Clase 3: Material Complementario

Sitio:	Centro de E-Learning - UTN.BA	Imprimido por:	Nilo Crespi
Curso:	Curso de Backend Developer - Turno Noche	Día:	Friday, 23 de January de 2026, 10:15
Libro:	Clase 3: Material Complementario		

Tabla de contenidos

1. TypeScript

Typescript es un lenguaje de programación de código abierto creado por el equipo de Microsoft como una solución al desarrollo de aplicaciones de gran escala con Javascript dado que este último carece de clases abstractas, interfaces, genéricos, etc. y demás herramientas que permiten los lenguajes de programación tipados. Son ejemplos la compatibilidad con el intellisense, la comprobación de tiempo de compilación, entre otras.

A typescript se lo conoce además como un superset (superconjunto) de JavaScript ya que es un lenguaje que transpila el fuente de un lenguaje a otro pero, incluye la ventaja de que es verdaderamente orientado a objetos y ofrece además, muchas de las cosas con las que estamos habituados a trabajar los desarrolladores como por ejemplo: interfaces, genéricos, clases abstractas, modificadores, sobrecarga de funciones, decoradores, entre otras varias.

"Los superset compilan en el lenguaje estándar, por lo que el desarrollador programa en aquel lenguaje expandido, pero luego su código es "transpilado" para transformarlo en el lenguaje estándar, capaz de ser entendido en todas las plataformas" (desarrolloweb.com)

Entonces, si posees algo de conocimiento de Javascript, ¡tienes ventaja! Podemos cambiar los archivos de JavaScript a TypeScript de a poco e ir preparando la base del conocimiento para incorporar en su totalidad este nuevo lenguaje.

JavaScripts	TypeScripts
JavaScripts se ejecuta en el navegador. Es un lenguaje de programación interpretado.	Typescripts necesita ser "transpilado[1]" a JavaScript, que es el lenguaje entendido por los navegadores.
JavaScript se ejecuta en el navegador, es decir en el lado del cliente únicamente.	TypeScripts se ejecuta en ambos extremos. En el servidor y en el navegador.
Es débilmente tipado.	Es fuertemente tipado (tipado estático)
Basado en prototipos.	Orientado a objetos.

Tabla comparativa de los lenguajes JavaScript y TypeScript

Tipado estático

Una de las principales características de typescript es que es fuertemente tipado por lo que, no sólo permite identificar el tipo de datos de una variable mediante una sugerencia de tipo sino que además permite validar el código mediante la comprobación de tipos estáticos. Por lo tanto, TypeScript permite detectar problemas de código previo a la ejecución cosa que con Javascript no es posible.

Sugerencias para la escritura: Los tipos también potencian las ventajas de inteligencia y productividad de las herramientas de desarrollo, como IntelliSense, la navegación basada en símbolos, la opción Ir a definición, la búsqueda de todas las referencias, la finalización de instrucciones y la refactorización del código, podes consultar la documentación oficial acá.

Se agrega además que Typescript permite describir mucho mejor el fuente ya que, además de ser tipado, es verdaderamente orientado a objetos (avanzaremos en esto más adelante) lo que permite a los desarrolladores un código más legible y mantenible.

Variables

Antes de avanzar en tipos y subtipos de datos en Typescript, recordemos el concepto de variable:

Una **variable** es un espacio de memoria que se utiliza para almacenar un valor durante un tiempo (scope) en la ejecución del programa. La misma tiene asociado un tipo de datos y un identificador.

Debido a que Typescript es un superset de Javascript, la declaración de las variables se realiza de la misma manera que en Javascript:

var: Es el tipo de declaración más común utilizada.

En este punto es importante agregar que, si bien TypeScript es muy flexible y puede determinar el tipo de datos implícitamente, es aconsejable utilizar la nomenclatura que recomienda la página oficial (tipado estricto), ya que de este modo permitirá un mejor mantenimiento de nuestro código y el mismo será más legible.

Sintaxis:

```
var medida= 10;
var m=10;
```

let: Es un tipo de variable más nuevo (agregado por la ECMAScript 2015). El mismo reduce algunos problemas que presentaba la sentencia var en las versiones anteriores de Javascript.

Este cambio permite declarar variables con ámbito de nivel de bloque y evita que se declare la misma variable varias veces, puedes profundizar en la documentación oficial haciendo clic acá

Sintaxis:

```
let precio=0;
```

const: Es un tipo constante ya que, al asumir un valor no puede modificarse

(agregado también por la ECMAScript 2015)

Sintaxis:

```
const ivaProducto = 0.10;
```

Nota: Como recordatorio, la diferencia entre ellas es que las declaraciones **let** se pueden realizar sin inicialización, mientras que las declaraciones **const** siempre se inicializan con un valor. Y las declaraciones **const**, una vez asignadas, nunca se pueden volver a asignar. (Referencia).

Inferencia de tipo en TypeScript

Typescript permite asociar tipos con variables de manera explícita o implícita como veremos a continuación:

Sintaxis de la inferencia explícita:

```
<variable>: <tipo de datos>
```

Ejemplo inferencia explícita:

```
let edad: number; = 42;
```

Ejemplo inferencia implícita:

```
let edad = 42;
```

Nota: Si bien las asociaciones de tipo explícitas son opcionales en TypeScript, se recomiendan dado que permiten una mejor lectura y mantenimiento del código.

Veamos cómo funciona:

1. Abrir VSCode y crear un nuevo archivo titulado **example01.ts** 2. En dicho archivo, escribir las

siguientes declaraciones de variables:

```
let a: number;  //* Inferencia explícita
let b: string;  //* Inferencia explícita
let c=101;      //* Inferencia implícita
```

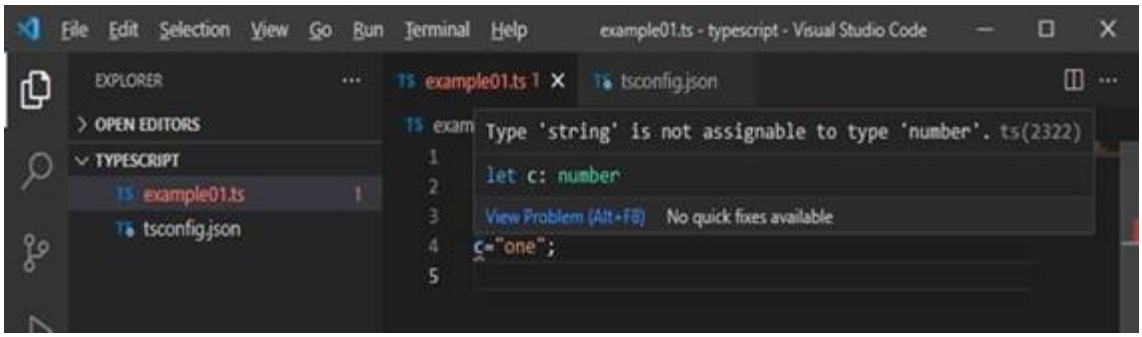
En este caso, typescript interpreta que la variable **a** es del tipo number y **b** del tipo string dado que la declaración es explícita. En el caso de la variable **c** infiere que es del tipo number dado que éste es el tipo de datos que corresponde al valor con el que se ha inicializado la variable.

Nota: Observa que al posicionar el puntero del mouse sobre la variable c, VSCode abre un tooltip con la declaración explícita "**let c:number**"

Pero ¿qué ocurre si intentamos asignar un tipo de datos diferente a la variable c?. Para ello, escribir a continuación la siguiente línea:

```
c="one";
```

VSCode marca un error en la línea de la asignación y en el explorador puedes ver el archivo en rojo. Observa además que al posicionar el puntero del mouse sobre la línea VSCode muestra el mensaje de error **"Type string is not assignable to type number"**

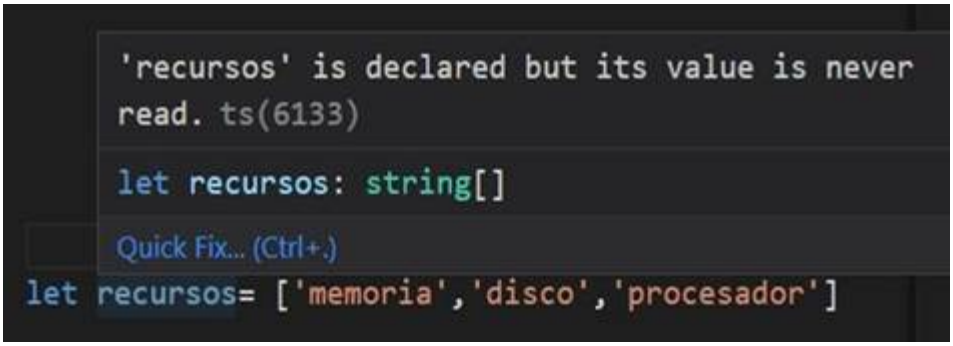


Error typescript. Asignación de tipos.

Analicemos otro ejemplo, dada la siguiente declaración:

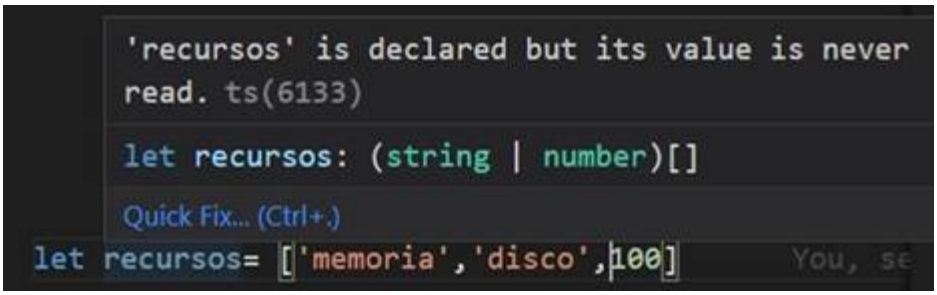
```
let recursos: ['memoria', 'disco','procesador'];
```

Hasta aquí sabemos que: "let", es la declaración de un arreglo cuyo nombre es "recursos" y el tipo no ha sido definido explícitamente.



Entonces, si posicionamos el puntero del mouse sobre la variable “recursos” podemos observar que Typescript automáticamente entiende por sí solo que se trata de un arreglo del tipo String.

Sin embargo, si luego introducimos en el array un valor de otro tipo, y posicionamos el puntero del mouse sobre la variable recursos, podremos observar que el arreglo admite variables del tipo “string” o (símbolo para “o” es “|”) “number”.



Y quizás, este no sea el comportamiento que deseamos por ello, se aconseja como buena práctica especificar el tipo de datos de manera explícita.

En este caso:

```
let recursos: string [] = ['memoria','disco','procesador']
```

De esta manera, si intentamos introducir un elemento number u otro tipo a nuestro arreglo, el compilador nos marcará un error.

Iniciando con TypeScript

¿Cómo instalar TypeScript?

Para ejecutar código en javascript necesitamos instalar:

- NodeJS, dado que el compilador de Typescript está desarrollado en NodeJS.
- TSC (Command-line TypeScript Compiler), herramienta que permite compilar un archivo TypeScript a Javascript nativo.

¿Cómo instalar NodeJS?

Para ello, seguir los siguientes pasos:

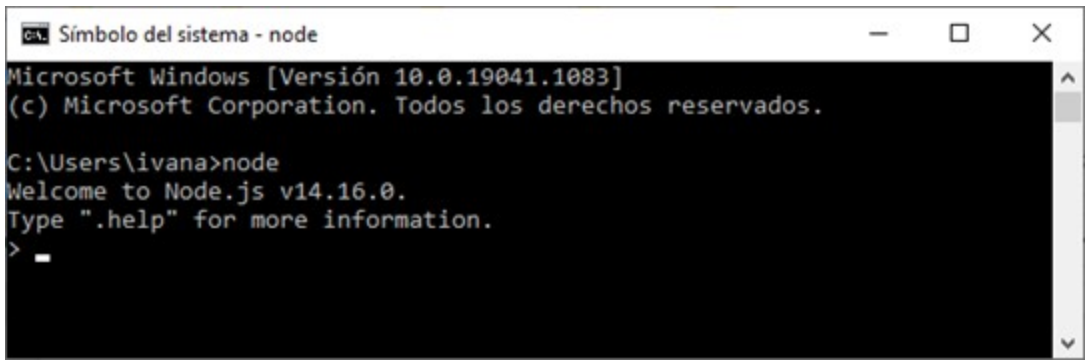
1. Descargar del sitio oficial el instalador acorde a su sistema operativo.

2. Instalar NodeJs.

a. Si tienes Windows o Mac, simplemente ejecuta el instalador y ¡listo!

b. Si tienes Linux, la página de instalación de NodeJS ofrece los comandos para instalar en Linux. Es relativamente sencillo.

3. Evaluar que la instalación fue exitosa. Para ello, ir a la línea de comandos del sistema e introducir el comando: **node**. A continuación, el sistema mostrará por pantalla la versión de NodeJS instalada como sigue:



Línea de comandos del sistema Windows después de ejecutar el comando “node”.

¿Cómo instalar TSC (Command-line TypeScript Compiler)?

La misma se realizará vía comando mpn como sigue:

1. Abrir la línea de comandos del sistema.

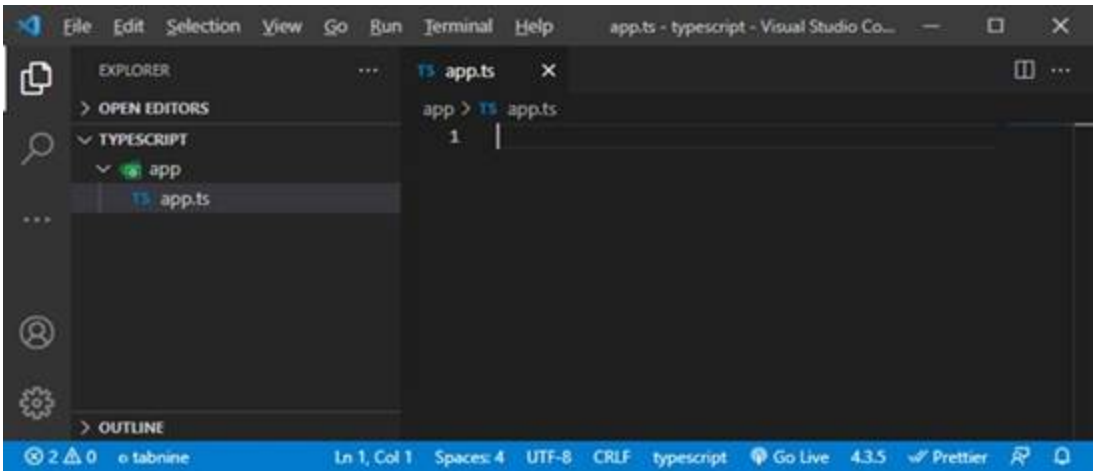
2. Ejecutar el siguiente comando: **npm install -g typescript**

3. Evaluar que la instalación fue exitosa. Para ello ejecutar el comando: **tsc -v**

Línea de comandos del sistema luego de ejecutar el comando tsc -v

¿Cómo crear y compilar un archivo Typescript en VSCode?

1. Desde el explorador de archivos, crear un nuevo archivo titulado: **app.ts** dentro de una carpeta app (opcional) como sigue:

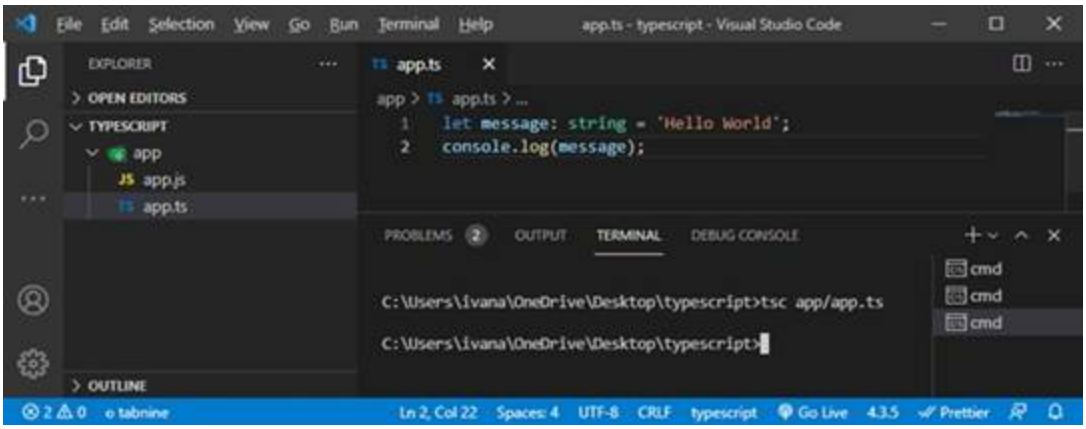


Creación del archivo helloworld.ts

2. Luego, escribir el código typescript en dicho archivo:

```
let message: string = 'Hello World';
console.log(message);
```

3. Finalmente, haciendo uso de la terminal de VSCode ejecutar el comando: **tsc app/app.ts.** Este comando compila y si no hay ningún error, crea el nuevo archivo js.



Compilación de Typescript a Javascript

Comandos y opciones del compilador CLI de tsc

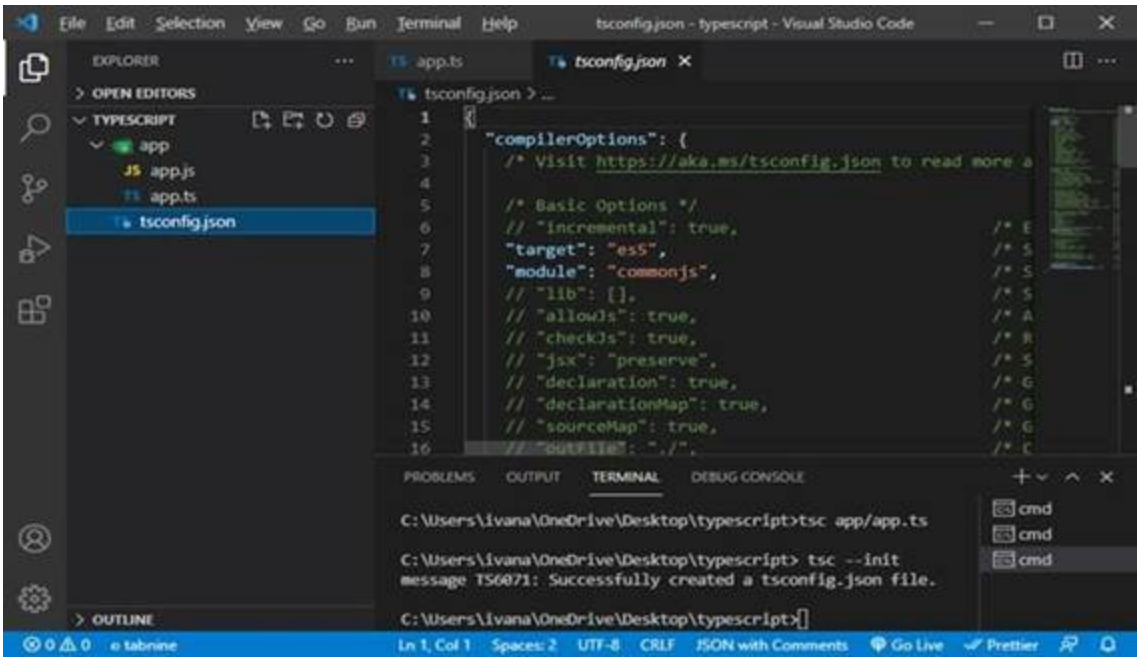
“Las opciones del compilador permiten controlar cómo se genera el código JavaScript a partir del código TypeScript de origen. Puedes establecer las opciones en el símbolo del sistema, como haría en el caso de muchas interfaces de la línea de comandos, o en un archivo JSON denominado tsconfig.json.

Hay disponibles numerosos comandos para las opciones del compilador por línea

de comandos. Puedes ver la lista de opciones el sitio oficial haciendo clic acá, también te dejamos mas información sobre la compilación de TypeScript que puedes leer haciendo clic acá

Para modificar el comportamiento predefinido del TSC con VSCode:

1. Abrir la terminal.
2. Ejecutar el comando: **tsc --init.** A continuación, se creará el archivo **tsconfig.json**



Archivo tsconfig.json generado después de ejecutar el comando tsc --init

3. Editar las configuraciones según se requiera.

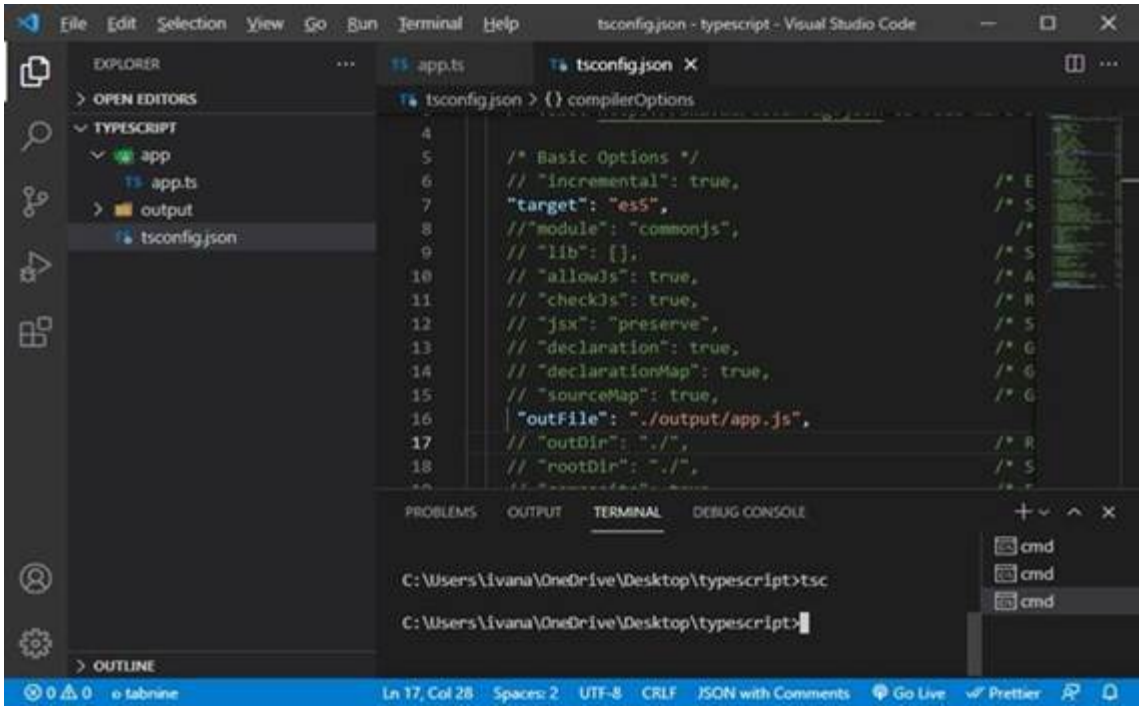
Por ejemplo, podemos crear una carpeta que contenga todos los archivos .js generados por el compilador TSC (el output dir/file). Para ello, descomentar la entrada “outFile” y a continuación ejecutar como sigue:

```
"outFile": "./output/app.js",
```

y comentamos la entrada “module”:

```
//"module": "commonjs",
```

y finalmente, ejecutar en la terminal de VSCode el comando **“tsc”**. A continuación, se creará la carpeta **output** conteniendo el archivo **app.js**.

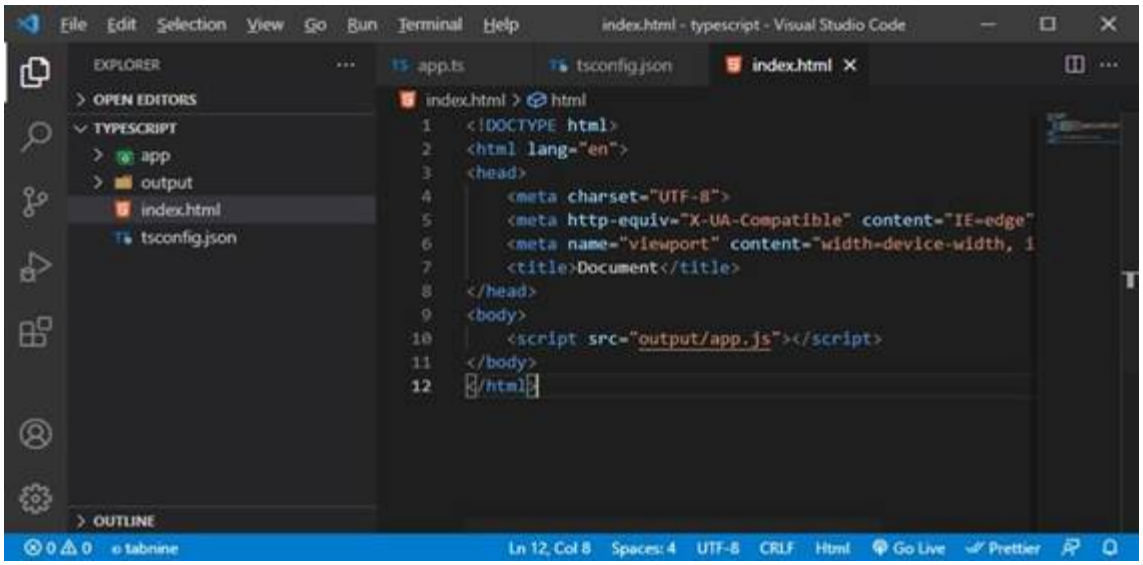


Manipulando la configuración del TSC para que tire los archivos generados a una carpeta output.

Hasta el momento hemos creado un archivo **app.ts** y lo hemos transpilado a un archivo equivalente en javascript **app.js** usando el compilador TSC pero, ¿cómo podemos ejecutarlo para ver los resultados en la consola?

Para ello, realizar los siguientes pasos:

1. Crear un archivo html que incluya el script app.js:



Archivo index.html

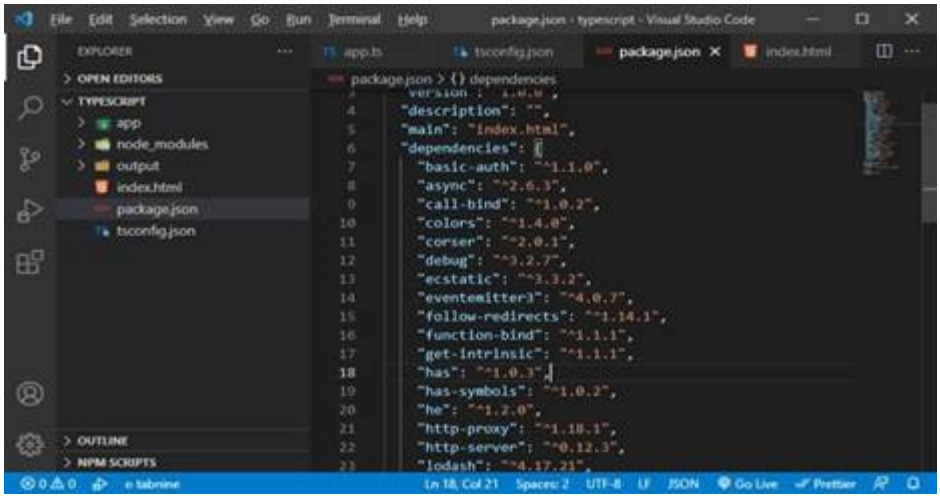
2. Ejecutar el archivo index.html o configurar un servidor de prueba para el entorno de desarrollo.

¿Cómo crear un servidor de pruebas en VSCode?

Para ello, seguir los siguientes pasos:

1. Ejecutar el siguiente comando **“npm install --global http-server”**. A continuación, se creará la carpeta **node_modules**.

2. Ejecutar el comando **“npm init”**. A continuación, se creará un archivo **package.json**



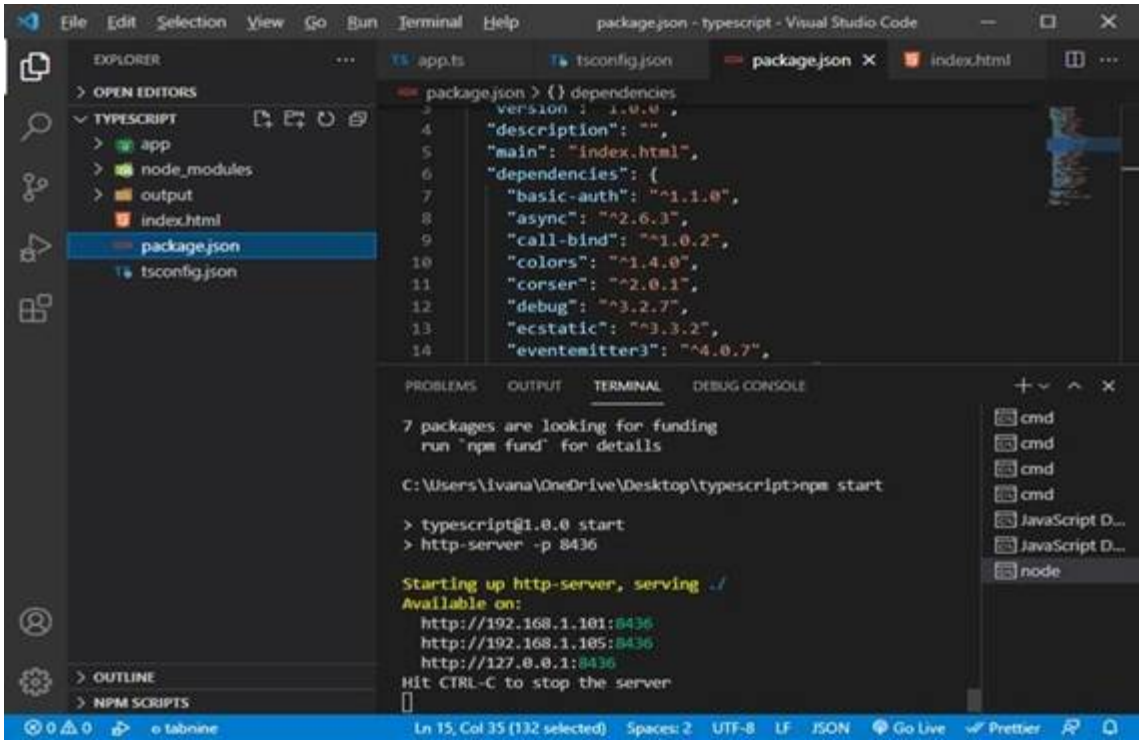
Archivo package.json

Nota: El archivo package.json es un archivo que contiene todos los metadatos acerca del proyecto. Son ejemplos: descripción, licencia, autor, dependencias, scripts, entre otros.

3. Configurar la entrada “scripts” como sigue:

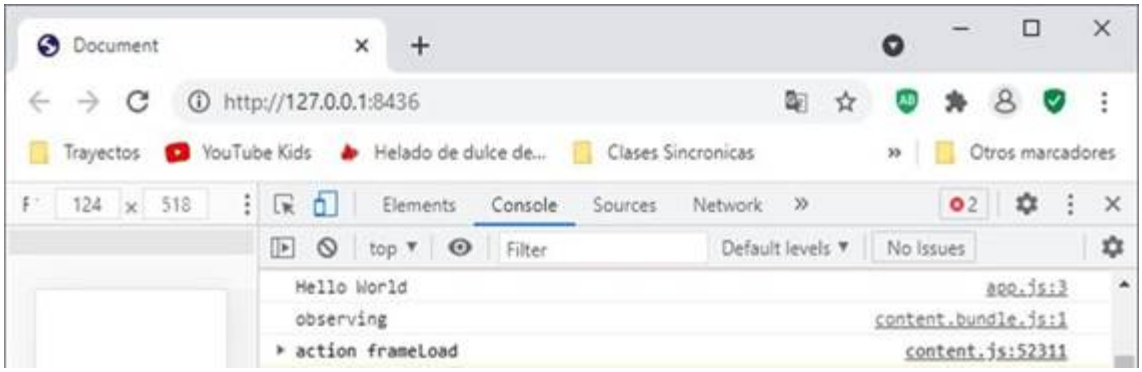
```
"scripts": {
  "start": "http-server -p 8456"
}
```

4. Finalmente, ejecutar el comando **“npm-start”** :



Iniciando el servidor.

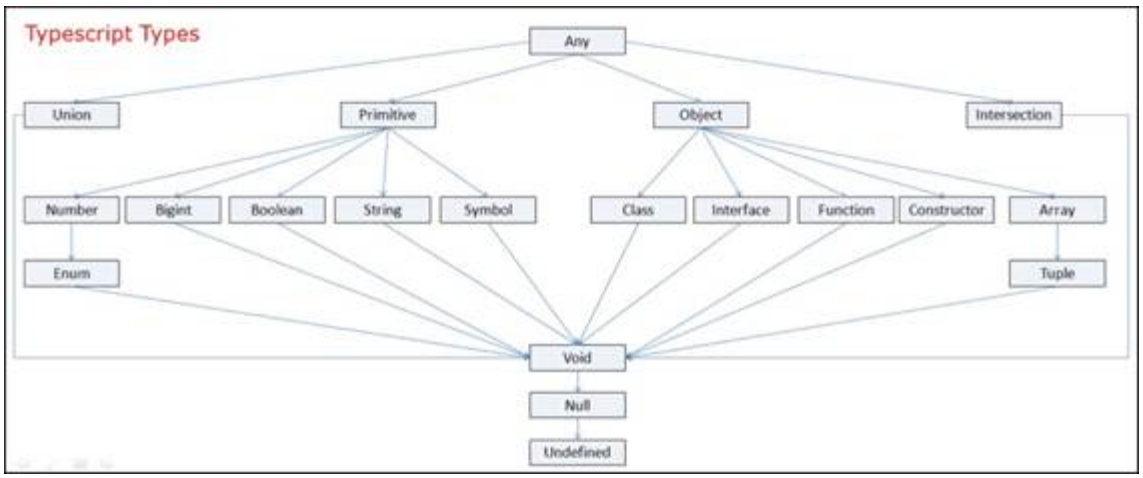
Como podemos observar en la Terminal de VSCode, accediendo a la url: http://127.0.0.1:8436 podremos visualizar nuestro html y, si inspeccionamos el fuente el mensaje “Hola Mundo” en la consola.



Inspección de código index.html

Tipos de datos y subtipos

Todos los tipos en TypeScript son subtipos de un único tipo principal denominado tipo any. Any es un tipo que representa cualquier valor de JavaScript sin restricciones.

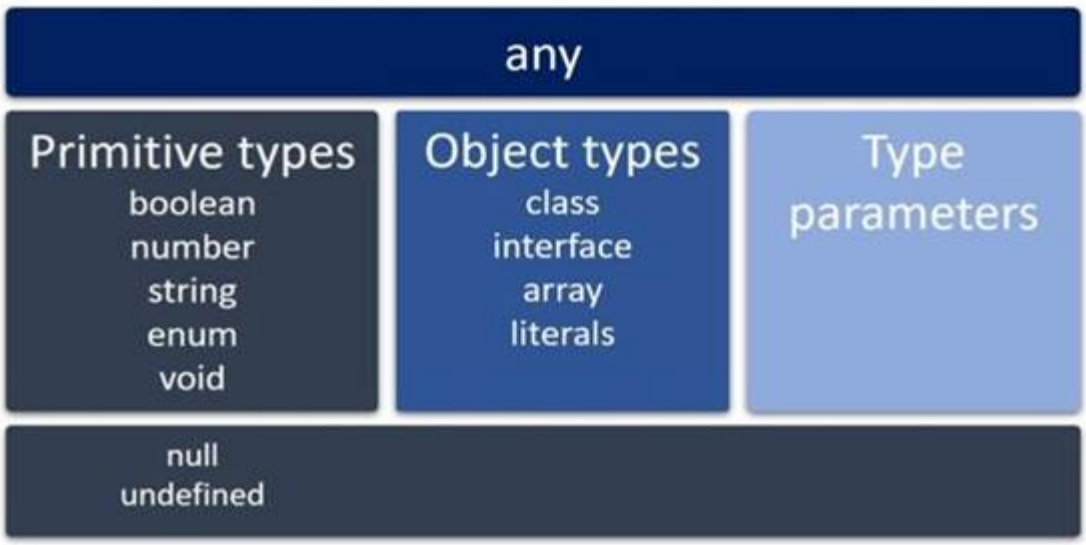


Any: Puede ser de cualquier tipo y su uso está justificado cuando no tenemos información a priori de qué tipo de dato se trata. Este tipo de definición es propia de TypeScript

Sintaxis:

```
let cantidad: any= 4;
let desc: any [] =[1,true,"verde"]
```

Todos los demás tipos se clasifican como primitivos, de objeto o parámetros. Estos tipos presentan diversas restricciones estáticas en sus valores.



Overview of types in TypeScript - Learn. (s. f.). Microsoft Docs. Recuperado 11 de octubre de 2021, de https://docs.microsoft.com/enus/learn/modules/typescript-declare-variable-types/2-types-overview

Tipos de datos primitivos

Los tipos primitivos son: boolean, number, string, void, null, undefined y enum.

string: Representa valores de cadena de caracteres (letras); Sintaxis:

```
let saludo: string = "hola, mundo";
```

TypeScript permite también usar plantillas de cadenas con las que podemos intercalar texto con otras variables: ${ expr } Ejemplo:

```
let nombre: string = "Mateo";
let mensaje: string = `Mi nombre es ${nombre}.
    Soy nuevo en Typescript.`;
console.log(mensaje);
```

number: Representa valores numéricos, como enteros (int) o decimales (float).

Sintaxis:

```
let codigoProducto: number = 6;
```

boolean: Es un tipo de variable que puede tener solo dos valores, Verdadero (true) o Falso (false).

Sintaxis:

```
let estadoProducto: boolean = false;
```

Void: El tipo void existe únicamente para indicar la ausencia de un valor, como por ejemplo en una función que no devuelve ningún valor.

Sintaxis:

```
function mensajeUsuario(): void {
 console.log("Este es un mensaje para el usuario");
}
```

Enum: Las enumeraciones ofrecen una manera sencilla de trabajar con conjuntos de constantes relacionadas. Un elemento enum es un nombre simbólico para un conjunto de valores. Las enumeraciones se tratan como tipos de datos y se pueden usar a fin de crear conjuntos de constantes para su uso con variables y propiedades.

Siempre que un procedimiento acepte un conjunto limitado de variables, considere la posibilidad de usar una enumeración. Las enumeraciones hacen que el código sea más claro y legible, especialmente cuando se usan nombres significativos" (referencia)

Ejemplo:

```
//**Crear la enumeración */
enum Color {
    Blanco,
    Rojo,
    Verde
}

//**Declarar la variable y asignar un valor de la enumeración */
let colorAuto: Color= Color.Blanco;

console.log(colorAuto); //return 0
```

Tipos de objetos

Los tipos de objeto son todos los tipos de clase, de interfaz, de arreglos y literales.

Array: Es un tipo de colección o grupos de datos (vectores, matrices). El agrupamiento lleva como antecesor el tipo de datos que contendrá el arreglo.

Sintaxis:

```
let list : string[]=['pimiento','papas','tomate'];
let rocosos:  boolean[] = [true, false, false, true]

let perdidos: any[] = [9, true, 'asteroides'];

let diametro: [string, number] = ['Saturno', 116460];
```

Generic.: También puedes definir tipos genéricos como sigue. Sintaxis:

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

Los genéricos son como una especie de plantillas mediante los cuales podemos aplicar un tipo de datos determinado a varios puntos de nuestro código. Sirven para aprovechar código, sin tener que duplicarlo por causa de cambios de tipo y evitando la necesidad de usar el tipo "any".

Los mismos se indican entre “mayores y menores” y pueden ser de cualquier tipo incluso clases e interfaces.

Veamos el ejemplo que nos provee la fuente oficial de typescript:

Si tenemos la siguiente función:

```
function identity(arg: number): number {
  return arg;
}
```

Pero, necesitamos que la misma sea válida para otros tipos de datos entonces podríamos cambiar el tipo number por any como sigue:

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

Sin embargo, el tipo any permite cualquier tipo de valor por lo que la función podría recibir un tipo number y devolver otro. Entonces, estamos perdiendo información sobre el tipo que debe devolver la función. Para solucionarlo, y obligar al compilador que respete el mismo tipo (parámetros de entrada y salida) podemos utilizar genéricos.

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

Observa que cambiamos any por la letra **T**.

T nos permite capturar el tipo de datos por lo que el tipo utilizado para el argumento es el mismo que el tipo de retorno.

Object: Es un tipo de dato que engloba a la mayoría de los tipos no primitivos.

```
let persona:object={nombre:"Ana", edad:45}
```

Desestructuración: La desestructuración permite acceder a los valores de un array o un objeto.

Ejemplo - desestructuración de un objeto:

```
var obj={a:1,b:2,c:3};
console.log(obj.c);
```

Ejemplo - desestructuración de un array:

```
var array=[1,2,3];
console.log(array[2]);
```

Ejemplo - desestructuración con estructuración:

```
var array=[1,2,3,5];
var [x,y, ...rest]= array;
console.log(rest);
```

Observa que la sintaxis **...rest**, nos permite agregar más parámetros. En este caso el resultado en consola será: [3, 5]

Estructuración: Como se pudo observar en el apartado anterior, la estructuración facilita que una variable del tipo array reciba una gran cantidad de parámetros.

Ejemplo en funciones:

```
function rest(first, second, ...allOthers)

{console.log(allOthers);
}
```

Observa que la sintaxis **...allOthers** nos permite pasar más parámetros.

Luego, al llamar a la función con los siguientes parámetros:

```
rest('1', '2','3','4','5'); //return 3,4,5
```

Tipos null y undefined

“Los tipos null y undefined son subtipos de todos los demás tipos. No es posible hacer referencia explícita a los tipos null y undefined. Solo se puede hacer referencia a los valores de esos tipos mediante los literales null y undefined”

Aserción de tipos (As)

Una aserción de tipos le indica al compilador "**confía en mí, sé lo que estoy haciendo**". Se parece al casting en otros lenguajes de programación, pero no tiene impacto en tiempo de ejecución sino que le dice al compilador el tipo de datos en cuestión a fin de acceder a los métodos, propiedades, etc. del tipo de datos en tiempo de desarrollo.

Sintaxis (dos posibles):

```
// primera posibilidad
(nombre as string).toUpperCase();

// primera posibilidad
(<string>nombre).toUpperCase();
```

Funciones

Una función es un **conjunto de instrucciones** o sentencias que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente y se caracterizan porque:

- deben ser invocadas por su nombre.
- permiten **simplificar el código** haciendo más legible y reutilizable.

La declaración de una función consiste en:

- Un nombre
- Una lista de parámetros o argumentos encerrados entre paréntesis.
- Conjunto de sentencias o instrucciones encerradas entre llaves.

Sintaxis:

```
function nombre (parámetro1, parámetro2)
{
//**instrucciones a ejecutar */

}
```

Ejemplo:

```
function calcularIva (produtos:Producto[]):[number, number]{
  let total=0;
  produtos.forEach(({precio}) =>{
    total+= precio;
  });
  return [total, total*0.15];

}

// Clase Producto
class Producto {
  precio:number;

}
```

Nota: Puedes crear tus propias funciones y usarlas cuando sea necesario.