¿Por qué utilizar TypeScript?

Como ya sabemos, TypeScript es un super lenguaje creado por Microsoft, que incorpora todo javascript incluyendo las versiones nuevas como Ecmascript6.

Además TypeScript incorpora todas las ventajas de la programación orientada a objetos.

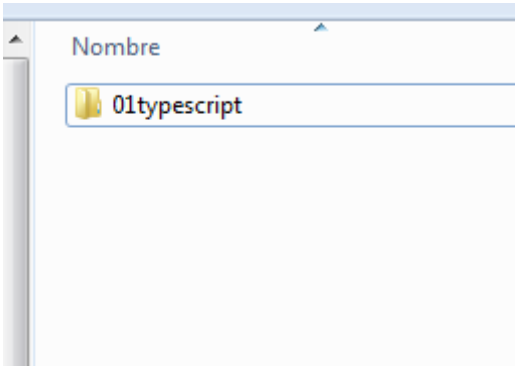
En Angular, todo el código lo vamos a hacer en TypeScript, pero no hay que asustarse, ya que la mayoría conoce javascript y eso nos da ya un conocimiento del 80% de Type Script, ya que todo el código válido en javascript es válido en TypeScript.

Para comenzar, vamos a ver las principales diferencias entre el uso de javascript y TypeScript, pero lo vamos a ir viendo con ejemplos.

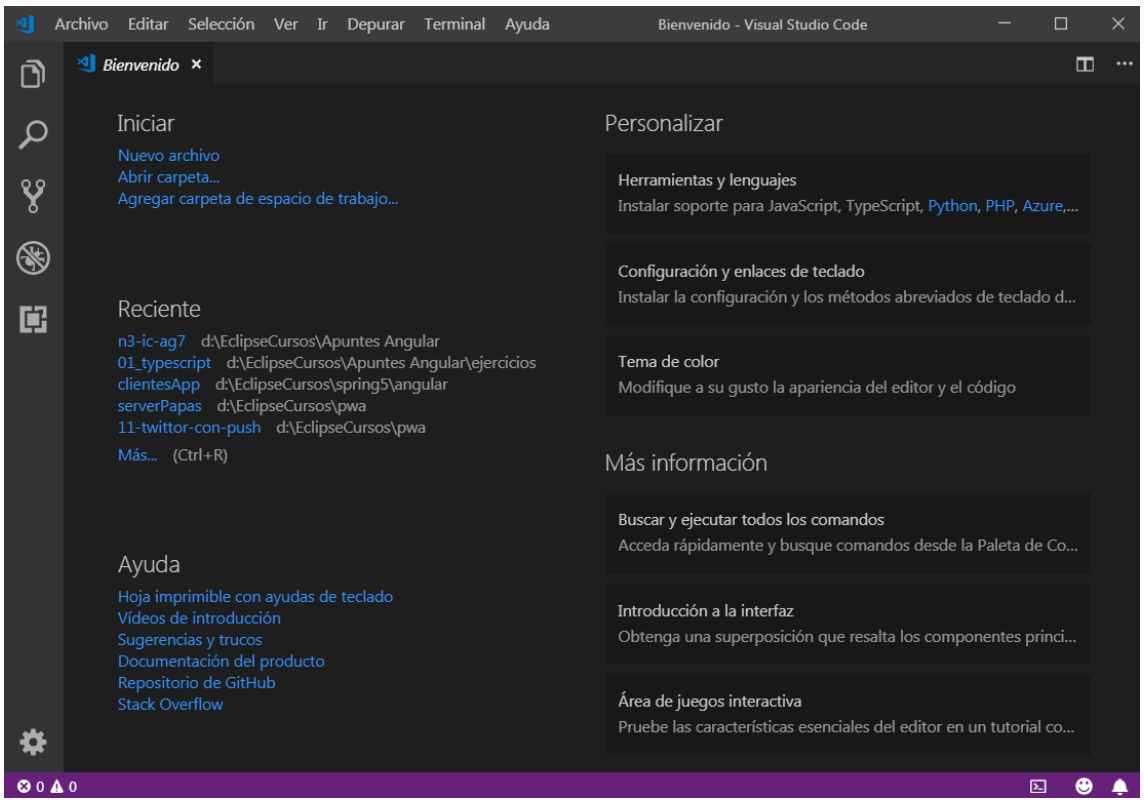
Ejemplo 1 donde veremos por qué es bueno el uso de type Script. Tipado de variables.

Para comenzar a hacer un programa, vamos a seguir estos pasos. Más adelante, los pasos previos ya los omitiré.

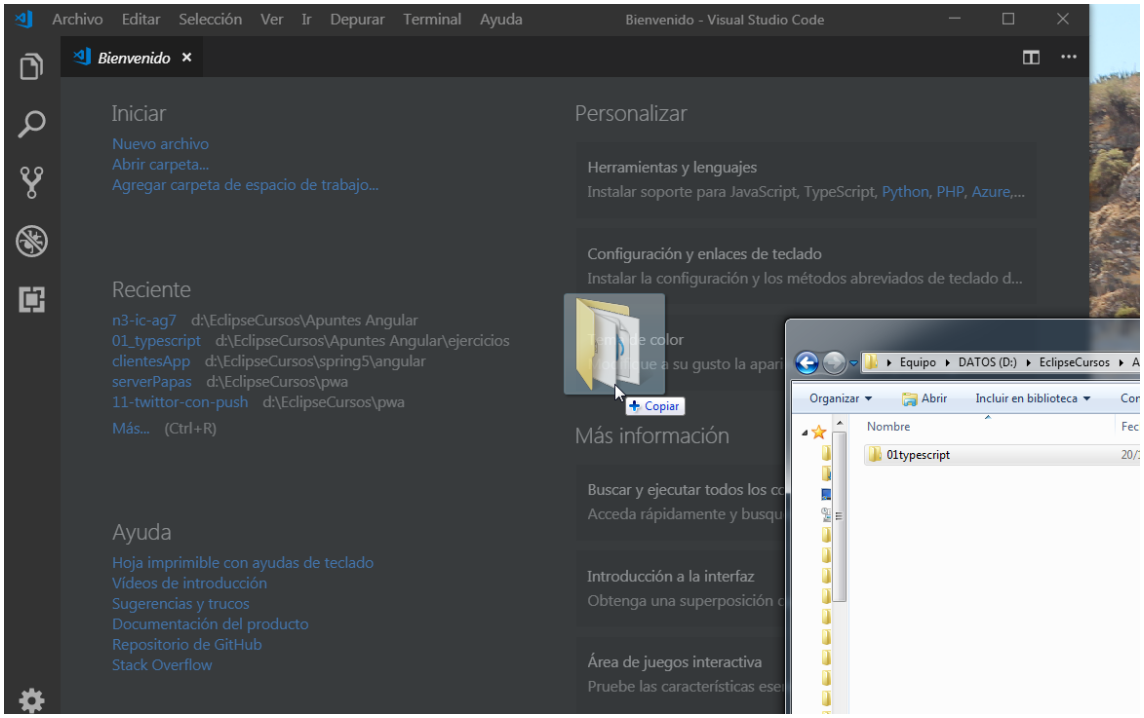
1. Creamos una carpeta en el disco duro llamada 01typescript. Conviene que no tenga espacios, que sea con minúsculas y que además no tenga caracteres raros



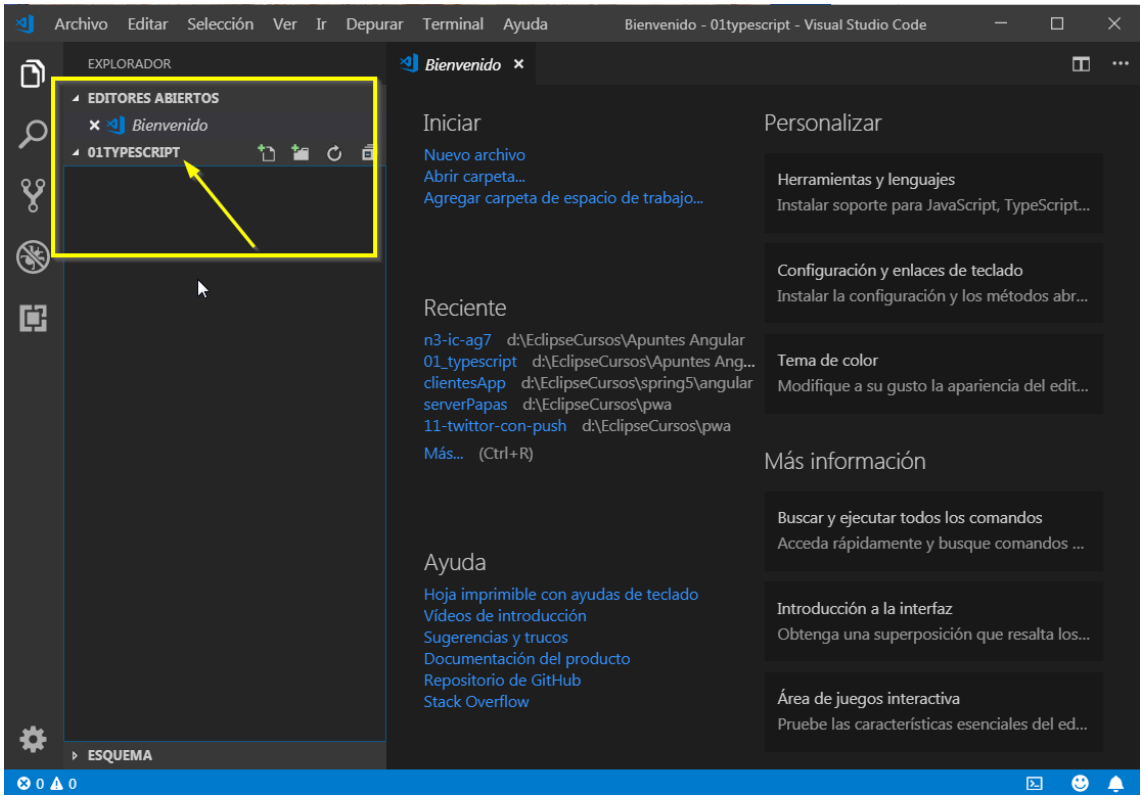
2. Abrimos nuestro editor de código, que como comentamos utilizaríamos Visual Studio Code (VS).



Una vez abierto, podemos arrastrar la carpeta creada al editor o pulsar Archivo -> Abrir Carpeta

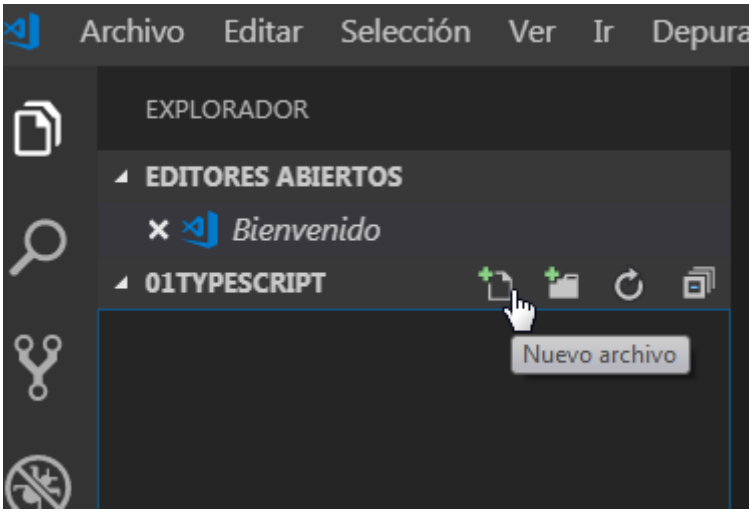


Soltamos la carpeta y deberíamos ver algo así



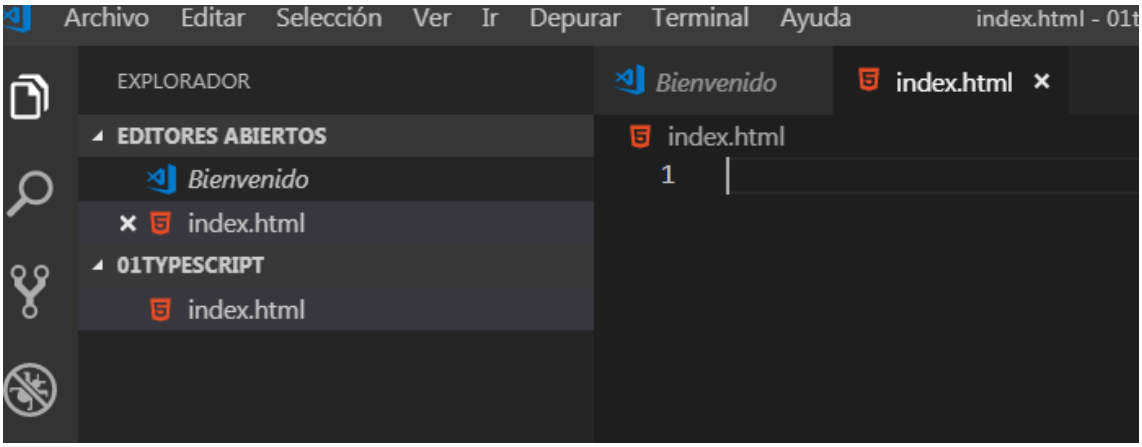
Vamos a pulsar en nuevo archivo y vamos a crear nuestro archivo index.html

Pulsando en nuevo archivo

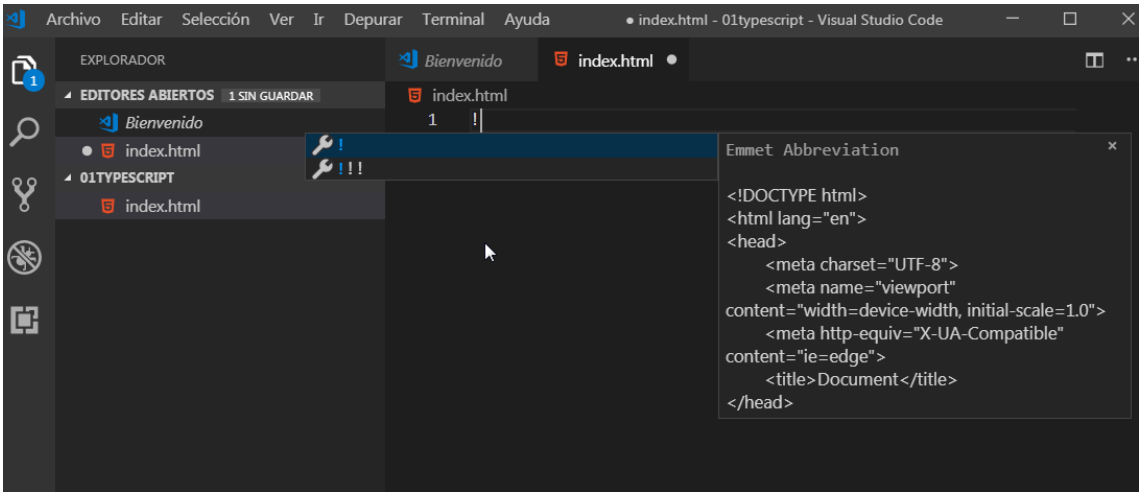


Escribimos index.html

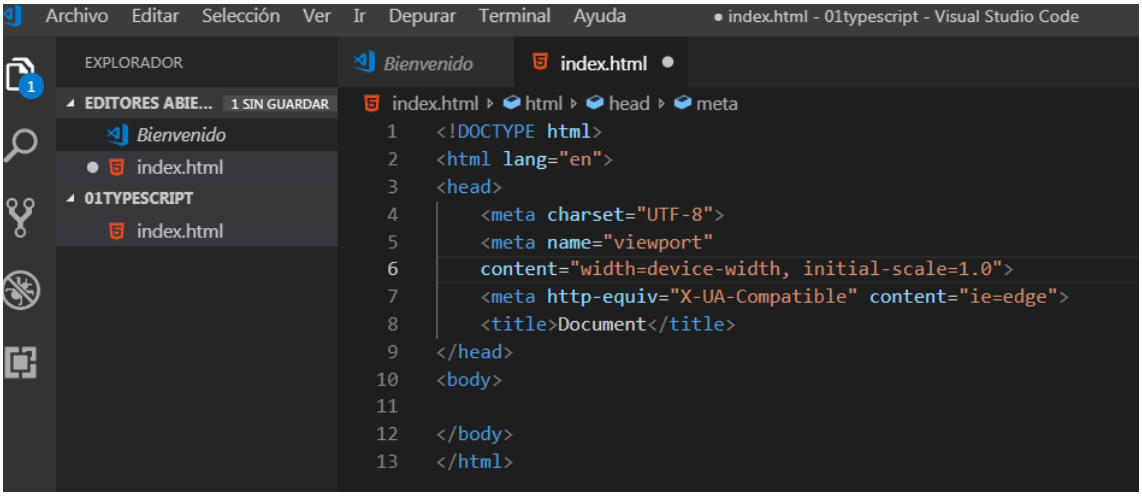
Tendremos algo así



En este punto, ya podemos escribir nuestro primer html, pero si hemos instalado las extensiones que habíamos sugerido en el apartado de instalaciones necesarias, veremos que nos facilita un poco la vida. Escribiemos ! y esperamos a que el Ide nos enseñe un trozo de código y nos ofrece esto:



Pulsamos la tecla “tabulador” y ya nos escribe este código html.



Si no lo escribe, pues bueno, es una plantilla básica de HTML5.

Vamos a poner en el body un h3 con un título, lo grabamos con CTRL+S o Archivo Guardar y lo probamos.

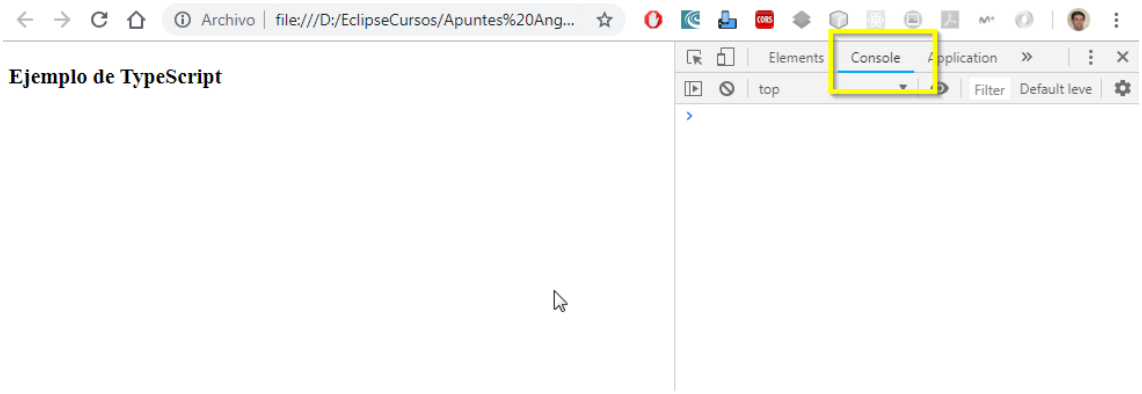
```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <h3>Ejemplo de TypeScript</h3>
</body>

</html>
```

Para probarlo, navegamos a la carpeta y damos doble click sobre el archivo index.html

Conviene abrirlo con Google Chrome porque utilizaremos las herramientas de desarrollo del mismo.

Lo que vamos a hacer ahora es mostrar las herramientas de desarrollo pulsando la tecla F12 y tendremos algo así.



Así es como tenemos que estar; viendo la consola.

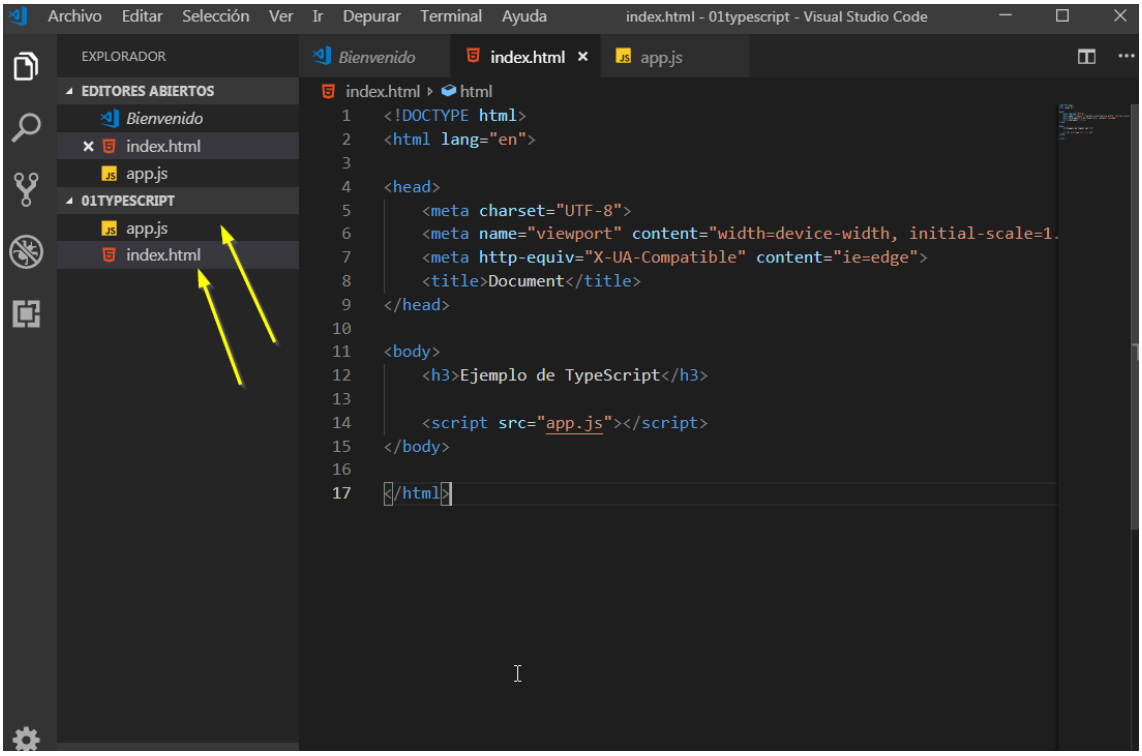
En nuestro ide, vamos a crear un archivo llamado app.js con el siguiente código

```
console.log('Codificando con JavaScript!!!');

function saludar(nombre) {

    console.log('Hola ' + nombre);
}

usuario = {
    nombre: 'Juan Carlos',
    apellidos: 'Fernández'
};

saludar(usuario);
```

y lo vamos a llamar desde nuestro index.html

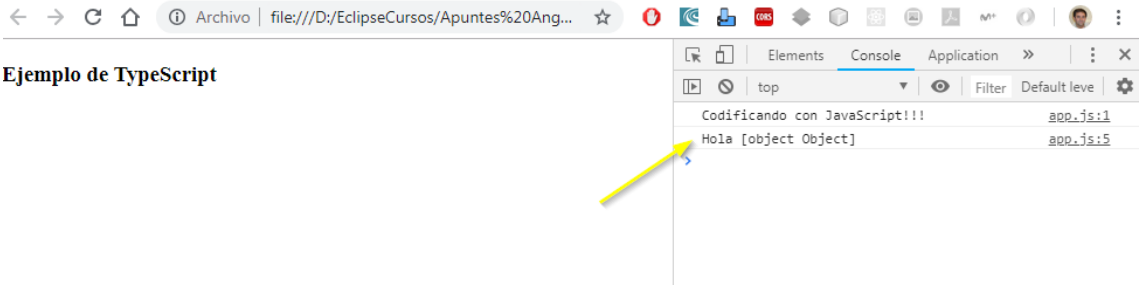
```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <h3>Ejemplo de TypeScript</h3>

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

</html>
```

Nuestro proyecto en el ide, tiene que tener esta pinta:



Si ahora volvemos a refrescar nuestro navegador veremos que no funciona correctamente, pero también que el ide no nos muestra ningún error.

Pero el error es muy básico, hemos hecho una función que esperaba un string y la hemos enviado un objeto.



Esperábamos que nos saludase.

Por lo tanto, tenemos esto

```
console.log('Codificando con JavaScript!!!');

function saludar(nombre) {

    console.log('Hola ' + nombre);
}

usuario = {
    nombre: 'Juan Carlos',
    apellidos: 'Fernández'
};

saludar(usuario);
```

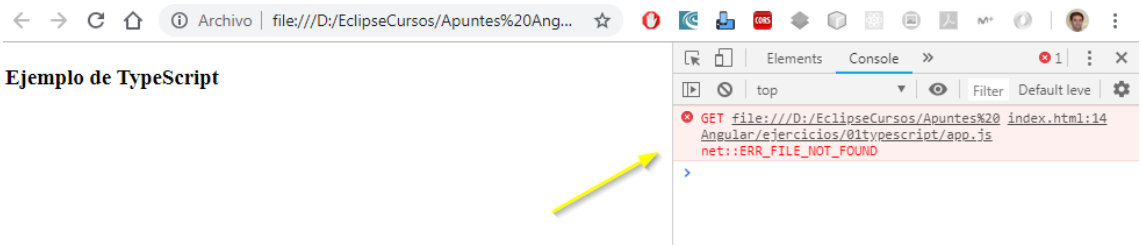
y además sin errores.

¿Cómo lo convertimos a TypeScript?

Vamos a cambiar el nombre del archivo app.js a app.ts y vamos a ver dos cosas.

Por una parte, ya nos muestra errores y podemos ver de qué errores se trata situándonos con el ratón encima.

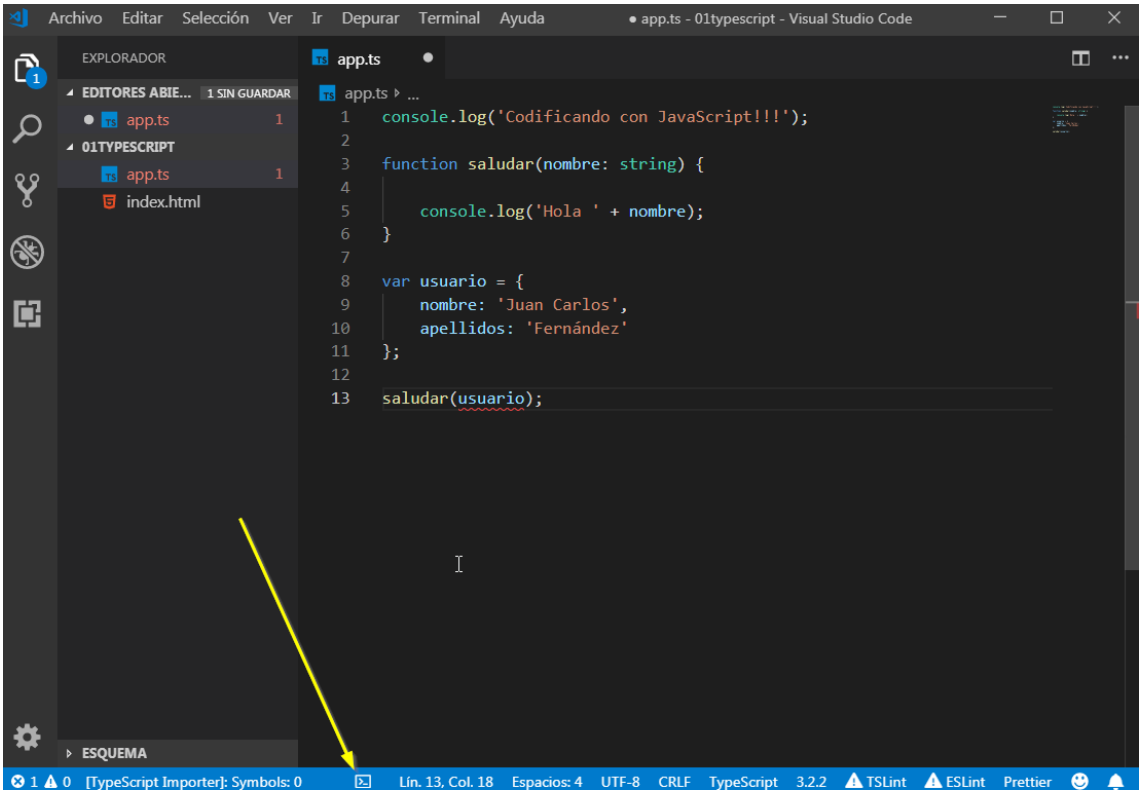
Por otra parte, si ahora actualizamos el navegador, nos va a dar un error en la consola, ya que no tenemos el js que es lo que entiende. El navegador no entiende de ts.



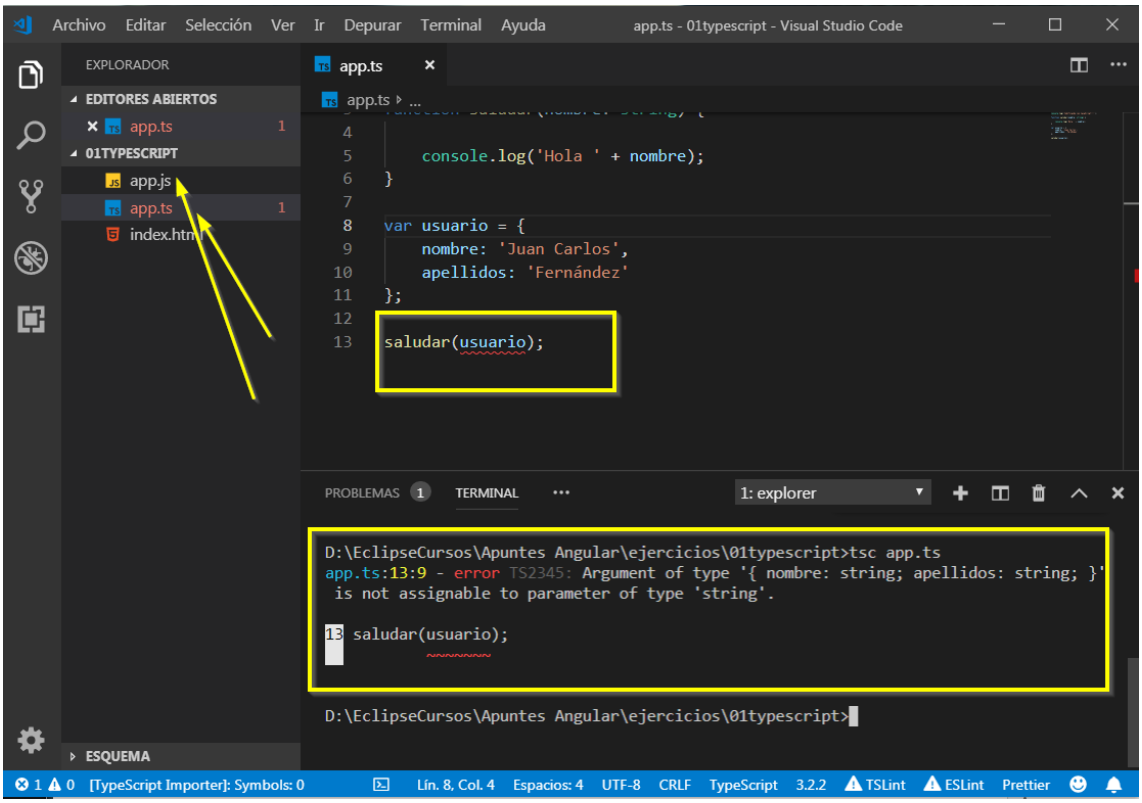
Pero esto este tipo de errores con javascript puede ser frecuente y TypeScript nos ofrece un tipado fuerte para las variables. De tal manera que nuestra función deberíamos decir que esperamos un nombre de tipo string, además, vamos a nombra nuestro objero con la palabra reservada var, para que lo encuentre; para declarar esa variable

Así es como lo he dejado y nos está diciendo en la llamada a la función que espera un string y le estamos mandando otra cosa. Pero vamos a compilar con este error a js para ver que es lo que pasa.

Para hacerlo, tenemos que estar con la consola de comandos en la carpeta del proyecto o abrir en VS una terminal



Para compilar, en la consola de comandos, escribiemos tsc app.js



En esta pantalla vemos.

a. Nuestro fichero app.ts
b. Nuestro fichero transpilado app.js
c. Y el error que nos muestra la tranpilación informando que la función saludar espera un string y le estamos enviando un objeto.

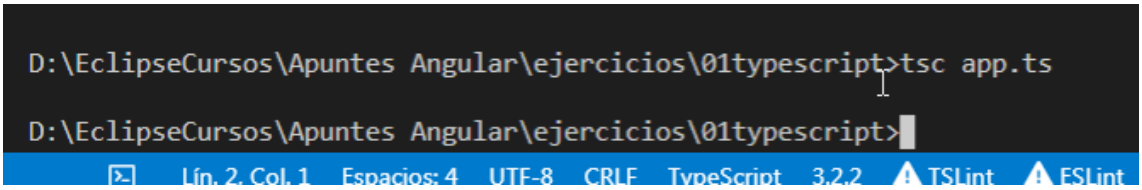
   Esta información de los errores en la compilación es básica en el desarrollo.

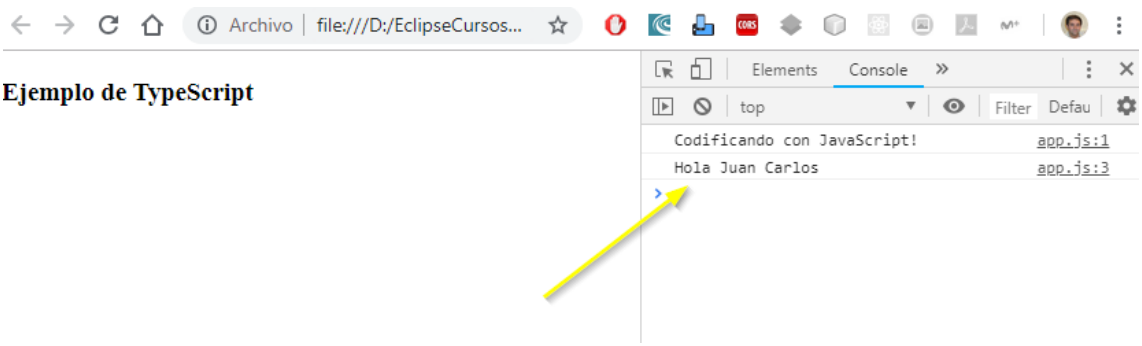
   Pero vamos a arreglarlo
   Corregimos el código de nuestro app.ts para que quede así:

```
console.log('Codificando con JavaScript!!!');

function saludar(nombre: string) {

    console.log('Hola ' + nombre);
```

```
}

var usuario = {
    nombre: 'Juan Carlos',
    apellidos: 'Fernández'
};

saludar( usuario.nombre );
```

y volvemos a compilar.
Ya no vemos errores y además





Ya funciona . (Lo mismo tienes que refrescar varias veces el navegador por el caché del js, pero veremos que esto no será problema)

Para no tener que estar compilando cada app.ts, podemos escribir en la consola de comandos tsc app –w y cada cambio que se haga en app.ts se recompila y mostrará si hay errores.

Podemos cambiar alguna cosa para ver que está funcionando.

Ahora, vamos a borrar nuestro ts y en nuestro js vamos a poner este código para **mostrar otro error habitual que tenemos con javascript.**

app.js

```
console.log('Variables en JavaScript!');

var numero = 6;

if (true) {
    var numero = 'Hola variables!';
}

console.log(numero);
```

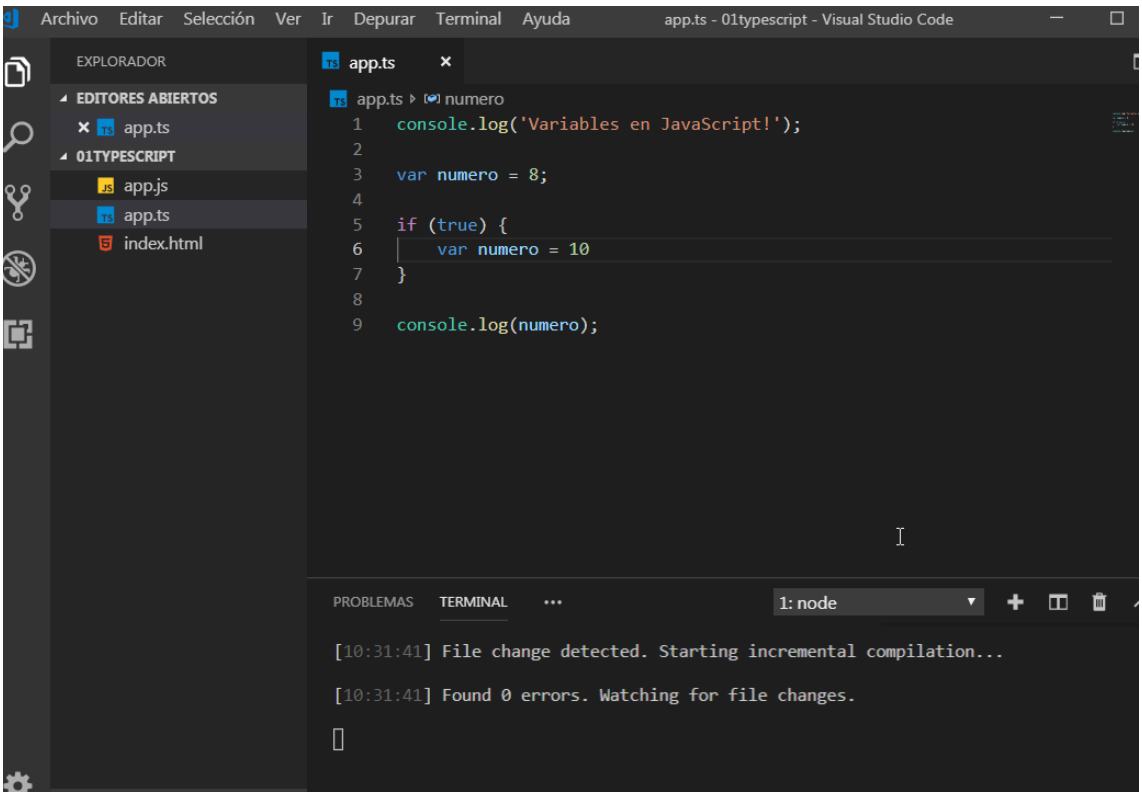
Cuando ejecutamos este código, vemos que el resultado que imprime en consola es

Hola variables .

¿Qué es lo que pasa?. En javascript puedes declarar una variable arriba y luego más abajo cambiarlo y esto es un follón.

Este mismo app.js, lo vamos a renombrar a app.ts y lo vamos a compilar, pero además, vamos a dejar el tsc escuchando los cambios para no tener que compilar todo el rato con la opción –w.

También vamos a cambiar un poco el app.ts y quedará todo así:



¿Qué pasa si ejecutamos este código?

Nos va a sacar un 10. Esto es un problema porque si en algún contexto de un bloque de código tipo if, se cambia o inicializa otra vez la variable, se lía.

Hasta aquí es todo javascript, pero vamos a cambiar var por let y miramos lo que sucede.

Vemos que imprime 8.

(En cualquier momento podemos abrir el js para ver lo que está generando el tsc)

¿Qué es lo que pasa?

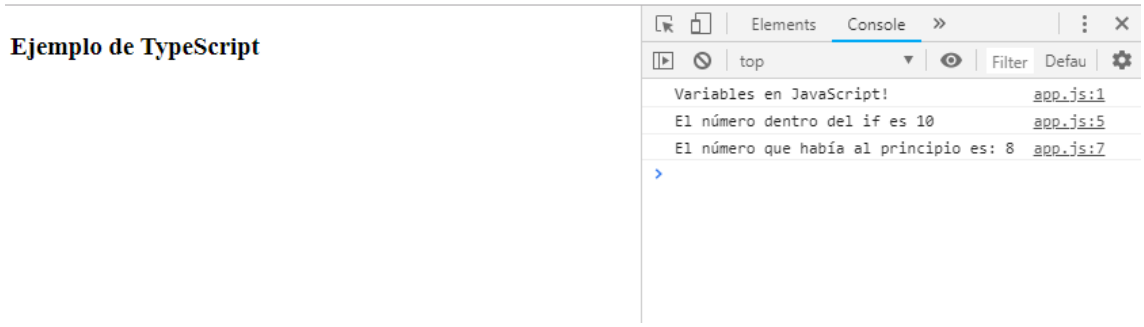
Pues que la variable numero dentro del if tiene ese valor, pero solo dentro del if.

Aprovecho este ejercicio para explicaros el concepto de “templates literales”.

Lo he puesto dentro del if y lo que se utiliza son backticks ``

```
console.log('Variables en TypeScript!');

let numero = 8;

if (true) {
    let numero = 10;
    console.log(`El número dentro del if es ${ numero } `);
}

console.log(`El número que había al principio es: ${ numero }`);
```

¿Qué sale ahora?



Como se puede ver sale el valor del número en su contexto. Luego el cambio de var por let ha hecho que dentro del scope if ha creado otra variable y la ha destruido cuando el bloque ha terminado.

```
console.log('Variables en TypeScript!');

let numero = 8;

if (true) {
    let numero = 10;
    console.log(`El número dentro del if es ${ numero } `);
}

console.log(`El número que había al principio es: ${ numero }`);
```

Incluso nos da un error al principio informando de que no se puede declarar dos veces la misma variable.

Es por esto que siempre utilizaremos let para declarar variables.

Por otra parte, utilizaremos const para declarar las constantes y como buena práctica, les pondremos el nombre de esas constantes en mayúsculas.

Funciones de flecha

No me voy a liar mucho, voy a poner una función normal y la misma como una función de flecha y además una ventaja de usar funciones de flecha.

```
let dimeNumero = function ( a ) {
    return a;
}


let dimeNumeroFlecha = recibe => recibe ;

console.log( dimeNumero(8) );
console.log( dimeNumeroFlecha(8) );
```

se puede ver que una vez compilado queda así:

```
var dimeNumero = function (a) {
    return a;
};
var dimeNumeroFlecha = function (recibe) { return recibe; };
console.log(dimeNumero(8));
console.log(dimeNumeroFlecha(8));
```

En estos ejemplos se justifica un poco el uso de funciones de flecha.
Por cierto que también hay algo parecido en java 8.

```
/***** Otro ejemplo pero además le decimos el tipo de dato. */

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

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

```
console.log( suma ( 8,3 ));
console.log( sumaFlecha ( 8,3 ));

/**** Si tenemos que ejecutar líneas de código en la función  */
/* por ejemplo decir si un número es primo */

let n = 17;

let esPrimo = function ( numero: number )  {

    if ( numero == 1 || numero == 2) {
        return 'es primo';
    } else {
        for ( let x=2; x < numero ; x++ ){
            if ( numero % x  == 0 ) {
                return 'no es primo';
            }
        }
        return 'es primo';
    }

}

let esPrimoFlecha = ( numero: number ) => {
    if ( numero == 1 || numero == 2) {
        return 'es primo';
    } else {
        for ( let x=2; x < numero ; x++ ){
            if ( numero % x  == 0 ) {
                return 'no es primo';
            }
        }
        return 'es primo';
    }
}


console.log(`El número ${ n } ${esPrimo( n ) }.` );
console.log(`El número ${ n } ${esPrimoFlecha( n ) }.` );


/* Pero para qué vienen bien las funciones de flecha??.. */
```

```
/* un objeto */

let carroAmx30 = {
    nombre: "AMX 30",
    dispara() {
        console.log( this.nombre + ' ha disparado');
    }

}


// Llamamos la función del objeto
carroAmx30.dispara();

// vamos a poner en otro carro de combate una timeout
// para que haga el disparo


let leopard = {
    nombre: "Leopard",
    dispara() {
        setTimeout( function (){
            console.log( this.nombre + ' ha disparado');
        }, 1500 );
    }

}

leopard.dispara();
// ojo que no sale el this.nombre.....
// y es que el this, no hace referencia al nombre del objeto,
// se puede resolver con función de flecha y funciona.

let leopard2 = {
    nombre: "Leopard 2",
    dispara() {
        setTimeout( () => {
            console.log( this.nombre + ' ha disparado');
        }, 1500 );
    }

}
```

```
leopard2.dispara();
// ahora si sale bien el nombre del Leopard 2E
```

Promesas del EMS 6.

Ejecutar una tarea, cuando ha terminado otra asíncrona.
Lo vemos con un ejemplo…

```
let promesa = new Promise( function ( resolve, reject ) {

        setTimeout( ()=> {
            console.log('Tarea Terminada');
            // si termina bien, llamamos a reolve y
            // si sale mal, pues reject y en lugar then con catch
            resolve();
        }, 1500 );
});


console.log('Comienza');

promesa.then( function() {
    console.log('Se hace si todo va bien');
});

console.log('Termina');
```

Creo que el código es bastante descriptivo. De todas formas, ya sabéis donde estoy...

Pasamos en el próximo fichero a crear nuestra primera aplicación Angular.