Un parámetro obligatorio en una función es aquel, que cuando se llama a dicha función debe ser enviado en la llamada, en caso de no enviarlo, entonces no se podría trabajar con dicha llamada. Además, el parámetro o parámetros obligatorios que deban enviarse, deben corresponder con el tipo de variable que recibe dicha función.



Parámetro por defecto, este parámetro cogerá el valor por defecto asignado en caso de no enviar ningún dato en la llamada de la función.



Parámetro opcional, este parámetro se puede enviar o no, pero debemos tener en cuenta, que si queremos enviarlo, en la siguiente imagen tenemos un parámetro por defecto, si queremos informar el opcional, también debemos informar el por defecto, ya que si solo enviamos 2 parámetros nada más y se nos olvida el por defecto, entonces el valor por defecto tomará el valor de la variable día que queríamos enviar.

```
//Vamos a definir un parametro opcional
//Este parametro opcional, se podrá enviar
//o no a la función
function activarOpcional( persona:string, personaLlamada:string = "Ana", dia?:string) {
    let mensaje:string;

    if (dia) {
        mensaje = `${ persona } ha llamado a ${personaLlamada} el día ${dia}`;
    } else {
        mensaje = `${ persona } ha llamado a ${personaLlamada}`;
    }


    console.log(mensaje);
}

// Muy importante si queremos enviar el dia, debemos informar también el parametro personaLlamada
activarOpcional("Jesús", "Ana", "10 de Marzo");
```

Por ley, no podemos poner el valor opcional al principio y el valor obligatorio al final. Por lo tanto, como buenas prácticas:

- Debemos siempre indicar los valores obligatorios al principio, y los opcionales al final.

FUNCIONES DE FLECHA

Ejemplo función simple de flecha:

```
/* FUNCIONES DE FLECHA */
let funcion = function(a) {
    return a;
}

//Recibimos como parametro (a) y devolvemos => a
let funcionFlecha = (a) => a;

console.log(funcion("funcion"));
console.log(funcionFlecha("funcionFlecha"));
```

Ejemplo función de flecha con dos parámetros:

```
//Enviamos más de un argumento a la función de flecha

let funcion2 = function(a:number, b:number) {
    return a + b;
}

let funcionFlecha2 = (a:number, b:number) => a + b;
```

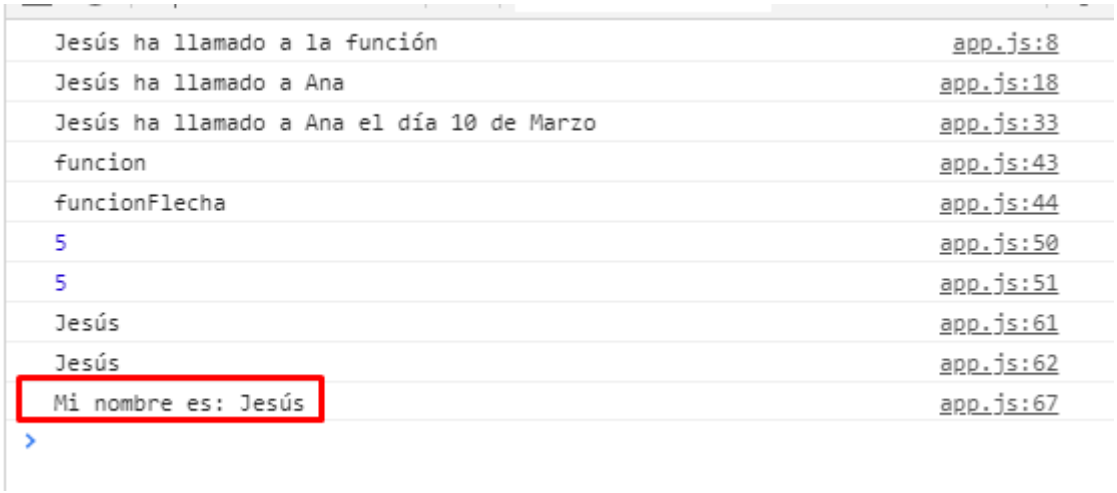
Ejemplo función flecha con más de una línea de código:

```
let funcion3 = function(nombre:string) {
    nombre = nombre.toUpperCase();
    return nombre;
}

let funcionFlecha3 = (nombre:string) => {
    nombre = nombre.toUpperCase();
    return nombre;
}
```

Ejemplo función flecha con objetos:

Este código es una función normal:

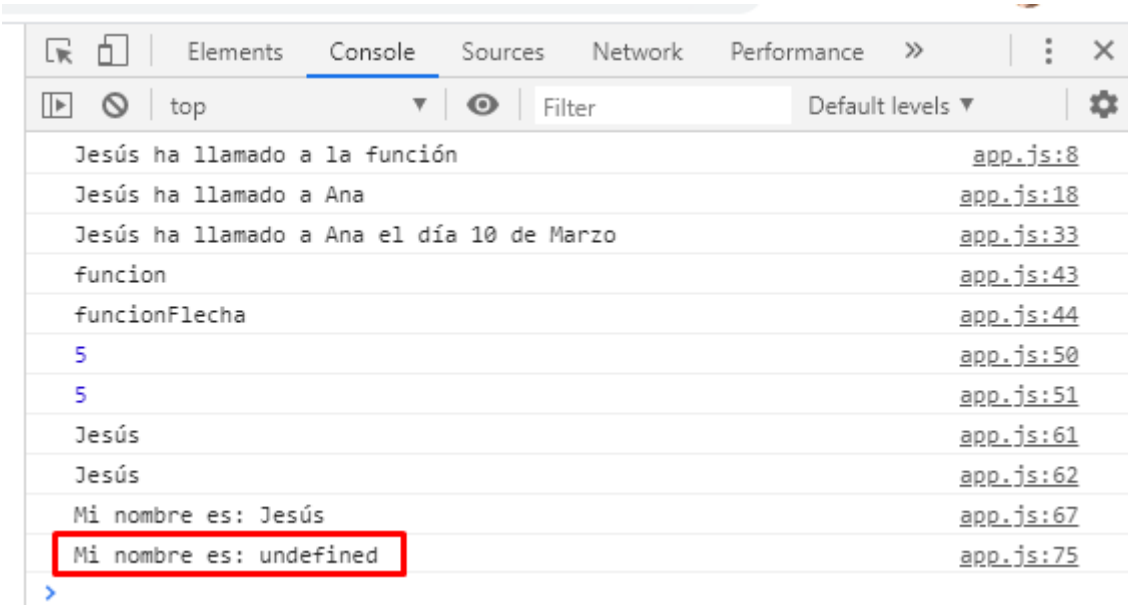
```
//Función de flecha con objetos
let persona = {
    nombre: "Jesús",
    miNombre() {
        console.log("Mi nombre es: " + this.nombre);
    }
}
```



Como podemos ver, no tiene ninguna complicación ni parece que pueda causar problemas, pero si a la función le realizamos un setTimeout, pasa lo siguiente:

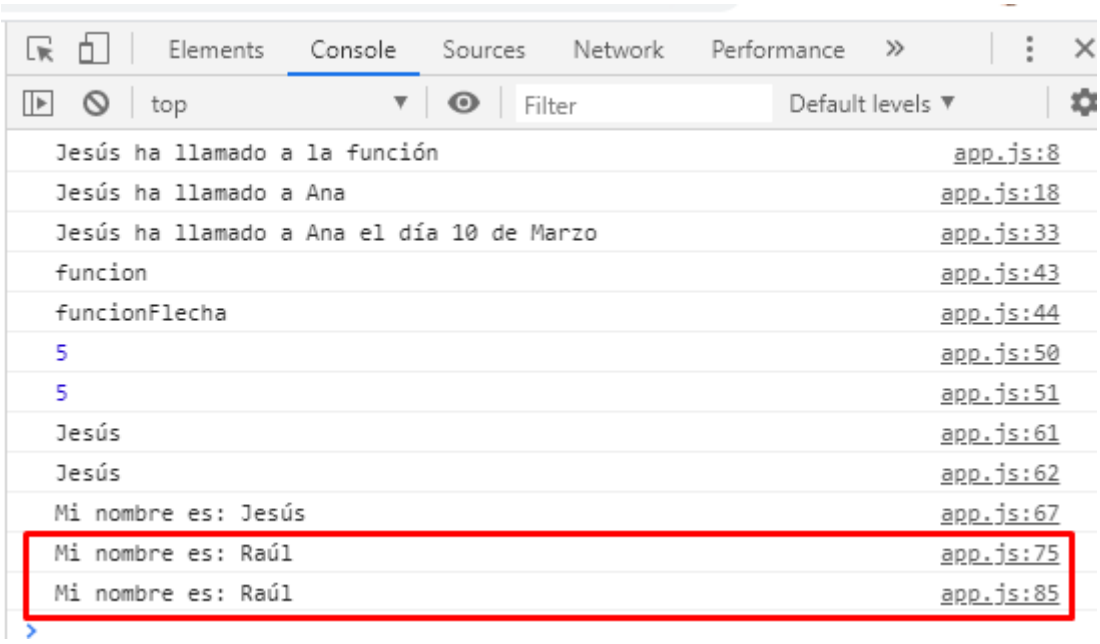
- Esto es debido a que cuando se setea un setTimeout dentro de un objeto, el this dentro del setTimeout apunta al objeto y en este caso es Windows debido a que transcurrido ese tiempo ya ha salido de dicha función.

```
let persona2 = {
    nombre: "Jesús",
    miNombre() {
        setTimeout(function() {
            console.log("Mi nombre es: " + this.nombre);
        }, 1500);
    }
}

persona2.miNombre();
```



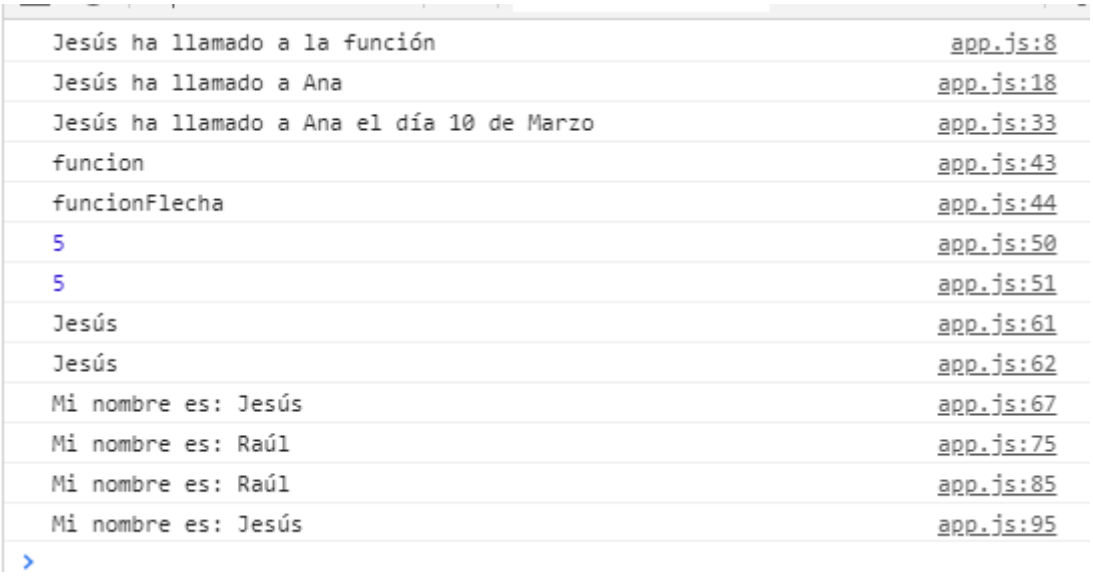
- Podemos ver el siguiente ejemplo, vamos a declarar un nombre como variable global y veremos qué pasa.

```
92  let persona2 = {
93      nombre: "Jesús",
94      miNombre() {
95          setTimeout(function() {
96              console.log("Mi nombre es: " + this.nombre);
97          }, 1500);
98      }
99  }
100
101 persona2.miNombre();
102
103 let nombre = "Raúl";
104 let persona3 = {
105     nombre: "Jesús",
106     miNombre() {
107         setTimeout(f  (method) Console.log(message?: any, ...optionalParams: any[]): void
108             console.log("Mi nombre es: " + this.nombre);
109         }, 1500);
110     }
111 }
112
113 persona3.miNombre();
```



Todo esto lo solucionamos utilizando la función de flecha, como podemos ver a continuación:

```
//Aquí utilizamos función de flecha para objetos
let personaFlecha = {
    nombre: "Jesús",
    miNombre() {
        setTimeout(() => console.log("Mi nombre es: " + this.nombre), 1500); //Se quitan las llaves porque solo tiene una línea
        //Es decir, solo retorna el log.
    }
}

personaFlecha.miNombre();
```



Desestructuración de Objetos y Arreglos

Cuando instanciamos un objeto, normalmente para acceder a sus propiedades debemos realizar la siguiente sentencia **objeto.propiedad**, en este punto es dónde interviene la desestructuración de objetos, ya que nos facilita poder obtener la propiedad de un objeto sin necesidad de realizar **objeto.propiedad**, simplemente con poner el nombre de la **propiedad** estamos accediendo a la **propiedad** de dicho objeto. A continuación veremos un ejemplo:

```
(() => {

    //Creamos un objeto
    const persona = {
        nombre: 'Jesús',
        apellido1: 'Rodríguez',
        apellido2: 'Jurado'
    }

    //Para obtener las propiedades del objeto necesitamos realizar las siguientes sentencias
    console.log(persona.nombre);
    console.log(persona.apellido1);
    console.log(persona.apellido2);

    //Para evitar tener que acceder cada a una propiedad del objeto cada vez que queramos obtener un valor de el
    //Realizamos Desestructuración de objetos que veremos a continuación
    //Existe let, const y var
    // let --> Permite cambiar el valor del objeto
    // const --> No permite cambiar el valor del objeto ya que es definido como constante.

    //La siguiente sentencia se puede traducir como, tomamos el objeto persona y me extraes una o
    //las propiedades que indiquemos dentro de { } (Podemos indicar de 1 a todas las propiedades del objeto),
    //de esta manera estamos creando constantes que provienen del objeto persona
    const { nombre, apellido1, apellido2 } = persona;

    console.log(nombre);
    console.log(apellido1);
    console.log(apellido2);
})();
```

En este tipo de **desestructuración**, no importa el orden en el que indiquemos las propiedades que queremos obtener siempre y cuando sea un **objeto**, es decir, podemos realizar la siguiente sentencia:

const { apellido2, nombre} = persona; Como vemos estamos obteniendo dos propiedades del objeto persona y en ningún momento está siguiendo el orden en que están definidas.

Este tipo de desestructuración también funciona en los **argumentos de una función**, como podemos ver a continuación:

```
    //Desestructuración en los argumentos de una función
    const extraer = ( persona: any ) => {
        const { apellido2, nombre } = persona;

        console.log( nombre );
        console.log ( apellido2 );
    }

    extraer(persona);
```

También podemos extraer las propiedades del argumento que se envié a una función de la siguiente manera:

```
//Desestructuración de objetos en los parametros de una función en dónde se definen los argumentos,
//se puede extraer las propiedades de lo que se envie
// Lo correcto no sería definir any, si no el tipo de dato, es decir, en este caso sería persona
//(Este paso corresponde a interfazes y clases).
const extraer2 = ( { nombre, apellido2 }: any ) => {

    console.log( nombre );
    console.log ( apellido2 );
}

extraer2(persona);
```

La desestructuración de arreglos/arrays sigue la misma filosofía que se ha explicado anteriormente, pero tiene algunas peculiaridades que a continuación se van a definir.

- Cuando declaramos la variable que realizará la función de **desestructuración de arreglos/arrays** no se define con **{ }** como hemos realizado anteriormente con las **propiedades del objeto** si no que se utiliza **[ ]**.
- En este caso, el orden que indiquemos sí que debe ser el correcto, es decir, debe llevar el mismo orden que el array, pero la definición de la propiedad no debe ser la misma.

Procedemos a ver un ejemplo:

```
52    //Desestructuración de Arreglos
53    // Indicamos que personas va a ser un arreglo en el que solo va a aceptar como tipo variable strings.
54    const personas: string[] = ['Jesús', 'Juanma', 'Pepe'];
55
56    //Extraer elementos de un arreglo
57    console.log( personas[0] );
58    console.log( personas[1] );
59    console.log( personas[2] );
60
61    //Para evitar tener que acceder cada posición del array/arreglo
62    //Realizamos Desestructuración de arrays o arreglos como hemos visto anteriormente.
63    //Debemos tener en cuenta que como queremos desustructurar un arreglo/array en vez de un objeto
64    // la variable que declaramos en vez de empezar por {} empezará por []
65    const [ persona1, persona2, persona3 ] = personas;
66
67    //Llamada a las propiedades de la constante de la desestructuración de arreglos.
68    console.log( persona1 );
69    console.log( persona2 );
70    console.log( persona3 );
```

Como podemos ver persona1 apunta a Jesús, persona2 a Juanma y persona3 a Pepe, esto corresponde con el punto 3, es decir, lleva el orden del array/arreglo definido pero la nomenclatura de las propiedades desestructuradas no coincide con la propiedad definida en el arreglo/array, pero si podemos ver que el orden que sigue es exactamente igual que el del array, es decir, si declaramos la constante de desestructuración de la siguiente manera:

Const [persona2, persona1, persona3] = personas;

Persona2 apuntará a Jesús, persona1 a Juanma y persona3 a Pepe.

En el caso de solo querer desestructurar una propiedad del array o arreglo, se realiza de la siguiente manera:

```
// En el caso de no querer desestructura a Jesús y Juanma y solo a Pepe se realiza de la siguiente manera
const [ , , personaPepe ] = personas;

console.log( personaPepe );
```

También se puede realizar desestructuración de arreglos/array en los argumentos de una función como se ha realizado anteriormente con los objetos, a continuación, dejo un ejemplo:

```
//Desestructuración de arreglos/arrays en los argumentos de una función
const extraerArr = ( personas: string[] ) => {
    console.log( personas[0] );
    console.log( personas[1] );
    console.log( personas[2] );
}

extraerArr(personas);

// En este caso cualquier array que le pasemos de tipo String
//la posición 0 corresponde con persona1
//la posición 1 corresponde con persona2
//la posición 2 corresponde con persona3
const extraerArr2 = ( [ persona1, persona2, persona3 ]: string[] ) => {
    console.log( persona1 );
    console.log( persona2 );
    console.log( persona3 );
}

extraerArr2(personas);
```

Promesas

Las promesas nos sirven para ejecutar un código sin bloquear el código de nuestra aplicación, algunas de las características de una promesa son las siguientes:

- Dentro de la promesa se necesita una función que recibe 2 argumentos
    - Resolve: Es lo que nosotros vamos a devolver o llamar cuando todo se completa correctamente.
    - Reject: Es lo que nosotros vamos a devolver o llamar si ocurre algún error.
- Se suelen utilizar cuando se requiere realizar una llamada a un servicio web.
    - Si todo va correcto se llama o devuelve resolve.
    - Si ocurre algún error se llama o devuelve reject.

A continuación, os dejo un ejemplo sencillo:

Cuando se realiza la llamada a una promesa podemos observar dos cosas:

Symbol es como un identificador.

Then → es lo que queremos ejecutar cuando todo ha ido correcto.

Catch → es lo que queremos ejecutar cuando ha surgido algún error.

```
(() => {

    console.log('Inicio');

    // Para ejecutar una promesa en typeScript debemos tener en el archivo tsconfig el targer como es6.
    // Dentro de la promesa se necesita mandar una función que recibe 2 argumentos.
    // Resolve es lo que nosotros vamos a devolver cuando todo se completa correctamente.
    // Reject es lo que nosotros vamos a devolver si ocurre algún error.
    // Se suelen utilizar normalmente cuando se requiere realizar una llamada a un servicio web
    // Si todo va correcto se llama a resolve y si va mal a reject
    const prom1 = new Promise(( resolve, reject) => {
        setTimeout(() => {
            resolve('Se terminó el timeout');
        }, 1000);
    });

    prom1.then( mensaje => console.log( mensaje ));
    console.log('Fin')
})();
```

Cuando ejecutamos el código el resultado es el siguiente:

Como podemos ver, lo primero que se muestra es Inicio y Fin, pero la promesa se ha definido antes del Fin esto ocurre debido a que como el código de la promesa no es bloqueante se sigue ejecutando el resto del código y 1 segundo después que es lo definido en la promesa, se muestra el resultado de la misma.

```
Inicio                                  app.js:2
Fin                                     app.js:15
Se terminó el timeout                   app.js:14
>
```

Qué pasaría si ejecutamos el código y en vez de devolver resolve devolvemos reject. A continuación, un ejemplo:

```
(() => {

    console.log('Inicio');

    // Para ejecutar una promesa en typeScript debemos tener en el archivo tsconfig el targer como es6.
    // Dentro de la promesa se necesita mandar una función que recibe 2 argumentos.
    // Resolve es lo que nosotros vamos a devolver cuando todo se completa correctamente.
    // Reject es lo que nosotros vamos a devolver si ocurre algún error.
    // Se suelen utilizar normalmente cuando se requiere realizar una llamada a un servicio web
    // Si todo va correcto se llama a resolve y si va mal a reject
    const prom1 = new Promise(( resolve, reject) => {
        setTimeout(() => {
            reject('Se terminó el timeout');
        }, 1000);
    });

    prom1.then( mensaje => console.log( mensaje ));
    console.log('Fin')
})();
```

```
Inicio                                                  app.js:2
Fin                                                     app.js:15
▶Uncaught (in promise) Se terminó el timeout            index.html:1
>
```

Como vemos nos devuelve un error, esto sucede debido a que no estamos controlando los errores, lo correcto sería tratarlos con el catch. Por lo tanto, lo correcto sería lo siguiente:

- Algo a tener en cuenta es en vez de mostrar un console.log mostrar un conselo.warn de esta manera es más fácil de identificar a la hora de hacer debug.

```
(() => {

    console.log('Inicio');

    // Para ejecutar una promesa en typeScript debemos tener en el archivo tsconfig el targer como es6.
    // Dentro de la promesa se necesita mandar una función que recibe 2 argumentos.
    // Resolve es lo que nosotros vamos a devolver cuando todo se completa correctamente.
    // Reject es lo que nosotros vamos a devolver si ocurre algún error.
    // Se suelen utilizar normalmente cuando se requiere realizar una llamada a un servicio web
    // Si todo va correcto se llama a resolve y si va mal a reject
    const prom1 = new Promise(( resolve, reject) => {
        setTimeout(() => {
            reject('Se terminó el timeout');
        }, 1000);
    });

    prom1
        .then( mensaje => console.log( mensaje ))
        .catch ( err => console.warn( err ));
    console.log('Fin')
})();
```

Cuando ejecutamos, el mensaje ya si aparece correctamente y no como un error por no haber capturado la excepción:

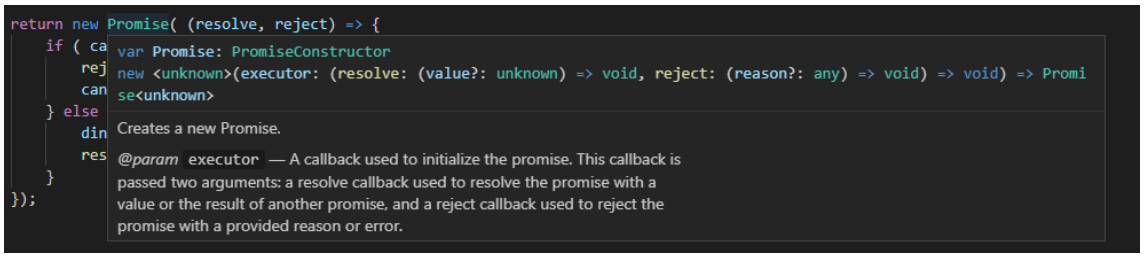
```
Inicio                                  app.js:2
Fin                                     app.js:16
⚠ ▶ Se terminó el timeout               app.js:15
>
```

A continuación, veremos un ejemplo más complejo:

Realizamos una función que devuelve una promesa, por lo tanto, cuando llamamos a dicha función, en la llamada debemos realizar el tratamiento de una promesa, ya que es lo que devuelve la función, lo vamos a ver a continuación en un ejemplo:

```
(() => {

    const retirarDinero = ( cantidadRetirar: number): Promise<number> => {
        let dineroActual = 1000;

        return new Promise( (resolve, reject) => {
            if ( cantidadRetirar > dineroActual) {
                reject('No se puede retirar ya que la cantidad a retirar ' +
                cantidadRetirar + " es superior al dineroActual " + dineroActual);
            } else {
                dineroActual -= cantidadRetirar;
                resolve(dineroActual);
            }
        });
    }

    retirarDinero( 1500 )
        .then( dineroActual => console.log(`Le queda en la cuenta una cantidad de ${dineroActual} euros`))
        .catch( err => console.warn( err ));
        //La línea de arriba también podemos resumirla de la siguiente manera
        // Solo cuando se esté ejecutando una sola línea
        //.catch(console.warn);
})();
```

Es muy importante definir el tipo de retorno para no tratar con objetos no conocidos o denominados unknown, para ello realizamos el tipado, en la siguiente imagen podemos ver que la promesa nos define que no sabe qué tipo es lo que devuelve.



Para definir que tipo va a devolver la función, que en este caso es una promesa, se especifica después de los argumentos que recibe la función, el tipo que se define es aquel que devuelve en caso de que todo vaya correcto, ya que si sucede algún error pues devuelve un error, a continuación, se muestra un ejemplo:

```
const retirarDinero = ( cantidadRetirar: number): Promise<number> => {
    let dineroActual = 1000;

    return new Promise( (resolve, reject) => {
        if ( cantidadRetirar > dineroActual) {
            reject('No se puede retirar ya que la cantidad a retirar ' +
            cantidadRetirar + " es superior al dineroActual " + dineroActual);
        } else {
            dineroActual -= cantidadRetirar;
            resolve(dineroActual);
        }
    });
}
```

Si ponemos otro tipado y es incorrecto al tipo que devuelve nos lo mostrará con un error:

```
const retirarDinero = ( cantidadRetirar: number): Promise<string> => {
    let dineroActual = 1000;

    return new Promise( (resolve, reject) => {
        if ( cantidadRetirar > dineroActual) {
            reject('No se puede retirar ya que la cantidad a retirar ' +
            cantidadRetirar + " es superior al dineroActual " + dineroActual);
        } else {
            dineroActual -= cantidadRetirar;
            resolve(dineroActual);
        }
    });
}
```

Interfaces

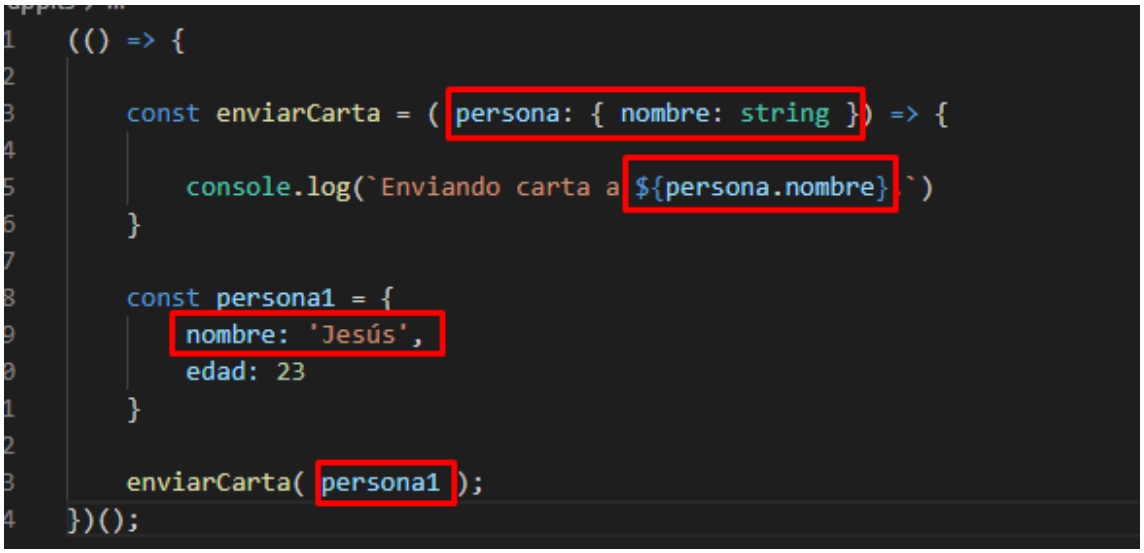
```
1   (() => {
2
3       const enviarCarta = ( persona: any) => {
4
5           console.log(`Enviando carta a ${persona.nombre}.`)
6       }
7
8       const persona1 = {
9           nombrePersona1: 'Jesús',
10          edad: 23
11      }
12
13      enviarCarta( persona1 );
14  })();
```

Este trozo de código devuelve la siguiente respuesta:

```
Enviando carta a undefined.                                    app.js:3
>
```

Esto está ocurriendo debido a que el objeto que estamos enviando la propiedad '**nombre'** que se obtiene en la función **enviarCarta** no corresponde con la propiedad del objeto debido a que es **nombrePersona1**.

La solución a esto es realizar un tipado, es decir, en vez de definir que el tipo que nos llega es any, definimos el tipo que vamos a recibir en este caso sería del objeto persona el nombre de tipo string, a continuación, veremos un ejemplo:

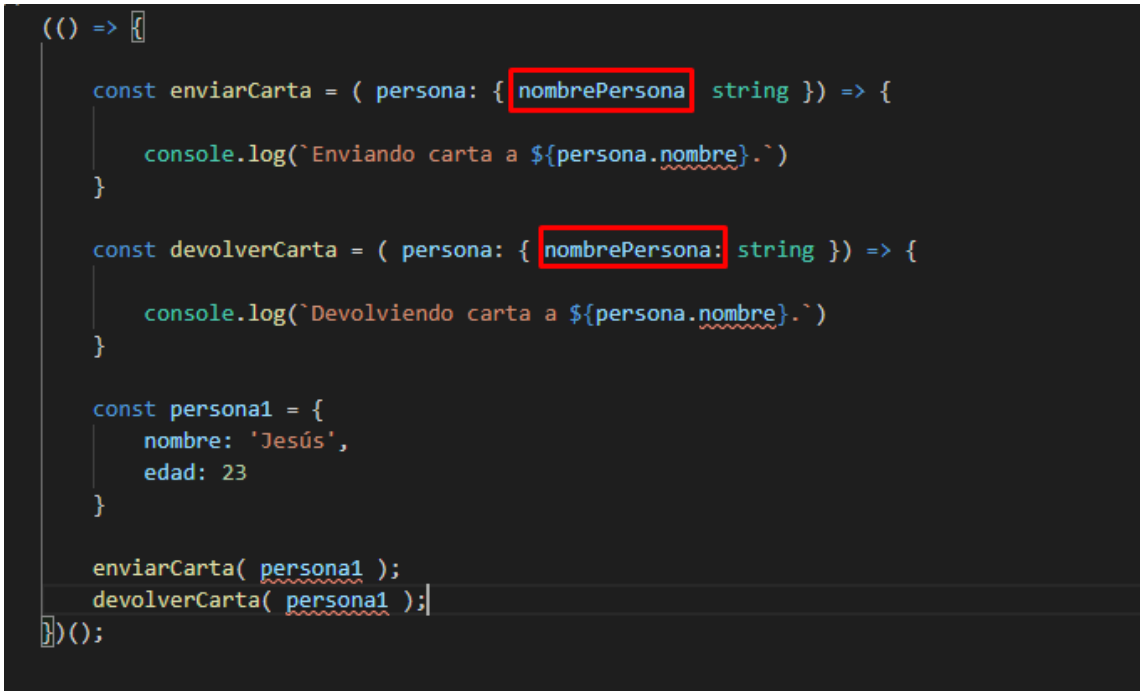


```
(() => {

    const enviarCarta = ( persona: { nombre: string }) => {

        console.log(`Enviando carta a ${persona.nombre}.`)
    }

    const persona1 = {
        nombre: 'Jesús',
        edad: 23
    }

    enviarCarta( persona1 );
})();
```

Lo definido en la función indica, que vamos a enviar un objeto, que al menos lleva una **propiedad** denominada '**nombre'** y que es de **tipo string**.

Pero esto anterior, también puede provocar ciertos problemas a futuro.

Es posible que necesitemos crear una función que se denomine devolverCarta, y en vez de la variable ser nombre debemos cambiarla por nombrePersona, entonces nos supondría cambiarlo en demasiados lados, como podemos ver a continuación:



```
(() => {

    const enviarCarta = ( persona: { nombrePersona  string }) => {

        console.log(`Enviando carta a ${persona.nombre}.`)
    }

    const devolverCarta = ( persona: { nombrePersona: string }) => {

        console.log(`Devolviendo carta a ${persona.nombre}.`)
    }

    const persona1 = {
        nombre: 'Jesús',
        edad: 23
    }

    enviarCarta( persona1 );
    devolverCarta( persona1 );
})();
```

Para resolver este problema, debemos utilizar interfaces:

Una interfaz es algo parecido a una clase, pero no exactamente lo mismo.

- A una interfaz podemos definir que es lo que va a hacer.
- Una interfaz no tiene constructores.
- Únicamente contiene las reglas que debe cumplir un objeto para poder utilizarlo como un tipo, es decir, es parecido a crearse un tipo de dato.
- Son únicamente para typeScript, es decir, cuando abramos nuestro fichero .js no van a aparecer dichas interfaces.

```
(() => {

    interface Persona {
        nombre: string;
        edad: number;
        dni?: string;
    }
    const enviarCarta = ( persona: Persona) => {

        console.log(`Enviando carta a ${persona.nombre}.`)
    }

    const devolverCarta = ( persona: Persona) => {

        console.log(`Devolviendo carta a ${persona.nombre}.`)
    }

    const persona1: Persona = {
        nombre: 'Jesús',
        edad: 23
    }

    enviarCarta( persona1 );
    devolverCarta( persona1 );
})();
```

El **signo** de **?** en la propiedad dni indica que puede ser opcional, como vemos el objeto persona1 tiene un tipado a la interfaz Persona, y cada parámetro que se recibe en las funciones cuyo tipado es persona, está regla es muy útil, debido a que si el día de mañana en la interfaz Persona se añade alguna propiedad más que sea obligatorio en todos los objetos cuyo tipado sea Persona nos indicará que falta dicha propiedad. Además, si alguna propiedad como puede ser nombre cambia a ser nombrePersona, nos indicará que debemos cambiarlo en todo el resto de código de nombre a nombrePersona dónde se utilizando. Esto nos sirve para tener más cerrado el código y evitar utilizar propiedades genéricas.

Introducción a las clases de POO

Una clase es una abstracción de un objeto de la vida real, es decir, algo que existe en la vida real.

Por ejemplo, un coche tienes sus **propiedades**, como puede ser:

- Número de puertas
- Color
- Marca
- Modelo
- Fabricante

Con un coche se en la **vida cotidiana** se puede realizar una serie de **funciones** como puede ser **arrancar**, **apagar**, **aparcar**, **conducir**, etc.

Las funciones anteriores que se puede realizar con un coche en la vida cotidiana en **POO** son conocidos como **métodos**.

También podemos encontrar la **herencia**, la **herencia** consiste en tener varias clases y una de ella es la **herencia**, el resto de clases son heredadas del padre (La clase herencia). Es decir, las clases hijas (Las que heredan de la clase padre, que es la clase herencia) contienen todos los métodos y propiedades de la clase que heredan.

Podemos verlo con un ejemplo:

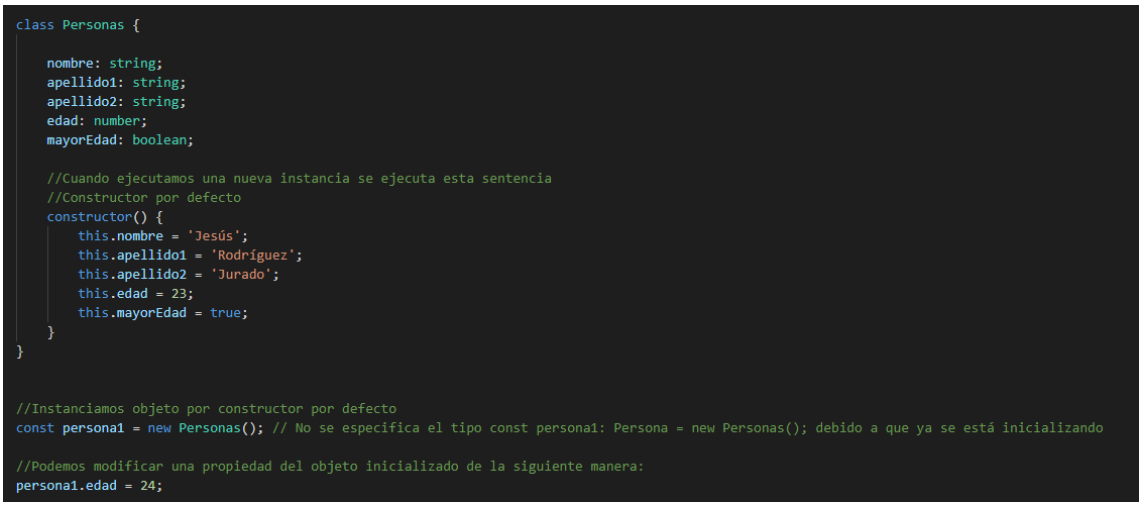
Como podemos ver la clase **Figura**, tiene dos propiedades (**lados** y **área**), y la clase **Cuadrado** y **Triángulo** están formadas por **dos propiedades** también que son las **mismas** que podemos encontrar en la clase **Figura**. Entonces aquí es donde debería intervenir la **herencia**, debido a que la clase **Cuadrado** y **Triángulo** podrían **heredar** de la clase **Figura** y contener las **propiedades** de la clase **Figura**.

Definición de una clase básica en TypeScript

Tenemos que tener en cuenta que para poder crear una clase en el fichero **tsconfig** en la **propiedad target** debemos tener **es6**.

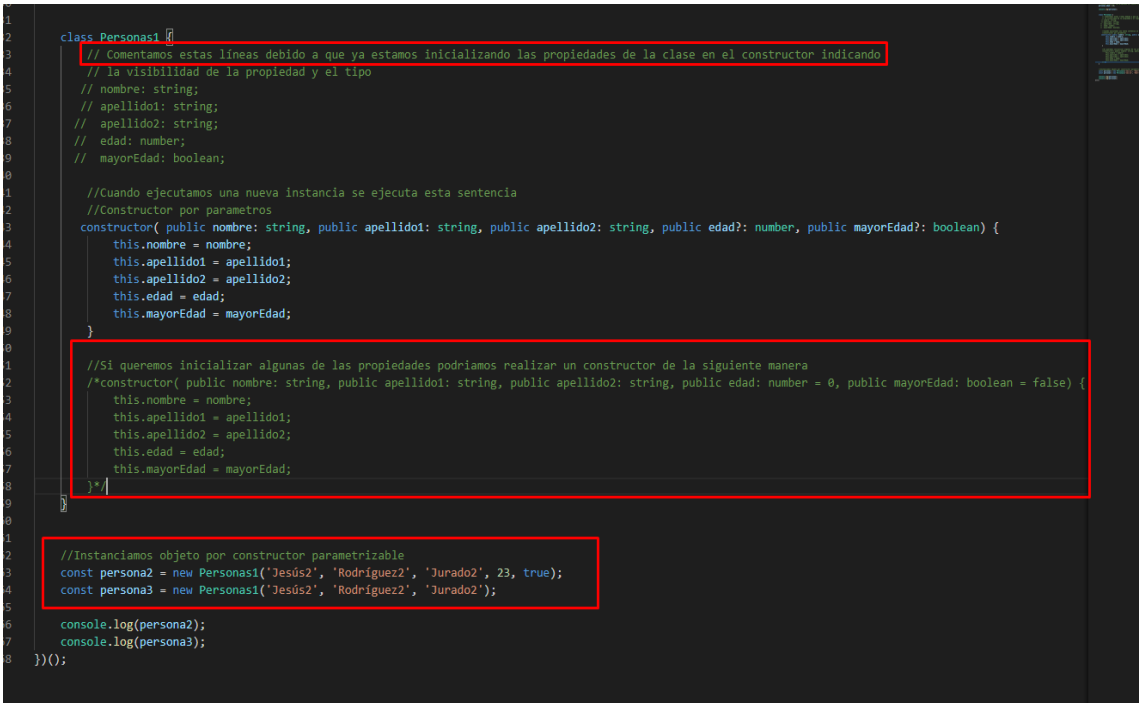
Lo primero que tenemos que saber, que para la **creación** de una **clase** debemos **inicializarla** con la palabra reservada **class**, además se debe seguir la conocida nomenclatura camello, Ejemplo **PersonasDiscapacitadas**. No obstante, a continuación, podemos ver un ejemplo de la creación de una clase en **TypeScript**.

En esta imagen, podemos ver la creación de una clase que está formada por un constructor por defecto.



```
class Personas {

    nombre: string;
    apellido1: string;
    apellido2: string;
    edad: number;
    mayorEdad: boolean;

    //Cuando ejecutamos una nueva instancia se ejecuta esta sentencia
    //Constructor por defecto
    constructor() {
        this.nombre = 'Jesús';
        this.apellido1 = 'Rodríguez';
        this.apellido2 = 'Jurado';
        this.edad = 23;
        this.mayorEdad = true;
    }
}


//Instanciamos objeto por constructor por defecto
const persona1 = new Personas(); // No se especifica el tipo const persona1: Persona = new Personas(); debido a que ya se está inicializando

//Podemos modificar una propiedad del objeto inicializado de la siguiente manera:
persona1.edad = 24;
```

En esta imagen, podemos ver la creación de una clase que está formada por un constructor parametrizable. Además, también podemos ver un ejemplo de constructor que es parametrizable y a la vez tiene datos inicializados por defecto (Método comentado).



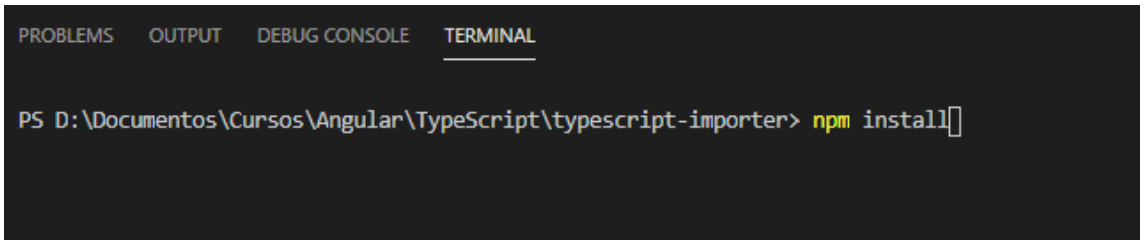
```
(function(){

    class Personas1 {
        // Comentamos estas líneas debido a que ya estamos inicializando las propiedades de la clase en el constructor indicando
        // la visibilidad de la propiedad y el tipo
        // nombre: string;
        // apellido1: string;
        //  apellido2: string;
        //  edad: number;
        //  mayorEdad: boolean;

        //Cuando ejecutamos una nueva instancia se ejecuta esta sentencia
        //Constructor por parametros
        constructor( public nombre: string, public apellido1: string, public apellido2: string, public edad?: number, public mayorEdad?: boolean) {
            this.nombre = nombre;
            this.apellido1 = apellido1;
            this.apellido2 = apellido2;
            this.edad = edad;
            this.mayorEdad = mayorEdad;
        }

        //Si queremos inicializar algunas de las propiedades podriamos realizar un constructor de la siguiente manera
        /*constructor( public nombre: string, public apellido1: string, public apellido2: string, public edad: number = 0, public mayorEdad: boolean = false) {
            this.nombre = nombre;
            this.apellido1 = apellido1;
            this.apellido2 = apellido2;
            this.edad = edad;
            this.mayorEdad = mayorEdad;
        }*/
    }


    //Instanciamos objeto por constructor parametrizable
    const persona2 = new Personas1('Jesús2', 'Rodríguez2', 'Jurado2', 23, true);
    const persona3 = new Personas1('Jesús2', 'Rodríguez2', 'Jurado2');

    console.log(persona2);
    console.log(persona3);
})();
```

Importaciones * URL

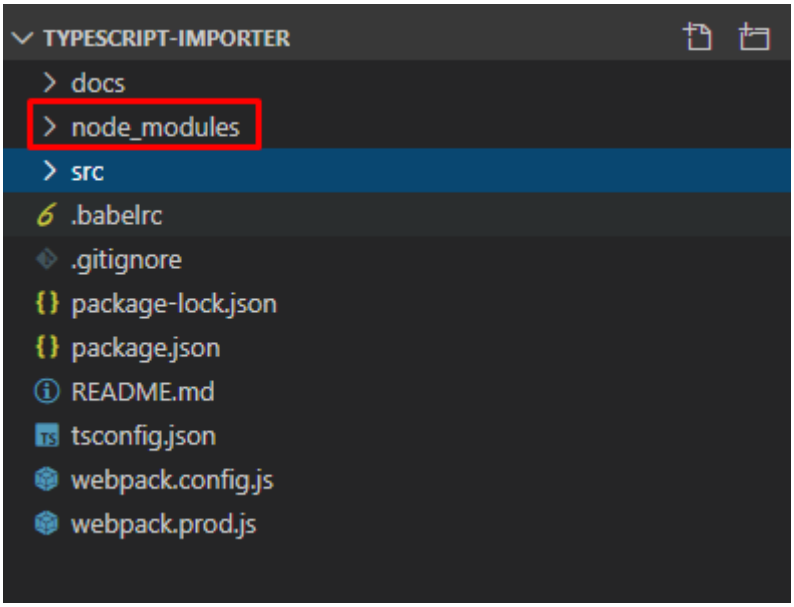
En el siguiente enlace podemos obtener un pack de inicio con typeScript, para la importación de módulos en TypeScript hay que realizar una serie de configuraciones (Todas estas configuraciones ya se encargan angular de realizarlas y no debemos nosotros hacerlas), por lo que, si queremos realizar importaciones en TypeScript, solamente debemos descargarnos el proyecto que hay en la siguiente URL, dónde ya está todo configurado para poder realizarlas sin problema.

https://github.com/Klerith/webpack-starter-typescript

Una vez que tenemos descargado el proyecto e importado en nuestro IDE, en mi caso **Visual Studio Code**, debemos abrir la terminar y ejecutar el siguiente comando:



Esto lo que va realizar es crear el directorio de los módulos de node y reconstruirlo para dejar todo el proyecto configurado con las dependencias necesarias para trabajar en él. Sí se ha instalado correctamente nos aparecerá el paquete de node_modules cuando hagamos refresh en el proyecto, en caso de no salir nos vamos al directorio ya que es posible que la carpeta esté en estado oculto.



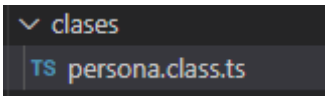
Para inicializar un proyecto que tiene dependencias npm y utiliza node debemos ejecutar el siguiente comando:

Como podemos ver en la consola, el proyecto se ha levantado en el puerto: 8080

Por lo tanto, para poder visualizar el proyecto nos iremos al navegador y pondremos: localhost:8080

Además, una de las ventajas es que cuando realizamos un cambio en el proyecto y guardamos, se realiza un **reload** automático de la página y podemos ver los cambios al instante y en caliente.

A continuación, nos vamos a basar en las importaciones, para ello en la carpeta src, creamos lo siguiente:



La extensión .class es opcional, pero es más indicativo ya que así sabemos automáticamente que este fichero se refiere a una clase.

Una vez que hemos creado la clase definimos las propiedades que queramos como hemos visto anteriormente, si que vamos a tener una peculiaridad, y es que debemos añadir la palabra reservada **export** antes de **class**, esto va a indicar que está clase va a ser poder utilizada en otros ficheros.

```
export class Persona {
    constructor (
        public nombre: string,
        public apellido: string
    ) {}
}
```

Posteriormente, nos vamos a nuestro fichero **index.ts** y realizamos una importación de la clase Persona, no deberíamos tener ningún problema ya que al haber hecho dicha clase export se va a poder utilizar desde otros ficheros, para ello debemos añadir la siguiente línea al principio del fichero dónde queremos importar la clase.

```
1  import { Persona } from './clases/persona.class';
2
3  const persona = new Persona('Jesús', 'Rodríguez');
4
5  console.log(persona);
```

Decoradores de clases

Un decorador es algo que se le va a colocar antes de la declaración de la clase, a continuación, podemos ver un ejemplo:

¡Muy importante! En el fichero tsconfig la propiedad experimentalDecorator debemos descomentarla y ponerla con valor true. Si hemos realizado el cambio y no desaparece el error, debemos primero reiniciar el npm start y volverlo a inicial, y en caso de no ir reiniciar Visual Studio Code.

```
1  function imprimirConsola ( constructorClase: Function ) {
2      console.log( constructorClase );
3  }
4
5  @imprimirConsola
6  export class Persona {
7      constructor (
8          public nombre: string,
9          public apellido: string
10     ) {}
11
12     imprimir () {
13         console.log(`${ this.nombre } - ${ this.apellido }`)
14     }
15 }
```

Como podemos ver, cuando ejecutamos nuestro código:

```
import { Persona } from './clases/persona.class';

const persona = new Persona('Jesús', 'Rodríguez');

//persona.imprimir();
console.log(persona);
```

Nos devuelve como resultado lo siguiente:

```
class Persona {                                          persona.class.ts:10
    constructor(nombre, apellido) {
        this.nombre = nombre;
        this.apellido = apellido;
    }
    imprimir() {
        console.log(`${this.nombre} - ${this.apellido}`);
 …
▶ Persona                                                index.ts:6
[WDS] Live Reloading enabled.                            client:52
> |
```

Por lo tanto, un decorador sirve para añadir funcionalidad a nuestra clase, es decir, al añadir el decorador lo que estamos indicando es que está clase va a ser un servicio, todo esto **angular** lo hará por defecto y de forma automática.

El decorador **@imprimirConsolar**, lo que indica es que la clase Persona va a tener todas las funcionalidades que añadamos a **imprimirConsola**.

En los decoradores se pueden enviar **parámetros** u **objetos**.

Tipado del retorno de una función

```
(() => {

    // Estamos indicando el tipado de la función posteriormente a la declaración de los parametros
    // que recibe la función, es decir, tipo number
    const sumar = (a: number, b: number): number => a + b;

    const nombre = (): string => 'Hola Jesús';

    const obtenerSalario = (): Promise<string> => {

        return new Promise (( resolve, reject ) => {
          // resolve(1) //Si descomentamos esta línea nos dará un error ya que 1 no es asignable al tipo string si no number
          resolve('Jesús');
        });

    }

    obtenerSalario().then(a => console.log( a.toUpperCase() ));
})();
```