Índice

[**¿Por qué usar TypeScript?** 1](#_Toc126830276)

[**TypeScript vs. JavaScript** 1](#_Toc126830277)

[**¿Como se pueden encontrar errores?** 1](#_Toc126830278)

[**Static analysis (Acá es donde TS cumple su rol)** 1](#_Toc126830279)

[**Unit Tests** 2](#_Toc126830280)

[**Integration tests** 2](#_Toc126830281)

[**Code review** 2](#_Toc126830282)

[**Atrapando bugs** 2](#_Toc126830283)

[**Activando poderes de TypeScript en JavaScript** 2](#_Toc126830284)

[**El compilador/transpilador de TypeScript** 2](#_Toc126830285)

[**Compilación de archivos TypeScript desde Node.js** 3](#_Toc126830286)

[**Compilación a una versión específica** 4](#_Toc126830287)

[**Enviando compilación a una carpeta** 4](#_Toc126830288)

[**Deno: un entorno nativo para ambos lenguajes** 4](#_Toc126830289)

[**Veamos el TSConfig.json** 4](#_Toc126830290)

[**Creando un archivo TSConfig.json** 5](#_Toc126830291)

[**Compilación en TypeScript** 5](#_Toc126830292)

[**Compilación en tiempo real** 6](#_Toc126830293)

[**Qué es el tipado en TypeScript** 6](#_Toc126830294)

[**La flexibilidad de JavaScript** 6](#_Toc126830295)

[**Controlando la flexibilidad** 7](#_Toc126830296)

[**Tipos inferidos** 7](#_Toc126830297)

[**Inferencia de tipos** 7](#_Toc126830298)

[**Nombres de variables iguales** 9](#_Toc126830299)

[**Numbers** 10](#_Toc126830300)

[**Operaciones** 10](#_Toc126830301)

[**Uso de variables sin inicializar** 11](#_Toc126830302)

[**Conversión de números de tipo string a tipo number** 11](#_Toc126830303)

[**Binarios y Hexadecimales** 12](#_Toc126830304)

[**Consejo** 12](#_Toc126830305)

[**Booleans** 13](#_Toc126830306)

[**Strings** 13](#_Toc126830307)

[**Arrays** 15](#_Toc126830308)

[**Tipado de arrays en TypeScript** 16](#_Toc126830309)

[**Any** 17](#_Toc126830310)

[**Importancia del Any** 17](#_Toc126830311)

[**Tratar Any como un primitivo** 17](#_Toc126830312)

[**Union Types** 18](#_Toc126830313)

[**Una mejor práctica** 18](#_Toc126830314)

[**Alias y tipos literales** 18](#_Toc126830315)

[**Tipos Literales (Literal Types)** 19](#_Toc126830316)

[**Alias + Tipos Literales** 20](#_Toc126830317)

[**Null y Undefined** 20](#_Toc126830318)

[**Null y Undefined como tipo Any** 21](#_Toc126830319)

[**Union Types como emergencia** 21](#_Toc126830320)

[**Funciones** 21](#_Toc126830321)

[**Retorno de funciones** 24](#_Toc126830322)

[**Retornos tipados en TypeScript** 24](#_Toc126830323)

[**Objetos en funciones** 26](#_Toc126830324)

[**Objetos como tipos** 27](#_Toc126830325)

[**Módulos: import y export** 28](#_Toc126830326)

[**Export** 28](#_Toc126830327)

[**Import** 28](#_Toc126830328)

[**Nota** 29](#_Toc126830329)

[**Usando librerías que soportan TypeScript** 29](#_Toc126830330)

[**Usando librerías que NO soportan TypeScript** 30](#_Toc126830331)

# **¿Por qué usar TypeScript?**

* Es uno de los lenguajes de programación más queridos por la comunidad.
* Alta adopción.
* Podemos prevenir muchos errores de JS.
* Mejor experiencia de desarrollo.
* Menor cantidad de bugs.

# **TypeScript vs. JavaScript**

* **En JavaScript** te das cuenta de los errores del código una vez que lo estés ejecutando.
* **En TypeScript** con el análisis estático de código nos damos cuenta de los errores en el editor o a la hora de transpilar, por lo que evitamos que los errores lleguen a la ejecución.

Un programador de **TS** es distinto a uno de **JS**. El programador de TypeScript añade una capa de análisis de código estático a Javascript.

## **¿Como se pueden encontrar errores?**

En el libro **Software Engineering at Google** lo dividen en distintas etapas:

### **Static analysis (Acá es donde TS cumple su rol)**

* Corre en el editor.
* Puede encontrar tipos, llamados incorrectos a funciones, etc.
* Nos permite autocompletar código.

### **Unit Tests**

* En pocos segundos nos permiten verificar si nuestro código hace lo que pensamos que hace.

### **Integration tests**

* Toma unos minutos validar si nuestro sistema funciona.
* Podemos encontrar distintos casos límite.

### **Code review**

* Toma algunas horas validar que estamos siguiendo las prácticas y normas de nuestro equipo.

# **Atrapando bugs**

## **Activando poderes de TypeScript en JavaScript**

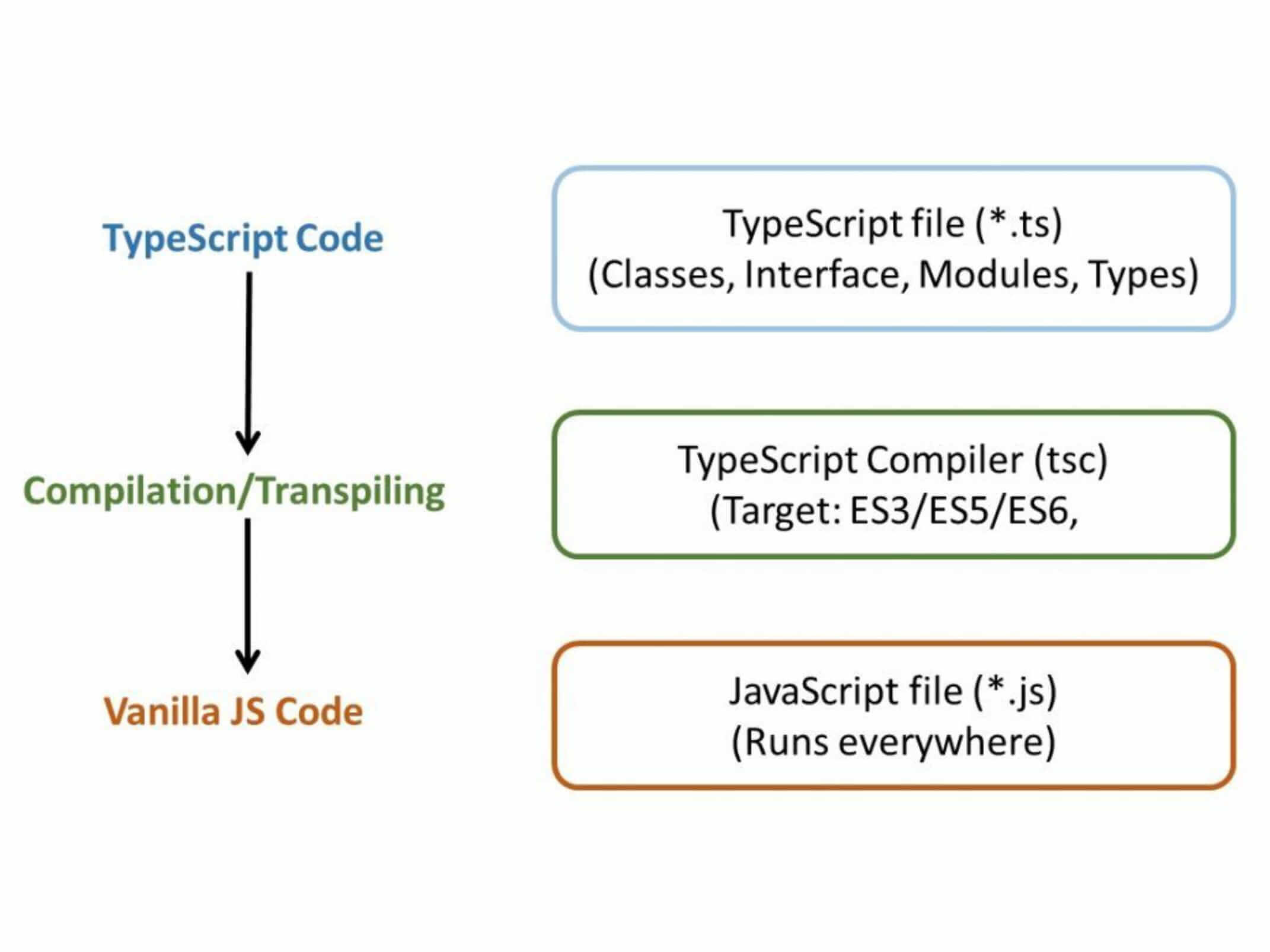
Si estás en Visual Studio Code, puedes activar el analizador de código estático de TypeScript sobre un archivo JavaScript. Para esto, en la primera línea del archivo debe ir lo siguiente:



# **El compilador/transpilador de TypeScript**

Nuestro navegador y Node no reconocen nativamente archivos TypeScript, lo que hace el transpilador es traducir el código de **.ts** a **.js**.

En nuestros archivos **.ts** vamos a tener las clases, interfaces, módulos, tipos. El transpilador nos va a convertir el código en **.js** que va a poder correr en cualquier lugar. Además, el transpilador nos permite elegir a que version de ECMAScript vamos a traducir nuestro código.



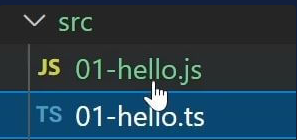
Para compilar un fichero, ejecutamos.

## **Compilación de archivos TypeScript desde Node.js**

Para realizar el proceso de transpilación en Node.js, ejecutemos lo siguiente en la terminal:



Tras esto, se creará un archivo JavaScript dentro de la misma carpeta donde este tu archivo TypeScript y con el **mismo nombre**. Por ejemplo, en nuestro proyecto realizamos esa operación dentro de la carpeta src con el archivo **01-hello.ts**, dando como resultado:



## **Compilación a una versión específica**

Podemos hacer que nuestro archivo TypesSript sea transpilado a un archivo JavaScript, por ejemplo, con el estándar ECMAScript 6. Para ello ejecutemos:



## **Enviando compilación a una carpeta**

Si deseas que los archivos transpilados no se generen en la misma carpeta donde están tus archivos TypeScript, puedes indicarle al compilador hacia donde quieres que vayan:



También podrías indicar que deseas aplicar la anterior operación a **todos** los archivos con extensión TypeScript:



## **Deno: un entorno nativo para ambos lenguajes**

[Deno](https://deno.land/), del mismo creador de Node.js, es un nuevo entorno de ejecución para JavaScript que puede correr también nativamente TypeScript. Sin embargo, aún no tiene la madurez en el ecosistema de Node.js

# **Veamos el TSConfig.json**

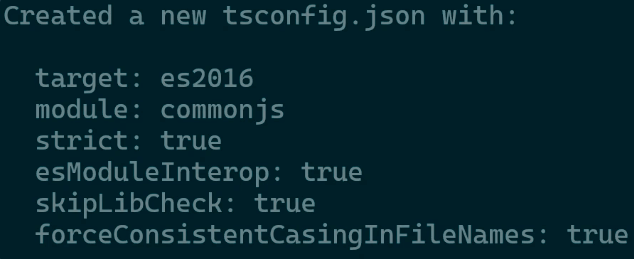
Nos ayuda a ahorrar mucho trabajo manual como transpilar archivo por archivo, indicar el target, etc.

## **Creando un archivo TSConfig.json**

En la terminal, ubicándonos dentro del directorio en el que queremos que se cree el archivo, ejecutemos:



Nos creará automáticamente el archivo con propiedades básicas activadas:



Dentro del archivo **TSConfig.json** podemos ver que tiene muchas propiedades comentadas (desactivadas) y de las cuales solo algunas están activadas.

## **Compilación en TypeScript**

Nuestro código TypeScript se transpilará según las propiedades indicadas en nuestro archivo **TSConfig.json**:



## **Compilación en tiempo real**

Nos puede resultar tedioso estar ejecutando el comando anterior siempre después de escribir nuestro código. Para evitar esto, podemos hacer que el compilador esté detectando cada cambio que realicemos en nuestros archivos TypeScript y haga la transpilación de inmediato:

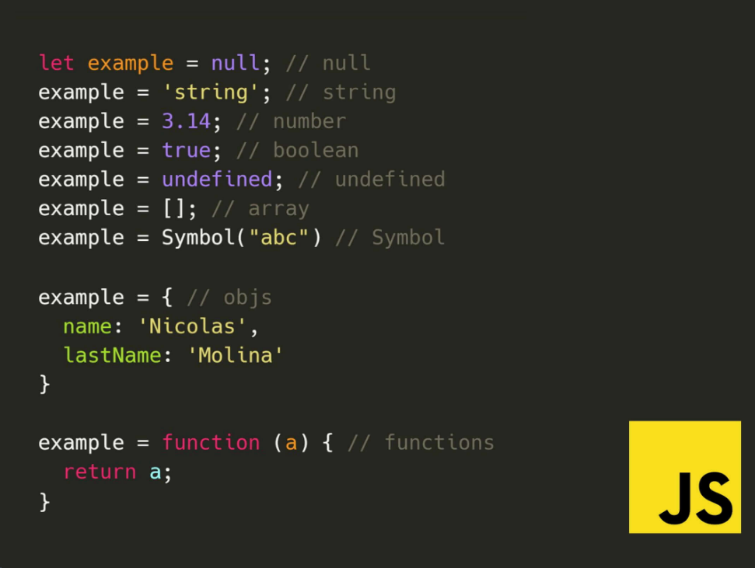


# **Qué es el tipado en TypeScript**

El tipado en TypeScript hace referencia a cómo declaramos una variable, necesitamos asignar el tipo de dato, conocido como **type annotation**, con esto evitamos mezclar distintos tipos de datos.

## **La flexibilidad de JavaScript**

Nosotros podemos declarar una variable de un tipo de valor y a lo largo del código ir cambiándolo si lo deseamos. Por lo que en un momento puede ser de tipo string y después de tipo boolean:



Para proyectos de software que tienen una gran escalabilidad, esto podría ser fuente de fallas en el programa.

## **Controlando la flexibilidad**

Gracias a TypeScript podemos manejar el tipado de las variables para evitar anomalías en el código.

En JavaScript, para declarar una variable constante lo realizamos así:



En TypeScript, para el caso anterior, es similar solo que añadimos **:** y el **tipo de dato de la variable**, la cual sería **number**. A esto último se le llama **type annotation** o **anotación de tipo**:



# **Tipos inferidos**

TypeScript puede inferir el tipo de dato de una variable a pesar de no haberlo declarado explícitamente.

## **Inferencia de tipos**

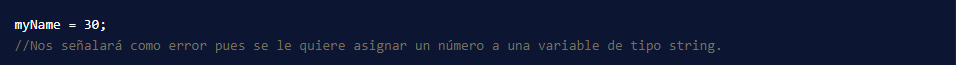
A partir de la inicialización de la variable TypeScript infiere el tipo que será a lo largo del código y este no puede variar. Por ejemplo:



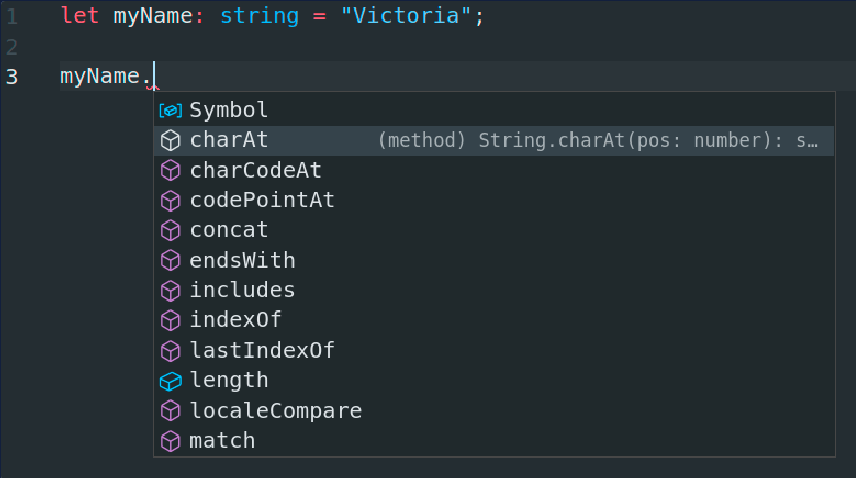
Si bien no indicamos el tipo de dato como se haría de esta manera:



TypeScript infiere que la variable **myName** será del tipo **string** y en adelante **no podrá** tomar un valor que no sea de este tipo de dato.



En Visual Studio Code puedes obtener autocompletado teniendo sugerencias según el tipo de dato que sea la variable:



## **Nombres de variables iguales**

TypeScript te indicará como error aquellas variables con el **mismo nombre** a pesar de estar en archivos distintos. Esto no sucederá en entornos preconfigurados como por ejemplo Angular o React, ya que estos trabajan de forma modular o tienen un alcance (scope) para cada variable.

Si deseas trabajar con los mismos nombres de variables en diferentes archivos, puedes crear una función anónima autoejecutada:



Lo mismo por cada variable que desees tener el mismo nombre (**myName** para este ejemplo) deberás crear este tipo de función para evitar que te den estas advertencias.

# **Numbers**

El tipo de dato **number** se usa para variables que contendrán números **positivos**, **negativos** o **decimales**.

## **Operaciones**

En JavaScript, una variable de tipo number puede fácilmente ser concatenado con otra de tipo **string**:



Sin embargo, esto podría llevar confusiones y errores durante la ejecución del programa, además de estar cambiando el tipo de dato de la variable. Por ello, en TypeScript solo se pueden hacer operaciones numéricas entre números valga la redundancia:



## **Uso de variables sin inicializar**

Serán señalados como errores aquellas variables que queramos usar sin haberles dado un valor inicial:



Señalar que, si no se va a inicializar aún la variable, definir explícitamente el tipo de dato, pues TypeScript no puede inferirlo si no tiene un valor inicial.

## **Conversión de números de tipo string a tipo number**

Para esto usaremos el método **parseInt**:

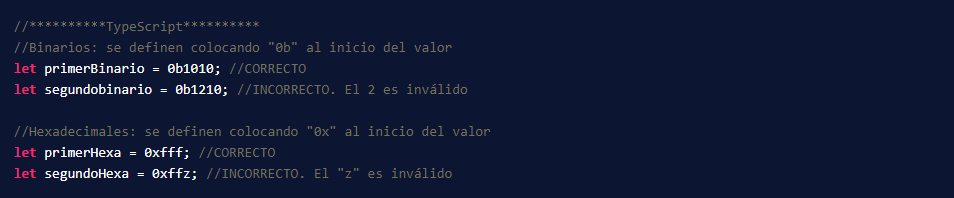


Esto funciona si el string tiene solo y exclusivamente números que no empiecen con **0**. De lo contrario, el resultado será de tipo **NaN** (Not a Number):



## **Binarios y Hexadecimales**

TypeScript nos puede indicar error si intentamos definir números binarios que tengan números que no sean 0 o 1 y si declaramos hexadecimales usando valores fuera del rango:



En consola, si están correctamente asignados, se hará una conversión a decimal de dichos números:



## **Consejo**

Cuando definas una variable de tipo de dato number, es preferible que el nombre de tipo sea en minúscula. Esto como buena práctica, pues se hará referencia al tipo de dato **number** y no al objeto **Number** propio del lenguaje:



# **Booleans**

Este tipo de dato puede tomar dos valores: **true** o **false**.



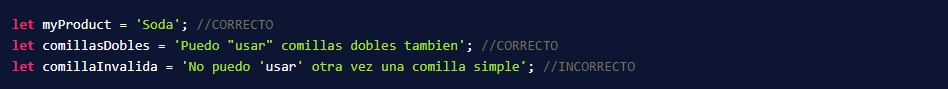
# **Strings**

Este tipo de dato nos permite almacenar una cadena de caracteres.

Podemos definir un **string** con:

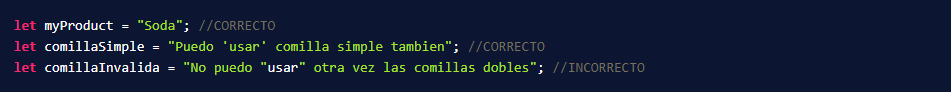
Comillas simples:

1. Comillas simples



Se pueden usar comillas dobles dentro, más no otra vez comillas simples.

1. Comillas dobles.



Se puede usar comillas simples dentro, más no otra vez comillas dobles.

1. Usando backticks.



Esta forma de asignar **string** trae algunas ventajas:

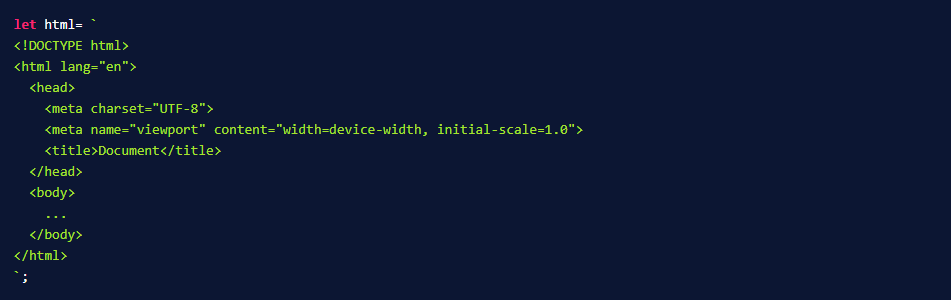
* + Declarar valores de múltiples líneas.



* + Concatenar dentro del mismo **string**. Para esto es necesario usar este símbolo del dólar seguido de llaves **${}** y escribir lo que queremos concatenar dentro de esas llaves.

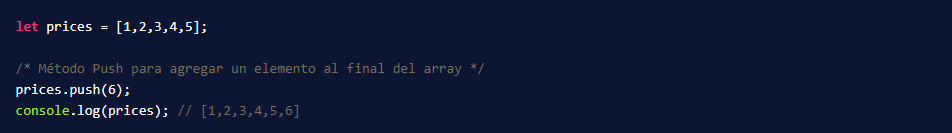


* + También respeta la indentación.

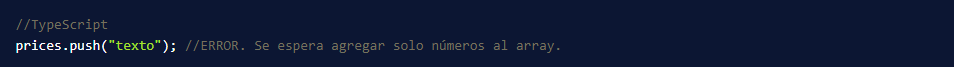


# **Arrays**

Es una colección de datos ordenada. Los definimos de la siguiente manera:



Para el array **prices**, TypeScript, de no indicarle explícitamente, va a inferir que este solo contendrá valores del tipo **number**, por lo que, si se quiere agregar un valor string, por ejemplo, nos indicará un error:



Esto debido a que en su inicialización se le asignó un array que solo contenía números.

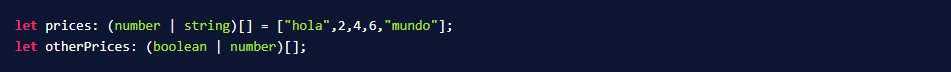
También nos indicará error si pretendemos hacer operaciones exclusivas de un tipo de dato sobre la de otro tipo:



## **Tipado de arrays en TypeScript**

Lo puedes definir así:

* + Indicar explícitamente los tipos de datos que almacenará el array:



Para este caso, a menos que la variable sea una constante, no es necesario que inicialices la variable, pues ya le indicaste el tipo de dato.

* + En la inicialización de la variable, colocar datos con el tipo de dato que quieres que soporte tu array en adelante para que lo pueda inferir TypeScript:



Dejamos claro que queremos que soporte los tipos de dato **string** y **number**.

# **Any**

Es un tipo de dato exclusivo de TypeScript. Su traducción sería “**cualquiera**”, pues literalmente nos permite almacenar cualquier tipo de dato en una variable:



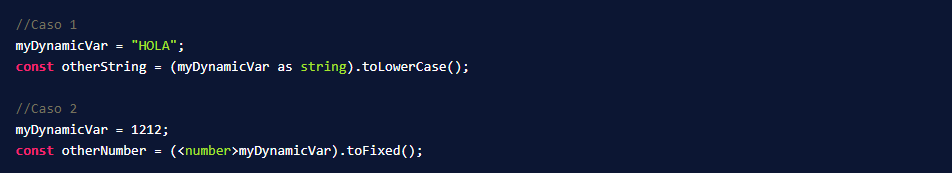
Se recomienda no usar este tipo de dato, pues **se considera mala práctica**.

## **Importancia del Any**

La utilidad de **any** radica cuando se quiere migrar de a pocos a TypeScript desde JavaScript, ya que incrementalmente definiríamos el tipo de dato donde sea necesario sin romper nuestro programa de golpe.

## **Tratar Any como un primitivo**

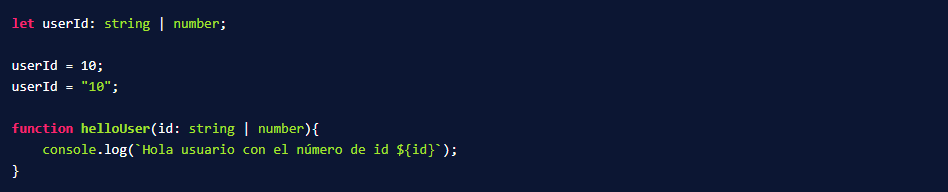
Se pueden realizar conversiones a tipos de datos primitivos de JavaScript:



Como observamos, podemos tratar nuestra variable **any** como **string** en el primer caso y como **number** en el segundo. Después de esto, podemos acceder a los métodos **toLowerCase**() y **toFixed**() según el tipo de dato correspondiente.

# **Union Types**

Nos permite definir **más de un tipo** de dato a una variable, argumento de una función, etc.



Aquí indicamos que **id** y **userId** pueden ser de tipo **string** o **number**.

## **Una mejor práctica**

El tipo de dato any nos brinda la flexibilidad de JavaScript en TypeScript con respecto al tipado. Sin embargo, si deseamos eso, es mejor hacer uso de los **Union Types**.

# **Alias y tipos literales**

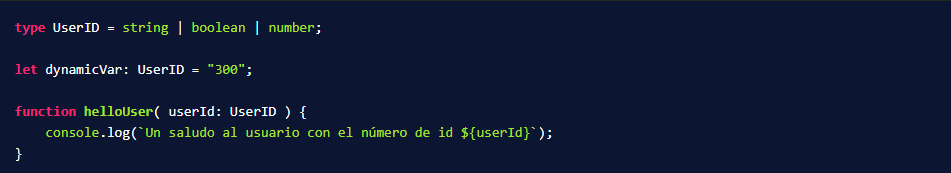
Los **Alias** nos permiten darle un nombre a uno o varios tipos de datos en conjunto. Un ejemplo de cómo se definen sería así:



¡Ahora **UserID** lo podemos usar como si fuese un tipo de dato **string**, **boolean** o **number**!



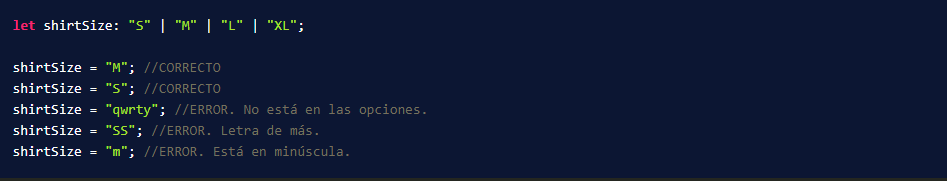
Los **Union Types** que vayamos a utilizar ahora serán menos tediosos de escribir, pues con los **Alias** podemos utilizar el mismo conjunto de tipos de datos en la definición de varias variables, beneficiándonos en escribir menos código.



**Nota**: La palabra **type** en los Alias es algo propio de TypeScript.

## **Tipos Literales (Literal Types)**

Gracias a esto podemos definir explícita y literalmente los posibles valores que puede tomar nuestra variable. Por ejemplo:



Definimos que la variable **shirtSize** pueda ser una de las 4 posibles opciones de valores, que estos sean de tipo **string** y que estén en mayúscula, por tanto, si queremos asignar un valor que no sea exactamente como lo declaramos, TypeScript nos mostrará un error.

## **Alias + Tipos Literales**

También podríamos combinarlas para facilitar aún más el desarrollo de nuestro programa:



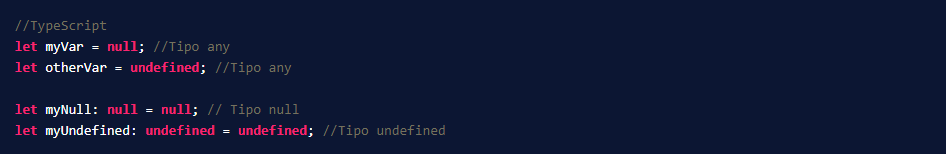
# **Null y Undefined**

Estos dos funcionan como dos tipos de datos, al igual que, por ejemplo, **string** o **number**.

El tipo de dato **null** es para indicar un valor **nulo** y **undefined** para algo indefinido. Son tipos diferentes.

## **Null y Undefined como tipo Any**

En TypeScript, si no especificamos que va a ser **null** o **undefined**, estos son inferidos como tipo any:



## **Union Types como emergencia**

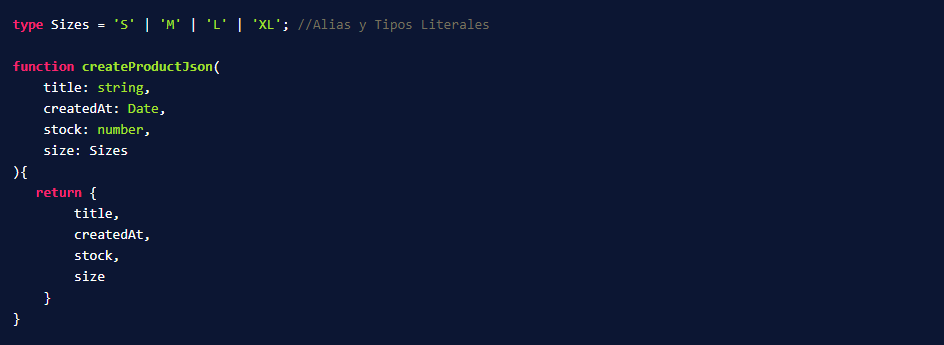
Hay casos en la que queremos que una variable sea de tipo string o number y que al inicializarlas sean de tipo null o undefined para luego asignarles un valor del tipo de dato de los primeros mencionados. En este contexto podríamos usar los **Union Types**:



# **Funciones**

Las funciones son nativas de JavaScript y esencialmente funcionan igual en TypeScript. Sin embargo, este último, con su sistema de tipado, nos ayudará a llevar a cabo una implementación más segura:

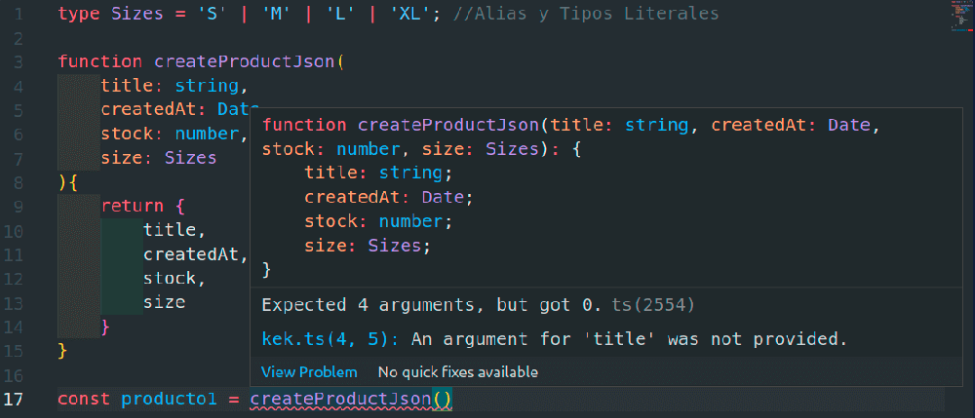
* + Podemos definir que los argumentos de la función tengan un determinado tipo de dato (o más de uno si se usa **Union Types**):



En el argumento **createdAt** se indica que es de tipo **Date** en alusión al objeto **Date** propio de JavaScript y no a un tipo de dato como **string** o **number**. Son diferentes las definiciones.

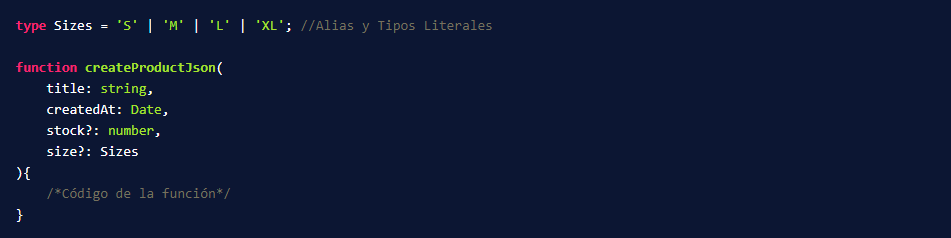
* + Cuando hagamos uso de nuestra función, TypeScript comprobará que le envíes todos los parámetros en orden y con el tipo de dato que se declaró en la función:



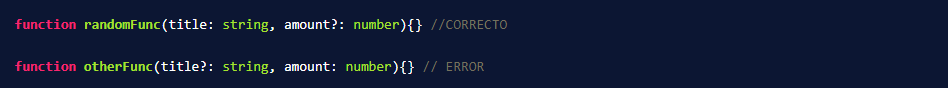


En **Visual Studio Code**, si dejas el cursor sobre el nombre de la función que vas a invocar, te mostrará un mensaje con los detalles de la función, lo que espera como parámetros y lo que devolverá indicando además el orden y el tipo de dato de cada variable.

* + Sí queremos que el argumento sea opcional de enviar, podemos usar el modificador **?** junto con el nombre del argumento.



**Nota**: cuando definamos argumentos opcionales en una función, estas **deben ubicarse al final**, si no TypeScript nos indicará un error, ya que podría haber confusiones al momento de invocar la función y enviar los respectivos parámetros.



# **Retorno de funciones**

En TypeScript podemos especificar el tipo de dato del valor que nos retornará una función o indicar si no se devolverá dato alguno:

## **Retornos tipados en TypeScript**

El tipo de retorno se especificará después de los paréntesis en los que se encuentran los argumentos de la función:

1. **Void:** **Funciones sin retorno**

Este tipo de función ejecuta ciertas instrucciones, pero no devuelve dato alguno. Estas son conocidas como funciones de tipo **void**. Se definen así:

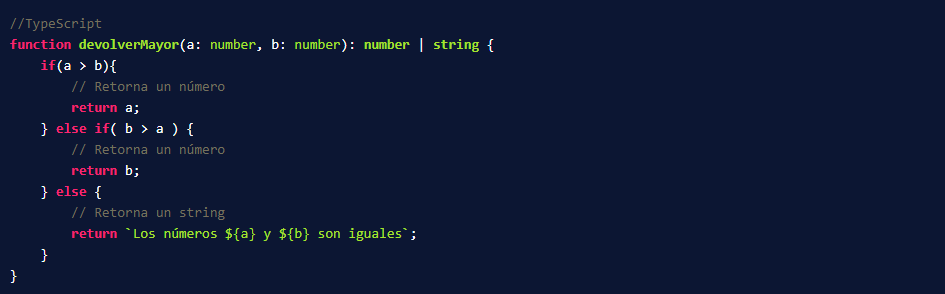


1. **Funciones con retorno**

Por el contrario, si en la función devolveremos algún valor, podemos especificar el tipo de dato de este:



También los retornos pueden ser más de un tipo de dato:

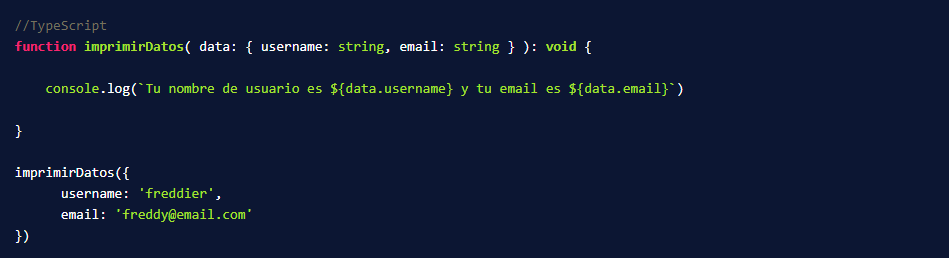


**TypeScript también lo infiere**

Si no indicamos en nuestra declaración de la función el tipado del retorno, TypeScript, al igual que con las variables, lo puede inferir según si retornas datos (sea **string**, **number**, etc.) o si nada es devuelto (tipo **void**).

# **Objetos en funciones**

Nuestras funciones pueden recibir objetos como argumentos. En TypeScript también podemos declarar el tipado de estos. Veamos un ejemplo:



En el ejemplo, el nombre **data** hace referencia al objeto que recibirá la función **imprimirDatos**. Por ello, para acceder al valor de **username** lo definimos en el **console.log** como **data.username** y para el email como **data.email**, pues así es como se accede a las propiedades de un objeto.

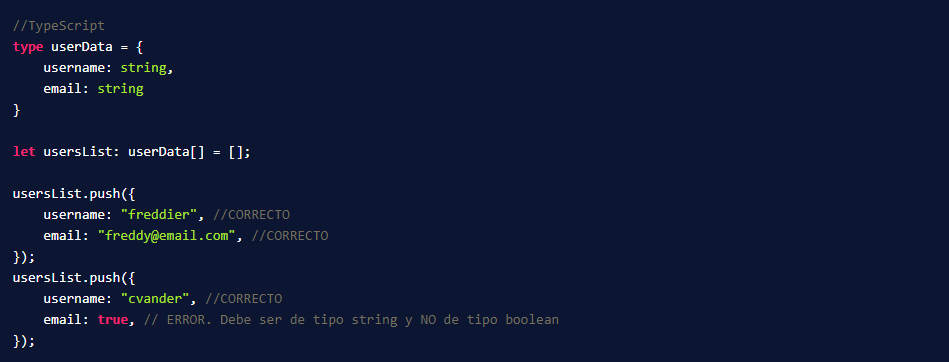
Finalmente, cuando invocamos **imprimirDatos** y queremos enviar el objeto que nos pide como parámetro, simplemente se colocará entre llaves los atributos del mismo sin colocar un nombre de referencia como data tal como lo hicimos en la definición de la función.

# **Objetos como tipos**

En TypeScript también podemos usar los Alias para definir la estructura de tipado que debería tener un objeto:



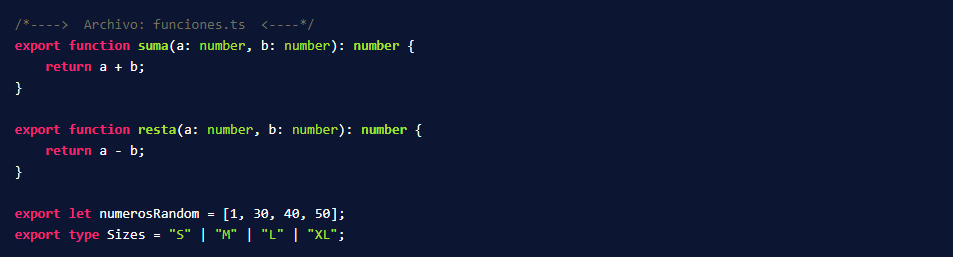
Y luego este “nuevo tipo” puede ser usado en un array, por ejemplo, para definir el tipado de los objetos que queramos añadir:



# **Módulos: import y export**

Nuestro código puede ser dividido en varios módulos (archivos), por lo que para poder usar las funciones o variables que existen en uno y queramos acceder desde otro, utilizamos import y export.

## **Export**



Como observamos, tenemos un archivo llamado **funciones.ts** la cual contiene dos funciones: **suma** y **resta**. Si estas queremos usarlas desde otro archivo, necesitamos usar la palabra reservada **export** justo antes de definir nuestra función (**lo mismo aplica para las variables**). De esta forma indicamos que serán exportados para ser utilizados desde otro archivo JavaScript/TypeScript.

## **Import**



Finalmente, las funciones o variables que queramos utilizar desde otro archivo son importadas de la siguiente manera:

* Usamos la palabra reservada **import**.
* Entre llaves indicamos las funciones y/o variables que queremos acceder. Hacemos una separación con comas.
* Usamos la palabra reservada **from**, seguido de, entre comillas dobles o simples, la ruta de la ubicación en la que se encuentra el archivo del cual estamos importando su código.

## **Nota**

Si es un módulo TypeScript lo que estamos importando, es importante que en la ruta de los **import** figure la extensión **.ts** de dicho archivo. Si es un archivo JavaScript, colocar la extensión **.js** es opcional.

# **Usando librerías que soportan TypeScript**

Las librerías que tienen soporte para TypeScript nos facilitan su uso, y más aún si usas editores de código que se integran bien con este “lenguaje”, pues brindan información muy útil como indicar:

* La cantidad de parámetros esperados por una función.
* El tipo de datos de los parámetros y variables.
* El tipo de dato que retornará la función.
* Autocompletado al usar métodos de un módulo.
* Mejores prácticas.

# **Usando librerías que NO soportan TypeScript**

El ecosistema de TypeScript ha creado unos módulos para agregar manualmente el tipado a las librerías que no tienen soporte de tipos.

Por ejemplo, si quieres trabajar con la librería [lodash](https://lodash.com/), en Visual Studio Code se te indicará que instales un sistema de tipos para que puedas desarrollar sin problemas desde TypeScript:

