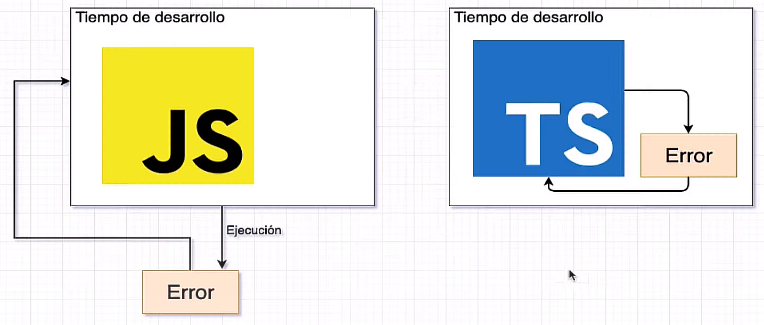
TypeScript

Para descargar TypeScript

npm install -g typescript

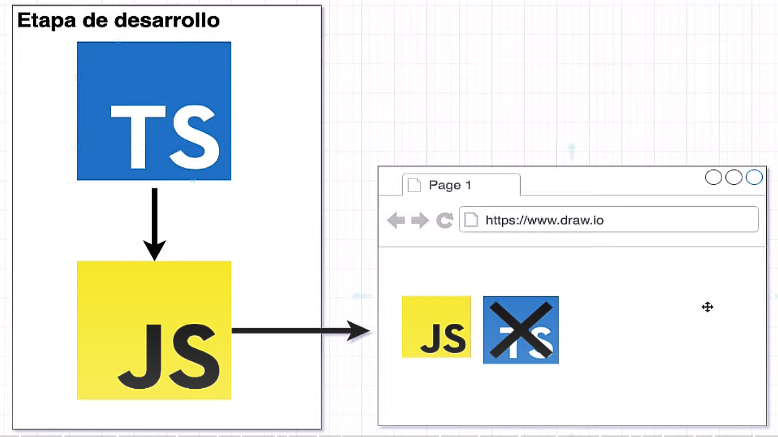
npm install typescript --save-dev

Cuando descargamos TS, lo que hacemos es descargar un compilador con TS.

Se puede ejecutar desde una Terminal en TypeScript el comando siguiente para compilar un código de TS o desde una ventana de comandos:

El comando para compilar es “**tsc**” y luego de ese comando le indicamos el nombre del archivo que queremos compilar, por su puesto que tenemos que estar en la carpeta donde está el archivo. Es decir, que para ejecutar el comando hay que estar dentro de su aplicación.

PS F:\TScript\first\_app> **tsc** .\index.ts

Si hay un error, TS nos avisa de tal error al momento de compilar.

Con TypeScript lo que se ejecuta realmente en el browser es código JS

Para que se ejecute el archivo de TS lo tenemos que llevar a JS para que finalmente se pueda ejecutar en el browser.

La compilación del archivo index.js genera un nuevo archivo de JS con el mismo nombre, pero con extensión JS index.js, para ejecutarlo se hace con el comando de **node** y el nombre del archivo desde la consola de comando:

>node index.js

Esto se puede ejecutar en la consola de comandos de la terminal de Visual Studio Code también

El resultado de ejecutar este código es 5, ya que 2 más 3 es 5.

CUANDO INTENTAS EJECUTAR POR PRIMERA VEZ EN VSTUDIO CODE el comando para compilación hay un error que puede corregirse de la siguiente manera.

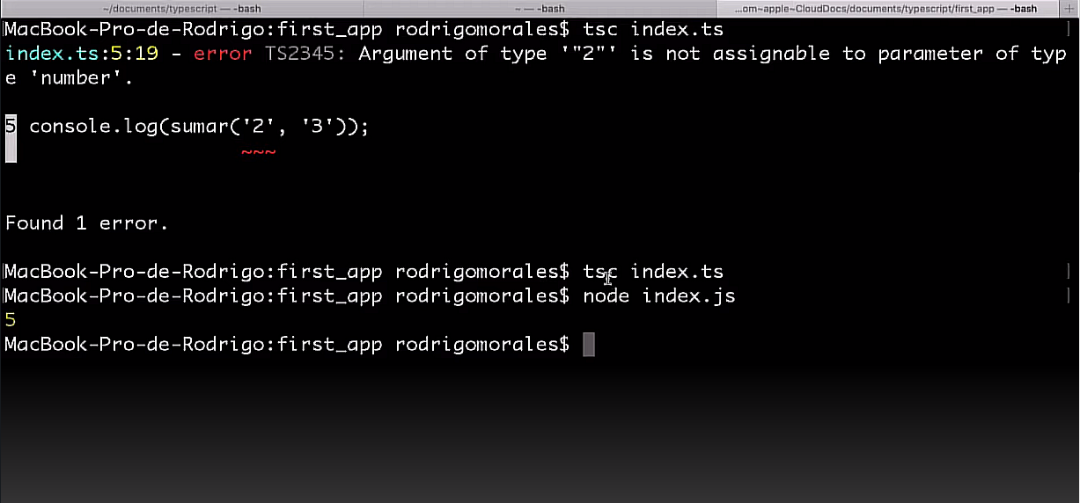
Problema con Scripts en Visual Studio Code

Ejecutar en la “terminal” el comando para compilar tsc y el nombre del archivo da un error.

PS F:\TScript\first\_app> **tsc** .\index.ts

**Error**: No se puede cargar el archivo C:\Users\admin\Documents\WindowsPowerShell\profile.ps1 porque la ejecución de scripts está deshabilitada en este sistema. Para obtener más información, consulta el tema about\_Execution\_Policies

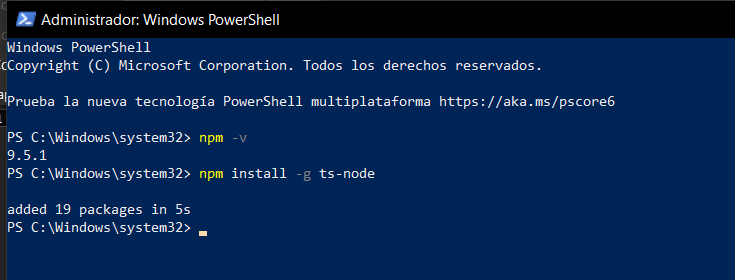
Para solucionar este error; debes ejecutar el **powershell** como administrador y ejecutar el comando **Set-ExecutionPolicy Unrestricted** esto solucionara tu problema y podrás *ejecutar scripts* en la consola de **VS Code**

Cuando se vuelva a ejecutar el comando no habrá más errores de este tipo. Y ahora si puede también ejecutar el comando para ejecutar el archivo

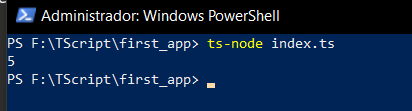
>node index.js

Proceso de compilación – “tsc” y ejecución “node” en imagen a la derecha.

Hay una forma más simple para evitar la aplicación de estos 2 comandos uniéndolos, y llevándolos a un solo comando. Esto se puede hacer a través de un paquete de “npm” que permite al usuario ejecutar archivos directamente de una sola vez. Básicamente nos permite combinar los 2 comandos; el comando para compilar y el comando para ejecutar un archivo en un solo comando. Se instala desde la consola de comandos de PowerShell, del siguiente modo:

npm install -g ts-node

el “-g” es parámetro que indica instalación de forma global a un paquete llamado “ts-node”, como se ve en la imagen a la derecha.

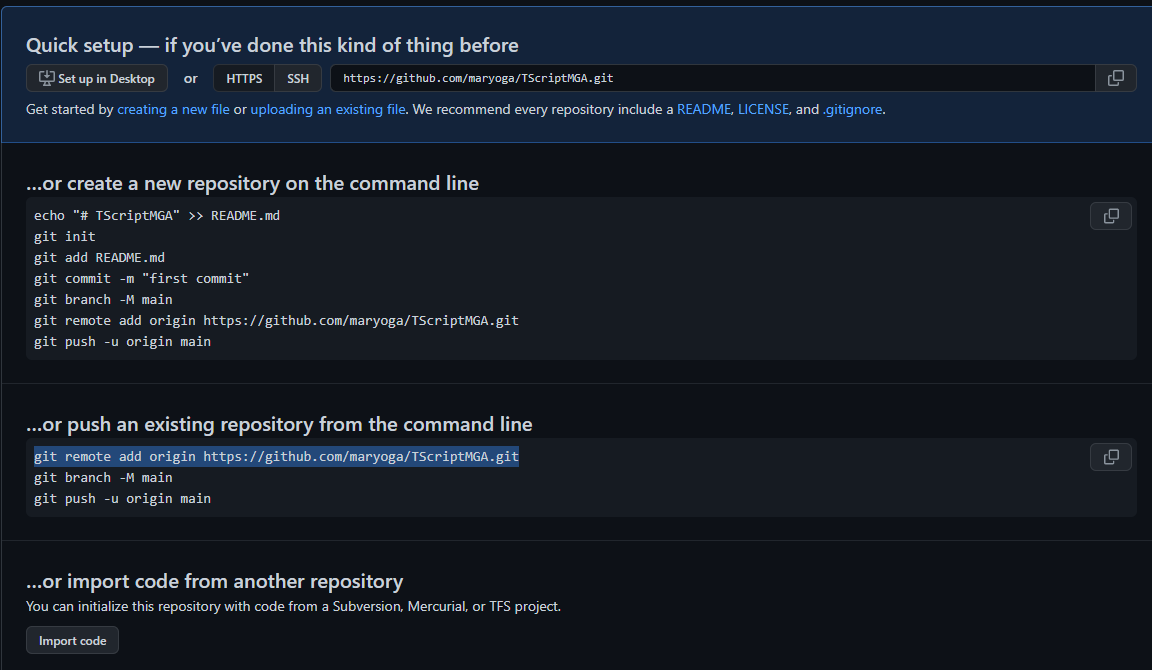
Ahora si quieres ejecutar un solo comando para poder llamar a una aplicación, simplemente llamar en la consola de PowerShelll a “ts-node” y llamamos al archivo de TypeScript en proceso de desarrollo, en este caso “index.ts”. Y lo que sucederá a continuación que aparecerá automáticamente el valor “5” que es el resultado de compilar y ejecutar el programa en un solo comando.

Para inicializar un repositorio local en VSCODE se puede hacer desde la terminal de VSCODE, escribiendo el siguiente comando:

>git init

Y si vamos al explorador de archivos se verá una carpeta oculta “git”, indicando que ya se creó el repositorio inicial.

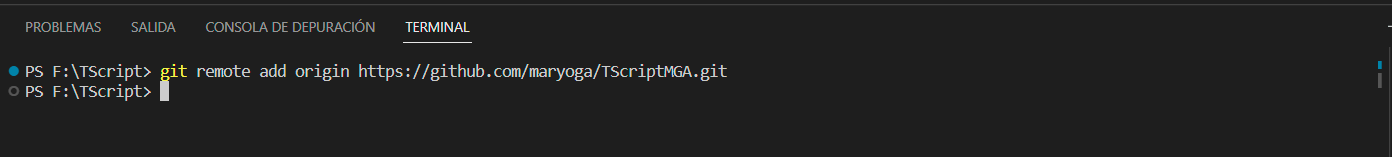
Para conectar el respositorio local de “git” con “github” primero creamos el repositorio en “github”, copiamos la ruta del nuevo repositorio creado, y la añadimos en la terminal de VSCODE “git remote add origin <https://github.com/maryoga/TScriptMGA.git>” y damos enter.

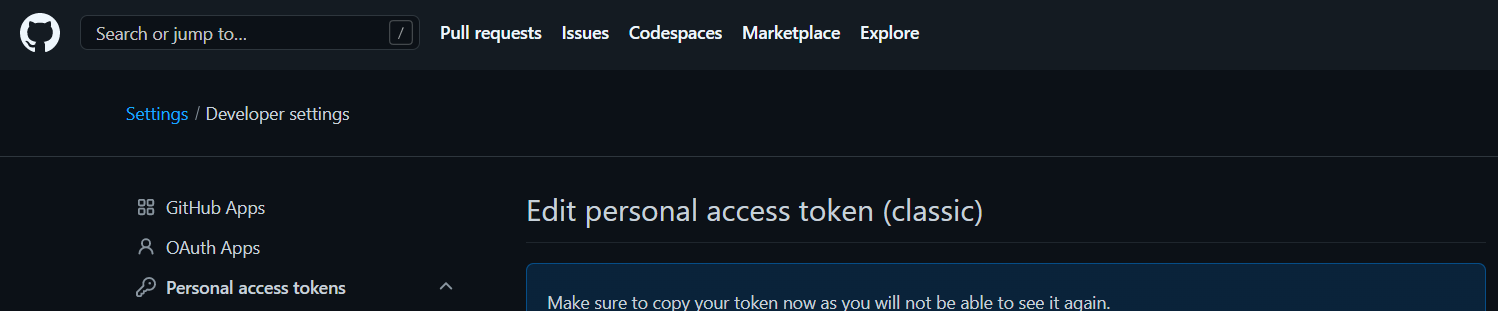


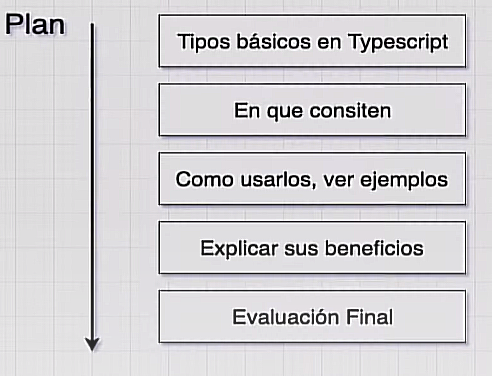
Vemos el nuevo repositorio, de acá copiamos la opción seleccionada en “…or push and existing repository from command line” que es:

“git remote add origin <https://github.com/maryoga/TScriptMGA.git>”

Y lo ponemos en la terminal de VSCODE

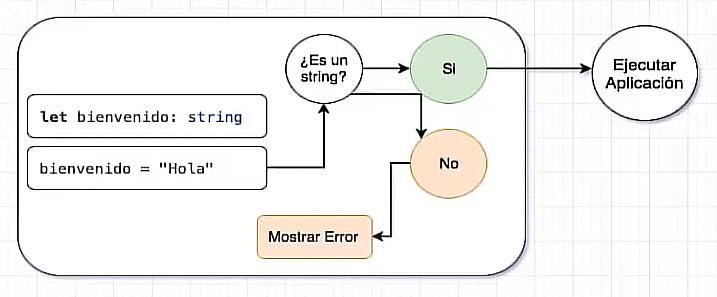


Para conectar el repositorio remoto de github, lo he conectado con el uso de un token que vence el 03-03-2024, estar pendiente de esto en “github” y en VSCODE.

Tipos de datos de TypeScript

A diferencia de JS que es un lenguaje de “tipado dinámico”, TypeScript es exactamente lo opuesto, es de “tipado estático”.

Lo cual quiere decir que las variables que se escriben, se van a chequear en tipo de compilación.



Por ejemplo, si tengo una variable “bienvenido”, y le defino el tipo de datos “string”, lo que va a suceder dentro de la aplicación es que antes que la ejecute, el compilador irá verificando que si esta variable es una “string” entonces va a ejecutar la aplicación. Pero si esta variable no es un “string” nosotros mostraremos un mensaje de error. Por ejemplo, si luego de definirla como “string” voy y cambio en la asignación un tipo númerico.

bienvenido = 1

En ese caso lo que va a suceder es que en la verificación se verá que no es un “string”, entonces mostrará “error” en TS, cosa que debemos solucionar inmediatamente. A diferencia de JS que solo nos damos cuenta cuando ejecutamos la aplicación, de los errores. En TS nos vamos dando cuenta en tipo de desarrollo de los errores.

**Cuáles son los Tipos de Datos Básicos en TS**

En TS al igual que en JS solo existe un tipo de datos para valores numéricos

Numbers: -1, 2.3, 100

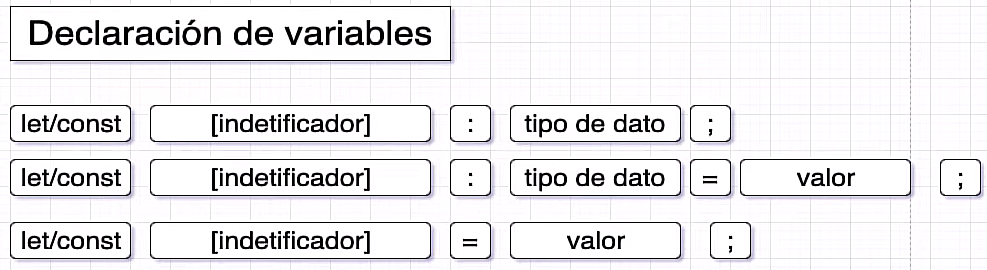
Strings: ‘Hola mundo’, ` `, “Hola”

Booleans: True, False

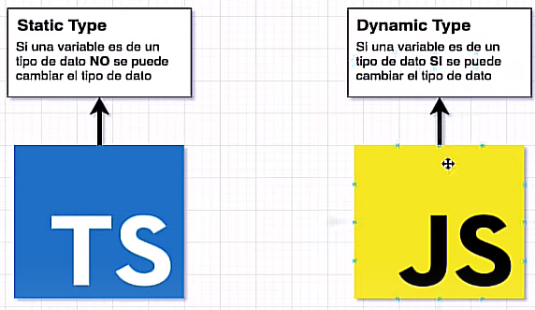
¿Porqué debemos definir tipos a variables?

Porqué no dejarlo abierto a que se pueda asignar cualquier tipo de valor

Hay un problema que sucede en JS que no sucede en TS, si dejamos libre la asignación de tipos a las variables y no definimos el tipo de datos que va a existir, para cada variable, en JS no tenemos restricción al cambiar el tipo de datos.

Esto se resuelve a través del tipado. En TS existen diferentes formas de tipar, de colocarle un tipo de dato a una variable.

En que consiste, 1ero definir de que tipo será esa variable, si crearemos una variable / constante, luego el “identificador”, luego dos puntos “:” y el tipo de datos correspondiente; numérico, string o booleano.

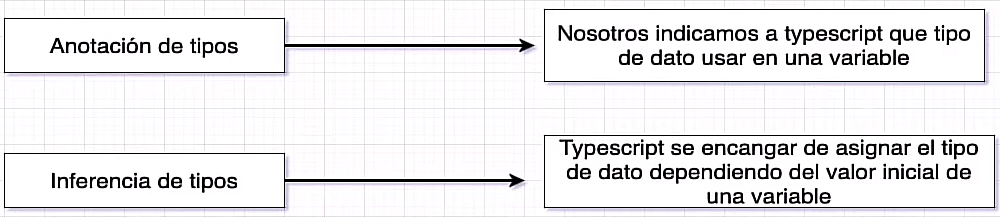
Ejemplo:

let valor1: number;

let valor2: number;

En TS una vez definimos un tipo de dato, no podemos cambiarlo. EN JS si se puede cambiar el tipo de datos, luego de haber definido una variable.

**Anotaciones inferencias de Tipos – Cómo poder declarar variables**

Hay 2 tipos de formas de poder declarar variables en TS.

1. Anotación de tipos
2. Inferencia de tipos

Cuando usar tipos por Anotación y cuando por Inferencia.

Se hace según los requerimientos. Si, por ejemplo, definimos una variable que tendrá un valor por defecto.

En el caso de la siguiente variable, si ya sabemos que tendrá el valor que le vamos a asignar un valor tipo string, y decírselo de manera explícita sería redundante let vehiculo: string = “automóvil” lo adecuado es; let vehiculo = “automóvil”

Pero, si dejo la variable si asignarle un tipo, ni un valor, esta podría ser de cualquier tipo. Y más adelante le asignamos un valor. TS no genera error porque al definirla no le definí ningún tipo o valor. Entonces, la idea básica es que si no tenemos un valor por defecto, lo adecuado es asignarle un tipo a la variable.

//Anotación de tipos

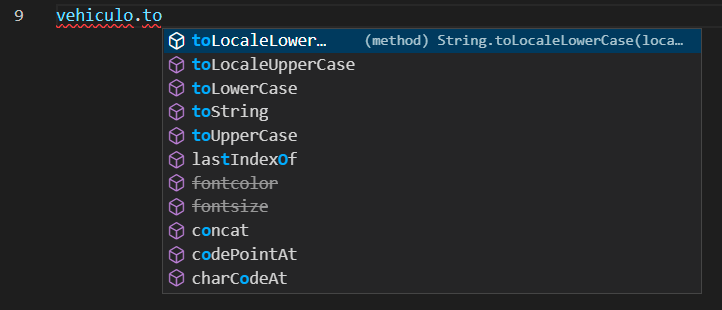
let vehiculo: String;

let cantidad: number;

let esAutomovil: boolean;

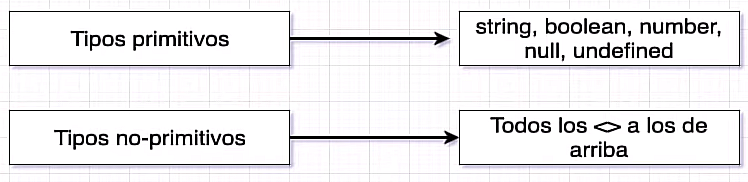
//Inferencia de tipos

let marcaVehiculo = "Honda";

¿Porque es importante usar los tipos? Es para que se generen errores en tiempo de desarrollo y si los hay corregirlos.

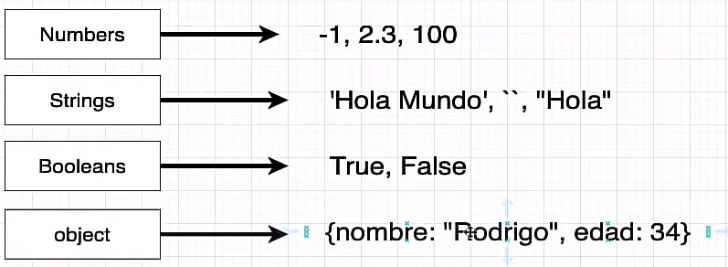
Una vez definimos un tipo de dato, tenemos disponible una serie de métodos según el tipo de dato. El ejemplo de la imagen es de un dato tipo string. Igual aparecen disponible métodos para tipo number.

**Tipo de dato Object – TS.**



En TS existen los tipos de datos; Primitivos y los no-primitivos.

Los objetos, se catalogan como tipo de datos “no-primitivos”

Un objeto se define a través de llaves “{ }” y luego componemos el objeto en TS.

//definiendo objeto TS – Tipo de dato por Inferencia

let persona = {

    nombre: 'Rodrigo',

    edad: 34

};

console.log(persona);

Compilar y ejecutar programa desde la terminal de VSCODE

>**PS F:\TScript\first\_app> ts-node types.ts**

Resultado en la terminal:

{ nombre: 'Rodrigo', edad: 34 }

Luego de ejecutar el comando ts-node, se muestra el contenido que tiene el Objeto “persona”

También podemos definir el tipo de datos que será el objeto persona.

//definiendo objeto TS Tipo de dato por anotación de tipo

let persona: **object** = {

    nombre: 'Rodrigo',

    edad: 34

};

console.log(persona.persona); //las propiedades no se visualizan al definir persona: Objetc

Pero no es la forma más adecuada de declararla, porque así, en el console.log(persona) no pueden visualizarse las propiedades del objeto en ese momento. Para corregir eso hay que especificar cuales son las propiedades que existen dentro de este objeto. Para hacerlo colocamos doble llave “{ }”

//definiendo objeto TS – Definiendo las propiedades que contiene el objeto entre llaves { }

let persona: {

    nombre: string,

    edad: number

} = {

    nombre: 'Rodrigo',

    edad: 34

};

console.log(persona.nombre);

En TS el tipo de datos Object, también podemos tener Objetos anidados al igual que en JS

//definiendo objeto TS - con un Objeto anidado

let persona: {

    nombre: string,

    edad: number,

    direccion: {

        calle: string,

        comuna: string

    }

} = {

    nombre: 'Rodrigo',

    edad: 34,

    direccion: {

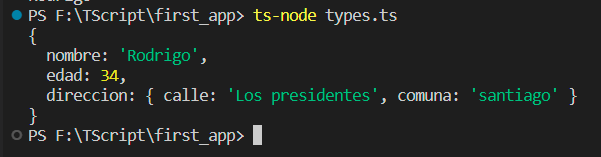
        calle: "Los presidentes",

        comuna: "santiago"

    }

};

console.log(persona);

La salida en terminal es la siguiente:

Otro detalle a mencionar, es que si usamos el comando para solo compilar:

>tsc types.ts

Este genera un archivo types.js (de JavaScript compilado), pero en este archivo no se muestra la definición del objeto de la misma manera que lo hemos hecho en TS, donde agregamos las propiedades del objeto entre llaves dobles { }.

Lo genera del siguiente modo:

Archivo compilado con el comando “tsc types.ts” desde terminal: **types.js**

//definiendo objeto TS - con un Objeto anidado

let persona = {

    nombre: 'Rodrigo',

    edad: 34,

    direccion: {

        calle: "Los presidentes",

        comuna: "santiago"

    }

};

console.log(persona);

Esto pasa porque en JS no tenemos tipados de datos. En JS solo asignamos esos valores a las variables.

Pero estando en TS, no es necesario que definamos las propiedades del objeto entre llaves dobles. Porque ya sabemos los valores por defecto. Así que podemos definir el objeto tal cual se ve en archivo compilado de JS, visto en el fragmento de código anterior.

//definiendo objeto TS - con un Objeto anidado

let persona = {

    nombre: 'Rodrigo',

    edad: 34,

    direccion: {

        calle: "Los presidentes",

        comuna: "santiago"

    }

}

console.log(persona);

**Tipos de datos Array**. [1, 2, 3]

//definiendo un Array dentro del Objeto Persona

let persona = {

    nombre: 'Rodrigo',

    edad: 34,

    direccion: {

        calle: "Los presidentes",

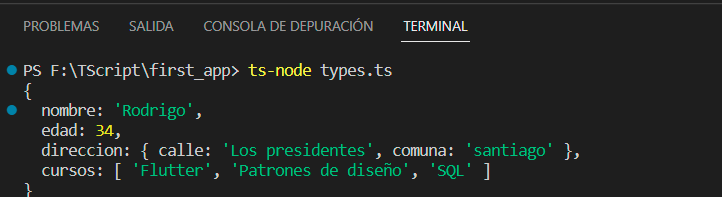
        comuna: "santiago"

    },

**cursos**: [**'Flutter', 'Patrones de diseño', 'SQL'**]

};

console.log(persona);

Resultado en terminal:

Se observa en la salida, el arreglo “cursos” definido dentro del objeto “persona” con sus elementos.

Un arreglo se puede definir de forma independiente.

let hobbies: string[];

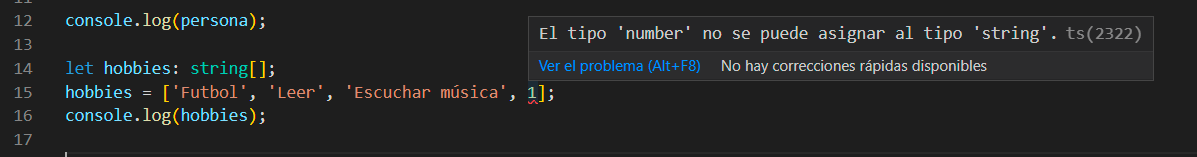
hobbies = ['Futbol', 'Leer', 'Escuchar música'];

console.log(hobbies);

Salida en la terminal:

[ 'Futbol', 'Leer', 'Escuchar música' ]

Cómo indicamos (por anotación de tipos) que nuestro arreglo es de tipo “string”, no podemos colocar valores que no sean “string” dentro de este arreglo.

Esto se puede solucionar (en caso de ser necesario); es colocando un tipo de dato “any” al array. Y con eso ya no aparece el error al agregar un tipo “number” como elemento del Array “hobbies”

let hobbies: **any**[];

hobbies = ['Futbol', 'Leer', 'Escuchar música', 1];

console.log(hobbies);

El tipo de dato “any” es un tipo de dato en TS que permite ingresar cualquier valor; sea un string, number, boolean, cualquier tipo de datos que queramos ingresar en nuestro arreglo. No tendremos problemas.

[ 'Futbol', 'Leer', 'Escuchar música', 1 ]

Puedo hacer la Definición y declaración del Array en una sola línea (con anotación de tipos)

let hobbies: string[] = ['Futbol', 'Leer', 'Escuchar música'];

console.log(hobbies);

Salida en terminal:

[ 'Futbol', 'Leer', 'Escuchar música' ]

La declaración puede ser por inferencia también, y TS deduce que el Array será del tipo de dato “string”.

let hobbies = ['Futbol', 'Leer', 'Escuchar música'];

Recorrer un Array a través de un “For”.

let hobbies: string[] = ['Futbol', 'Leer', 'Escuchar música'];

for(const hobbie of hobbies){

    console.log(hobbie);

}

Salida en pantalla:

Futbol

Leer

Escuchar música

Una vez que tenemos acceso dentro del “for” a la propiedad, podemos usar esta propiedad al igual que utilizamos un tipo de datos string. Lo que nos permite tener acceso a las distintas funciones que tiene el tipo de dato string.

Podría por ejemplo utilizar la función “toLocaleUpperCase()”, que hace que todos los valores del array se impriman en mayúsculas.

for(const hobbie of hobbies){

    console.log(hobbie.**toLocaleUpperCase**());

}

Salida en terminal:

FUTBOL

LEER

ESCUCHAR MÚSICA

Un Array tiene distintas propiedades. Podemos por ejemplo usar un método para ordenar el arreglo.

let hobbies: string[] = ['Futbol', 'Leer', 'Escuchar música'];

//ordenando el array por orden alfabetico

hobbies.sort();

for(const hobbie of hobbies){

    console.log(hobbie.toLocaleUpperCase());

}

Salida en pantalla:

**E**SCUCHAR MÚSICA

**F**UTBOL

**L**EER

**Tuples en TS (Tuplas)**

[Tupla en TypeScript - Línea de Código (lineadecodigo.com)](https://lineadecodigo.com/typescript/tupla-en-typescript/)

Una tupla en TypeScript es un array de elementos que están tipados. De esta manera cada vez que haya que insertar un elemento se validará que dicho elemento coincida con el tipo de dato establecido en la tupla.

Para definir una tupla en TypeScript utilizaremos la siguiente estructura:

let variable:[tipo1,tipo2,...,tipoN];

Por ejemplo, podríamos definir una tupla en TypeScript dónde los elementos fuesen una cadena, un número y otra cadena.

let tupla: [string, number, string];

Definir contenido de la tupla

Cuando vayamos a definir el contenido de la tupla deberemos de definir el contenido para todos los valores. Por ejemplo, para la tupla definida haríamos lo siguiente:

tupla = ['España', 10, 'Madrid'];

Es importante saber que en la tupla hay que dar valor a todos los elementos, ya que si intentásemos solo asignar valores de forma parcial, daría un error.

tupla = ['España',10]; //Error

Acceso al contenido de la tupla en Typescript

Para acceder al contenido de una tupla en TypeScript lo haremos de la misma forma que lo hacemos con un array mediante el operador []

console.log(tupla[0]);

tupla[0] = 'Francia';

Si intentamos insertar un contenido que no coincida con la definición de la tupla, dará error de codificación.

tupla[1] = 'Alemania'; //Error

En el caso de que accedamos o modifiquemos el contenido de un elemento que no esté definido no habrá ningún tipo de control.

Ejemplo de Tuplas:

let automovil = ['nazda', 2020, 'rojo'];

console.log(automovil);

Ejecución y salida en terminal:

PS F:\TScript\first\_app> ts-node .\tuplas.ts

[ 'nazda', 2020, 'rojo' ]

Podría ir a la 2da posición en la tupla, y reasignar un nuevo valor en esa posición.

Cómo declarar las tuplas en TS.

let automovil: [string, number, string] = ['nazda', 2020, 'rojo'];

Si declaro 2 tuplas, y les pongo las mismas restricciones

let automovil: [string, number, string] = ['nazda', 2020, 'rojo'];

let moticicleta: [string, number, string] = ['nazda', 2020, 'rojo'];

Para poder evitar eso, declaramos esta restricción a parte. Para poder hacerlo la forma es a través de “type”. Y declaramos la restricción una sola vez “**vehiculoType**”. Esto queda de la misma forma que declaramos un tipo de dato en TS, vendría siendo como un tipo de dato personalizado que definimos en nuestro proyecto, y esto lo podemos utilizar a lo largo de todo nuestro proyecto. De ese modo nos evitamos tener que ir colocando todos esos valores en cada una de las variables que definimos.

type **vehiculoType** = [string, number, string]

let automovil: **vehiculoType** = ['nazda', 2020, 'rojo'];

let moticicleta: **vehiculoType** = ['yamaha', 2023, 'negro'];

console.log(automovil);

console.log(moticicleta);

Salida en terminal:

[ 'nazda', 2020, 'rojo' ]

[ 'yamaha', 2023, 'negro' ]

Las tuplas son útiles, pero va a depender del requerimiento que se esté analizando.

Podriamos haber declarado el “vehiculo” como un objeto que es con llaves { } y vamos declarando a través de un key las propiedades y asignando un valor. Es mucho más fácil revisar y reutilizar en forma de objeto.

//de este modo nos ahorramos tener que ir definiendo el tipo de datos para un arreglo

let vehiculo = {

    marca: '',

    year: '',

    color: ''

};

Las tuplas también pueden ser útiles en otras situaciones, por ejemplo, cuando necesitamos representar un archivo “CSV” y restringir los valores de cada una de las columnas.

**Enums en TS { Izquierda, Derecha }**

[Enum en Typescript (codigoencasa.com)](https://codigoencasa.com/typescript/#:~:text=%C2%BFQu%C3%A9%20es%20un%20enum%20en%20Typescript%3F%20Los%20enums,la%20creaci%C3%B3n%20de%20un%20conjunto%20de%20casos%20distintos.)

¿Qué es un enum en Typescript? Los enums de TypeScript permiten **definir un conjunto de constantes con nombre**. Su uso puede facilitar la documentación de la intención o la creación de un conjunto de casos distintos. Hoy, exploraremos los fundamentos de los enums de TypeScript junto con los casos de uso, varios tipos de enum, y los próximos pasos para su aprendizaje.

Los enums se utilizan en la mayoría de los lenguajes de programación orientados a objetos como Java y C# y ahora también están disponibles en TypeScript. Son una de las pocas características de TypeScript que no es una extensión a nivel de tipo de JavaScript.

**¿Qué es un enum en Typescript?**

Los enums de TypeScript permiten definir un conjunto de constantes con nombre. Su uso puede facilitar la documentación de la intención o la creación de un conjunto de casos distintos.

Muchos lenguajes de programación, como C, C# y Java, tienen un tipo de datos **enum**, pero JavaScript no. Sin embargo, TypeScript sí lo tiene. TypeScript tiene enums tanto **numéricos** como basados en **cadenas**.

La sintaxis de los enums es la siguiente:

**enum** States {

Oregon,

Washington,

Idaho,

Montana,

Wyoming

}

// usage

**var** region = States.Washington;

¿Por qué usar enums en TypeScript?

Los enums son una gran manera de organizar tu código en TypeScript. Veamos algunas ventajas:

* Proporciona flexibilidad haciendo más fácil expresar y documentar intenciones y casos de uso
* Ahorra tiempo de compilación y de ejecución con el código **inline** en JavaScript
* Permite la creación de constantes personalizadas de memoria eficiente en JavaScript.

**Enums numéricas**

Las enumeraciones numéricas almacenan los valores de las cadenas como números. Pueden definirse utilizando la palabra clave **enum**. Digamos que quieres almacenar un conjunto de diferentes *tipos de coches*. El siguiente ejemplo muestra un **enum** numérico en TypeScript:

enum CarType {

Honda,

Toyota,

Subaru,

Hyundai

}

El valor **enum** **CarType** tiene cuatro valores: Honda, Toyota, Subaru y Hyundai. Los valores del **enum** empiezan por cero y se incrementan en uno para cada miembro, lo que tendría un aspecto similar al siguiente:

Honda = 0

Toyota = 1

Subaru = 2

Hyundai = 3

Si quieres, puedes inicializar tú mismo el primer valor numérico de esta manera:

enum CarType {

Honda = 1,

Toyota,

Subaru,

Hyundai

}

En el ejemplo anterior, hemos inicializado el primer miembro Honda con el valor numérico de uno. Los números restantes se incrementarán en uno.

**Nota**: No es necesario asignar valores secuenciales a los miembros de su enum. Puedes asignarles los valores que quieras.

**Enumeraciones de cadenas**

Los **enums** de cadena son similares a los **enums** numéricos, pero sus valores **enum** se inicializan con valores de cadena en lugar de valores numéricos. Las enumeraciones de cadena *son más legibles que las numéricas*, lo que facilita la depuración de los programas.

El siguiente ejemplo utiliza la misma información que el ejemplo de **enum** numérico, pero se representa como un enum de cadena:

**enum** CarType {

Honda = **"HONDA"**,

Toyota = **"TOYOTA"**,

Subaru = **"SUBARU"**,

Hyundai = **"HYUNDAI"**

}

// Access String Enum

CarType.Toyota; //returns TOYOTA

CarType[**'Honda'**]; //returns HONDA

En el ejemplo, definimos el enum de cadena **CarType** con los mismos valores que el enum numérico, excepto que los valores del enum se inicializan como *literales de cadena*.

**Nota**: Los valores del **enum** de las cadenas deben ser inicializados individualmente.

**Enums heterogéneas**

Los **enums** heterogéneos contienen tanto valores numéricos como de cadena. He aquí un ejemplo:

enum BooleanHeterogeneousEnum {

Yes = **0**,

No = **"NO"**,

}

**Mapeo inverso con enums**

Ya sabes que los valores numéricos se pueden recuperar utilizando sus correspondientes valores de miembro de enum. Con el mapeo inverso, puedes acceder al valor de un miembro y a un *nombre de miembro* *a partir de su valor*. Veamos un ejemplo:

**enum** CarType {

Honda = 1,

Toyota,

Subaru,

Hyundai

}

CarType.Subaru; // returns 3

CarType["**Subaru**"]; // returns 3

CarType[**3**]; // returns Subaru

**CarType[3]** devuelve su nombre de miembro "**Subaru**" debido al mapeo inverso.

Veamos otro ejemplo:

enum CarType {

Honda = 1,

Toyota,

Subaru,

Hyundai

}

console.log(CarType)

Verás la siguiente salida en la consola de tu navegador:

{

**'1': 'Honda'**,

**'2': 'Toyota'**,

**'3': 'Subaru'**,

**'4': 'Hyundai'**,

Honda: 1,

Toyota: 2,

Subaru: 3,

Hyundai: 4

}

Cada valor del enum aparece dos veces en el objeto enum almacenado internamente.

**Enums const**

Puedes utilizar const enums para mejorar el rendimiento de tus enums numéricos. Se definen utilizando el parámetro

**const enum** Enum {

**X** = 1

**Y** = **X** \* 2,

}

A diferencia de los enums regulares, los enums const se eliminan completamente durante la compilación. Sólo pueden utilizar expresiones constantes de enum y se inlinean en los sitios de uso.

**Enums computed**

El valor de un miembro de un **enum** puede ser un valor constante o un **valor computado**. El siguiente ejemplo incluye valores computados:

//Enum computed - Tipo de dato Enum calculado

enum CarType {

    Honda = 1,

    Toyota = getCarTypeCode('toyota'),

    Subaru = Toyota \* 3,

    Hyundai = 10

}

function getCarTypeCode(carName: string): number {

    if(carName === 'toyota'){

        return 5;

    }

    return 0

}

console.log(CarType);

El valor en consola sería el siguiente, al ejecutar en la línea de comando de la terminal de VSCODE, se ve el resultado de los valores calculados para **Toyota** y para **Subaru**:

PS F:\TScript\first\_app> ts-node .\tipoEnumComputed.ts

{

**'1'**: **'Honda'**,

**'5'**: **'Toyota'**,

**'10'**: **'Hyundai'**,

**'15'**: **'Subaru'**,

Honda: **1**,

Toyota: **5**,

Subaru: **15**,

Hyundai: **10**

}

console.log(CarType.Toyota); // returns 5

console.log(CarType.Subaru); // returns 15

5

15

Si el enum incluye miembros computados y constantes, los miembros del enum no inicializados van primero o después de otros miembros inicializados con constantes numéricas. El siguiente ejemplo mostrará un error:

enum CarType {

    Toyota = getCarTypeCode('toyota'),

    Honda, //Error: Enum member must have initializer

    Subaru = Toyota \* 3,

    Hyundai = 10

}

Puedes declarar el **enum** anterior así:

enum CarType {

Honda,

Hyundai,

Toyota = getCarTypeCode('toyota'),

Subaru = Toyota \* 3

}

**Resumen**

* La forma corta de enumeración es enum.
* Se utiliza como un tipo con algún conjunto de valores llamado constante con nombre.
* Internamente contiene valores de tipo numérico o en algunos casos podemos crear enums heterogéneos.
* La palabra clave enum se utiliza para definir los enums.
* Podemos crear un caso distinto por lo que facilita la documentación.
* Nos obliga a utilizar todos los valores posibles.
* Se puede acceder a los enums usando el índice o usando el operador punto después del nombre del enum y la constante nombrada.
* El valor inicial parte de 0 si no establecemos ningún valor.
* Auto incremental es el mejor enfoque a utilizar porque no necesitamos preocuparnos por los valores.

Usualmente son tipos de datos en donde enumeramos valores que son fijos dentro de un Proyecto.

Por ejemplo, si definimos una transacción bancaria, podríamos decidir enumerar sus “estados”; si fue “exitoso”, si se “canceló” la transacción bancaria, si hubo un “error”, si no tiene fondos “sinFondos”, estos estados los podríamos poner dentro de un tipo de dato enumerado.

//definiendo un Enum llamado “role” dentro del Objeto Persona

enum role{

    estudiante,

    profesor

}

let persona = {

    nombre: 'Rodrigo',

    edad: 34,

    direccion: {

        calle: "Los presidentes",

        comuna: "santiago"

    },

    cursos: ['Flutter', 'Patrones de diseño', 'SQL'],

    role: role.profesor

};

console.log(persona);

Resultado en terminal:

{

nombre: 'Rodrigo',

edad: 34,

direccion: { calle: 'Los presidentes', comuna: 'santiago' },

cursos: [ 'Flutter', 'Patrones de diseño', 'SQL' ],

role: **1**

}

El valor de “rol” es “1” porque es el 2do elemento de la enumeración que inicia con “0”, si hubiese puesto role.estudiante, en consola obtendría el valor “0”.

Esto se encontrará mucho en proyectos que tienen código Servidor, por ejemplo, ya sea que hubiese escrito en C#, Java, en varios lenguajes en proyectos grandes se encontrarán enumeradores porque de esta manera es mucho más fácil ir utilizando valores fijos para poder reutilizarlos después a lo largo de un proyecto.

**El tipo de dato “Any” en TS**

Las variables **any** en **TS** nos permiten ser un poco laxos en el control del tipado de las variables de un programa.

Si has trabajado con **TypeScript** sabrás que una de sus mayores ventajas es el *control del tipo de datos de las variables*. Si bien, si has trabajado con **JavaScript** sabrás que *una variable puede cambiar de tipo durante la ejecución de un programa* y que esto puede ser una gran ventaja en la codificación.

Es por ello, y de cara a solucionar esta situación, que **TypeScript** define las variables de tipo **any**. Estas serán unas variables cuyo tipo de dato podrá ser ‘cualquiera’, es decir, podrá cambiar de tipo de dato a lo largo de la codificación del programa.

Ejemplo:

let x:any = 3;

console.log(x);

En este caso la variable se comportará como una variable de tipo **number**.

Pero es que unas líneas de código más adelante podemos cambiar su contenido y, por ejemplo, asignar una cadena a dicha variable.

x = 'soy una cadena de texto';

console.log(x);

Vemos que ahora, al volcarlo por consola, lo que se vuelva es una cadena. Pero más aún, podemos volver a cambiar el tipo de la variable asignándola un nuevo valor.

x = true;

console.log(x);

Como podemos apreciar las variables any en TypeScript nos pueden permitir ese punto de flexibilidad sobre el control de tipo de datos, pero sin perder el resto de funcionalidades que nos da TypeScript.

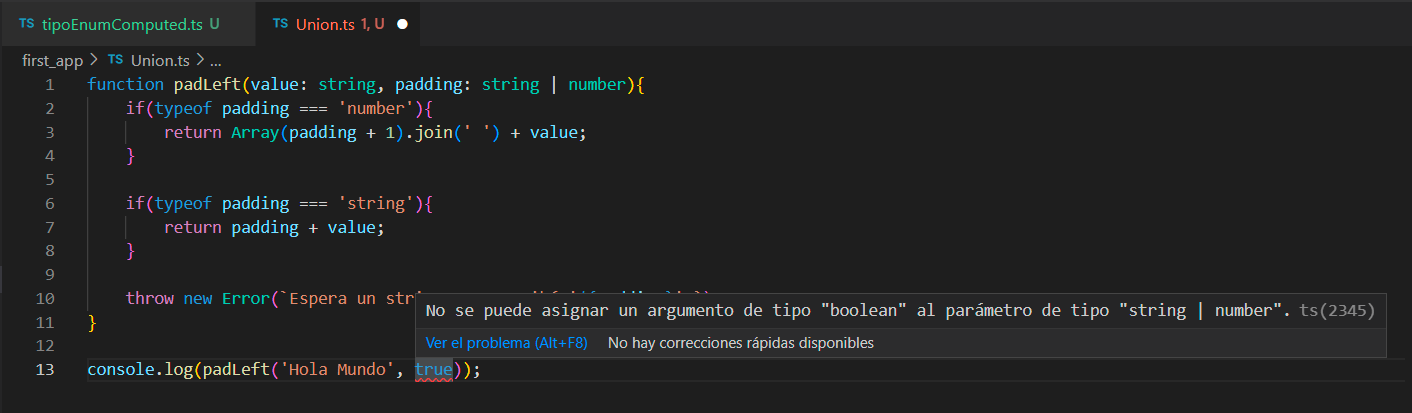
Este tipo de dato no es muy recomendable, ya que si solo definimos un arreglo con el tipo de dato Any, se perderá el propósito por el cual TS fue creado, dónde TS pueda verificar el tipo de dato en tiempo de compilación de una variable. Si quisiéramos hacer eso, lo podríamos hacer con las Tuplas, donde nosotros en cada una de estas filas, donde colocábamos el tipo de dato que tiene esa fila. Por ejemplo, para la 2da fila, podríamos indicar que el tipo de dato es numérico y de esta manera indicamos a TS que solo en la 2da fila deseo que se ingresen datos numéricos, no quiero que se ingresen tipos de datos “string”.

let hobbies: any[] = ['Futbol', 1, 'Leer'];

**El tipo de dato “Union” en TS – string | number | boolean**

Este tipo de dato, nos permite ir colocando distintos tipos de datos para una variable y los vamos separando por el símbolo – or – “|”

Tengo una función que me permite colocar un “string” en la parte izquierda de mi pantalla, o si pudiese colocar en la parte izquierda un cierto “padding” para que no se coloque ningún pin en la parte izquierda, si no que se coloque con algunas posiciones a la derecha. Por ejemplo, si coloco en el 2do parámetro un número 20, entonces habrá 20 posiciones en blanco y luego el valor del “string”.

Si la función la dejo libremente, y pongo en el paso del parámetro uno tipo “boolean” me dará error. Para mejorar la funcionalidad de la función es que está el pipe “|” (union), que sirve para definir una función que puede recibir sea un parámetro de un tipo o de otro. En nuestro ejemplo, o puede ser de tipo “string” o “number”. Si quiero pasar un parámetro de otro tipo mandará error.

Si quisiéramos también reutilizar este tipo de dato (en el parámetro de la función) con pipe “|” (padding: string | number) en nuestro proyecto, podríamos declararlo con un “**type**”, luego podría decirle que este va a ser un string o un number, y le colocamos que valores tendrá este tipo de dato.

type StringOrNumber = string | number;

De este modo puedo reutilizar este tipo de dato en donde lo requiera. Por ejemplo como parámetro de la función “padLeft”.

type StringOrNumber = string | number;

function padLeft(value: string, padding: StringOrNumber){

    if(typeof padding === 'number'){

        return Array(padding + 1).join(' ') + value;

    }

    if(typeof padding === 'string'){

        return padding + value;

    }

    throw new Error(`Espera un string pero recibí '${padding}'.`);

}

console.log(padLeft('Hola Mundo', 20));

**El tipo de dato “Literal” en TS - Literal Types**

Como vimos ya TypeScript soporta la **unión de tipos** para que una variable pueda *tener varios tipos de datos diferentes*.

Los Tipos Literales son básicamente lo mismo, pero lo limitamos por lo general, a valores concretos, por ejemplo:

let operaciones:string = "suma";

*operaciones* soporta cualquier *string*, es este caso la cadena "*suma*"

En el último ejemplo *operaciones* puede contener "*resta*", "*multiplicacion*" o "*division*", como también "*manzana*", "*payaso*" o "*tomate*", todos son cadenas válidas, pero no todas son cadenas apropiadas para nuestra variable. Por ello podemos usar ***Tipos Literales*** para limitar aún más los valores que puede admitir la variable *operaciones*.

*Veamos otro ejemplo*:

Tenemos una función *concatOrSumValues* que concatena los valores si son **string** y los suma si son **number**:

type numOrString = number | string;

function concatOrSumValues(value1:numOrString, value2:numOrString, placeholder:string = 'to-sum'):numOrString {

    let res;

    if(placeholder === 'to-sum'){

      res = +value1 + +value2

    }else{

      res = value1.toString() +  value2.toString();

    }

    return res;

  }

  console.log(concatOrSumValues(2,2)); //4

  console.log(concatOrSumValues("2","2", 'to-concat')); // "22"

Al ejecutar el comando para compilar y ejecutar, obtendremos la salida en terminal:

PS F:\TScript\first\_app> **ts-node** .\literalTypes

4

22

La función depende del parámetro *placeholder*, que por defecto es *to-sum* ergo la función sumará los valores por defecto.

Pero *placeholder* puede recibir cualquier cadena, ¿*cómo podemos limitarlo* a que solo pueda recibir *to-sum* para la suma, y *to-concat* para la concatenación? Acá entran los Tipos Literales.

Podríamos hacer algo como esto:

type StringOrNumber = string | number; //unión de tipos

type PlaceholderOperation = 'to-sum' | 'to-concat' // unión de tipos literales

function concatOrSumValues(value1:StringOrNumber, value2:StringOrNumber, placeholder:PlaceholderOperation = 'to-sum'):StringOrNumber {

  let res;

  if(placeholder === 'to-sum'){

    res = +value1 + +value2

  }else{

    res = value1.toString() +  value2.toString();

  }

  return res;

}

console.log(concatOrSumValues(2,2)); // 4

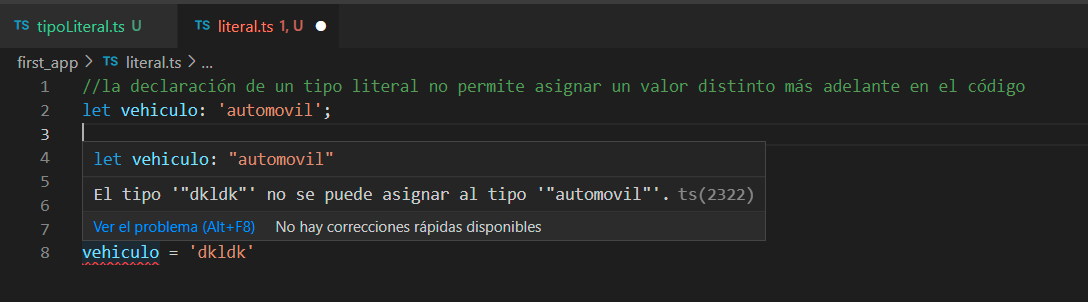
console.log(concatOrSumValues("2","2", 'to-concat')); // '22'

console.log(concatOrSumValues("3","3", **'to-conca'**)); // Argument of type '"to-conca"' is not assignable to parameter of type 'PlaceholderOperation'

*PlaceholderOperation* solo puede recibir 2 cadenas específicas: *to-sum* o *to-concat*, cualquier otro string será un error. Como se observa en la última línea de llamada a la función, llamamos con una cadena que no existe para la variable tipo: PlaceholderOperation, donde no existe el valor ‘**to-conca**’, si no ‘to-concat’, por ello la llamada da error.

type PlaceholderOperation = 'to-sum' | 'to-concat'

Conclusiones

* Los *tipos literales* reducen más los posibles *valores que puede aceptar una variable*.
* Son muy comunes y muy usados para aprovechar al máximo los beneficios de TypeScript.

**TIPOS LITERALES**

Se le llama tipo literal en TypeScript a aquel *tipo que contiene un único valor de tipo string, number, boolean o enum*. Por ejemplo:

**type** Two = 2;

**type** Duck = '**Duck**';

**const** x: Two = 2; **// OK**

**const** y: Two = 3; **// Error**

**const** a: Duck = '**Duck**'; **// OK**

**const** b: Duck = '**Fox**'; **// Error**

A primera vista no parece algo muy útil porque todas las variables que definamos del tipo tendrán el mismo valor, pero cuando los combinamos con otros tipos la cosa mejora.

Las reglas de inferencia de tipos para los tipos literales son un poco confusas (y han cambiado alguna vez a lo largo de la vida de TypeScript), por lo que hay que andarse con un poco de ojo:

**let** x = 2; **// el tipo inferido para x es "Number"**

**const** x = 2; **// el tipo inferido para x es 2**

Por supuesto, un valor de un tipo literal es asignable a valores del tipo «general», pero no a la inversa:

**const** x: Two = 2;

**const** y: Number = x; **// OK**

**const** z: Two = y; // **Error**

EL VALOR QUE ASIGNAMOS A UN TIPO LITERAL SE TOMA COMO UN TIPO QUE NO PUEDO CAMBIAR NUEVAMENTE

Este tipo de dato (literal) se usa en combinación con el tipo de dato “unión”.

**Funciones en TS**

Existen diferentes formas de definir o declarar una función en Javascript, y siendo Typescript un superset, no es diferente, ahora tienes muchas más formas o sintaxis para cumplir el mismo objetivo.

Primero, para crear una función en Javascript existen 4 formas base:

**Declaración de función**

function multiplicar(a,b) {

    return a \* b

}

**Expresión anónima**

const multiplicar = function(a,b) {

    return a \* b;

}

**Expresion de función con nombre**

const multiplicar = function **multiplicar**(a,b) {

    return a \* b;

}

**Función flecha / Arrow function**

const multiplicar = (a, b) **=>** {

    return a \* b;

}

**Retorno implícito**

const multiplicar = (a, b) => **a \* b**;

El objetivo de agregar Typescript al stack es definir los tipos tanto de los argumentos de la función, como del valor de retorno (aunque este muchas veces puede ser inferido).

**Comencemos por agregar tipos a los métodos mencionados**

**Declaración de función**

// Tipo de Retorno inferido

function multiplicar(a: number,b: number) {

    return a \* b

}

// Tipo de Retorno definido

function multiplicar(a: number,b: number): **number** {

    return a \* b

}

**Expresión anónima**

const multiplicar = **function**(a: number,b: number): number {

    return a \* b;

}

**Expresion de función con nombre**

const multiplicar = function **multiplicar**(a: number,b: number): number {

    return a \* b;

}

**Función flecha / Arrow function**

const multiplicar = (a: number, b: number): number **=>** {

    return a \* b;

}

**Retorno implícito**

const multiplicar = (a:number, b: number): number => **a \* b**;

También es posible extraer la definición de los tipos e incluirlos en una sección diferente que la propia función.

const multiplicar: (a:number, b: number) => number  = (a,b)=> a \* b;

También es posible extraer el tipo de la función para *mejorar legibilidad* o incluso para *utilizarlo en otro lugar*.

**type** **MulFn** = (a:number, b: number) => number

const multiplicar: MulFn  = (a,b)=> a \* b;

Esto también puede ser logrado utilizando la sintaxis de objeto

//Esto también puede ser logrado utilizando la sintaxis de objeto

type MulFnObj = {

    (a:number, b: number): number

}

const multiplicarObj: MulFnObj  = (a,b) => a \* b;

O también utilizando una interfaz

interface MulFnInterfaz {

    (a:number, b: number): number

}

const multiplicarI: MulFnInterfaz  = (a,b)=> a \* b;

**Parámetro opciones y parámetros por defecto**

Una característica interesante y muy útil *al definir los tipos de tus funciones* es la posibilidad de *declarar que parámetros son opcionale*s.

//parámetros opcionales con signo de interrogación

const multiplicarAB: (a:number, b?: number) => number  = (a,b) => a \* (b ?? 1);

Este ejemplo permite que el parámetro `*b`* no se utilice o sea *undefined*

**Importante**! El orden en que defines los parámetros opcionales es importante ya que al definir un parámetro opcional todos los parámetros siguientes deberán ser opcionales también.

*Otra forma de crear opcionalidad de parámetros* es *definir su valor por defecto*. Typescript tratará dicho parámetro como un parámetro opcional

CODIGO A REVISAR \*\*\*

//parámetro por defecto - es tratado como opcional

const multiplicarAB\_Opc: (a:number, b?: number) => number  = (a,b = 1) => a \* b;

multiplicarAB\_Opc(2) // = 2

multiplicarAB\_Opc(2, undefined) // 2

**Async**

Las funciones asíncronas en Typescript funcionan de la misma forma que en Javascript, solo que el tipo de retorno es en efecto una Promise genérica.

async function somePromise(a: number, b: string): Promise<number> {

    // logic

}

const somePromise2 = (a: number, b: string): Promise<number> => {

    // logic

}

Esperas Asincrónicas - Async / Await

Las async no pueden ser utilizados en *es5* deben ser usados en ***es6*** o superior.

En TypeScript cuando código JS se está ejecutando (**sincronamente**) y utiliza una función la cual contiene una o más *promesas* se podrá utilizar la palabra reservada *await* para parar la ejecución del código JS hasta que la función termine correctamente, en caso de fallo, está función generará un error de manera *síncrona* que podremos atrapar mediante un *try catch*.

// ----------- test.ts -----------

// No es código de verdad solamente es la prueba de concepto

async function foo() {

    try {

        var val = await getMeAPromise();

        console.log(val);

    }

    catch(err) {

        console.log('Error: ', err.message);

    }

}

* Si la función termina entonces devolverá un valor
* Si la función falla devolverá un error que podremos capturar

Esto convierte drásticamente la programación asíncrona tan fácil como la programación síncrona. ya que cumple 3 requisitos indispensables:

1. Capacidad de pausar la función en tiempo de ejecución
2. Capacidad de pasarle valores a funciones
3. Capacidad de lanzar excepciones en caso de fallo

El código que genera el ejemplo de arriba debería ya sonar de algo ya que esa es la sintaxis utilizada en TypeScript para crear generators, es decir, el código previamente visto se convertiría en:

var \_\_awaiter = (this && this.\_\_awaiter) || function (thisArg, \_arguments, P, generator) {

    return new (P || (P = Promise))(function (resolve, reject) {

        function fulfilled(value) { try { step(generator.next(value)); } catch (e) { reject(e); } }

        function rejected(value) { try { step(generator.throw(value)); } catch (e) { reject(e); } }

        function step(result) { result.done ? resolve(result.value) : new P(function (resolve) { resolve(result.value); }).then(fulfilled, rejected); }

        step((generator = generator.apply(thisArg, \_arguments)).next());

    });

};

// No es código de verdad solamente es la prueba de concepto

function foo() {

    return \_\_awaiter(this, void 0, void 0, function\* () {

        try {

            var val = yield getMeAPromise();

            console.log(val);

        }

        catch (err) {

            console.log('Error: ', err.message);

        }

    });

}

Obviamente esto no es lo mismo que lo que se explica en la lección anterior sino una versión más compleja.

**Generics**

Generics es en mi opinión una de las herramientas más poderosas de Typescript ya que permite total flexibilidad en la forma en que defines tus tipos, en el caso de declarar funciones lo puedes hacer de la siguiente forma

function someFunction<GenericType>(a: GenericType): Array<GenericType> {

    // some logic

}

En este ejemplo la función toma un parámetro tipo `***GenericType***` y retorna un arreglo del mismo tipo, ¿cómo se usa?

someFunction<number>(10) // [10]

Esto también se puede lograr con `***arrow functions***` pero, existe un problema: Si estás trabajando con **React** (o cualquier framework que acepte algo similar a **JSX**) el compilador de Typescript no sabrá si estás usando JSX o un **Generic**, por lo que necesitas agregar algo que le permita identificar que se trata de un genérico

const someFunction  = <GenericType extends unknown>(a: GenericType): Array<GenericType> => {

    //some logic

}

Los tipos genéricos, son aquellos que como las interfaces *no se verán compilados en Javascript* ya que solo están accesibles en tiempo de compilación, La manera adecuada de realizar la sobrecarga de métodos es con los tipos genéricos un ejemplo sería así:

Versión TypeScript

function echo<T>(arg: T) : T {

    return arg;

}

let a = echo<number>(1); // El typeof es Number

let b = echo<String>("Hola mundo"); // El typeof es String

Versión Javascript (Ya compilado)

function echo(arg) {

    return arg;

}

var a = echo(1); // El typeof es Number

var b = echo("Hola mundo"); // El typeof es String

La diferencia entre esta forma y la otra, es que, de esta forma, podríamos recibir cualquier tipo de objeto, y no deberíamos especificar el tipo de objeto que esperamos, esto está muy bien ya que está diseñado para los objetos que no son primitivos de javascript. Con esto evitamos el Any y mejoraría la manera de realizar la sobrecarga (Lejos de cómo sería en Java o C#).

Con los tipos genéricos se debe tener cuidado, ya que no todos los métodos están disponibles para todos los tipos de objetos.

**TypeScript**

ALGO NO ESTA BIEN EN ESTA FUNCION

class Generic<T> {

    add: (X: T, y:T) => T;

}

let myGeneric = new Generic<number>();

console.log(myGeneric.add = function (x,y) {return x + y});

console.log(myGeneric.add(3,4));

TypeScript Nota: Es muy importante que veaís que cuando se implementa una interfaz en un parámetro utilizamos la palabra reservada extends y no la palabra reservada implements

ALGO NO VA BIEN CON ESTA FUNCION

// Interfaz que asegura que el parametro tenga el metodo length

interface withLength {

    length: number;

}

// El parametro hereda de la interfaz la cual fuerza al parametro tenga el método length

function echo<T extends withLength>(arg: T): T {

    console.log(arg.length);

    return arg;

}

// Esto funcionará

let a = echo("aaa");

let t = echo({length: 2, name: "aa"});

// Esto NO funcionará

//let b = echo(1);

**NEVER en Funciones TS**

**INTERFACES vs CLASES**

TS ofrece la posibilidad de trabajar con interfaces, que son un tipo de construcción utilizada en la **POO**. Las interfaces nos pueden dar algo más de trabajo a la hora de tirar líneas de código, pero *nos ayudan mucho durante todo el tiempo que estamos desarrollando* un programa.

**Qué es una interfaz**

Las interfaces son un mecanismo de la **POO** que trata de suplir la carencia de herencia múltiple. La mayoría de los lenguajes que implementan la orientación a objetos *no ofrecen la posibilidad de definir una clase que extienda varias clases a la vez* y sin embargo a veces es deseable. Ahí es donde entran las interfaces.

Una clase puede extender otra clase, heredando sus propiedades y métodos y declarar que implementa cualquier número de interfaces. La diferencia de las clases que extiendes con respecto a las interfaces es que las interfaces no contienen implementación de sus métodos, por lo que la clase que implementa una interfaz debe escribir el código de todos los métodos que contiene. Por este motivo, se dice que las interfaces son como un contrato, en el que se especifica las cosas que debe contener una clase para que pueda implementar una interfaz o cumplir el contrato declarado por esa interfaz.

En **TS** una interfaz *puede definir propiedades*, mientras que *en otros lenguajes las interfaces sólo definen métodos*.

**Declarar una interfaz en TypeScript**

Las interfaces en **TS** se declaran de manera bastante similar a la de las clases, *indicando la lista de propiedades y métodos que contendrán*. Solo hay un detalle fundamental, que *las propiedades no pueden tener valores y los métodos no pueden tener código para su implementación*.

**interface** sumergibleInterface {

    tiempoMaxBajoElAgua: **number**;

    profundidadMaxima: **number**;

    repelerAgua(): void;

  }

Solo hemos indicado los tipos y los métodos, pero no hemos indicado sus valores. Hemos usado además el tipado de TS

**Interfaces**

* Solo existen en tiempo de compilación en TS
* Solo se usan para la verificación de tipos

**Clases**

* Existen en tiempo de compilación y durante el tiempo de ejecución
* Podemos inicializar propiedades e implementar métodos
* Crear instancias de dicha clase

**Access control keywords - MODIFICADORES DE ACCESO**

* Public
* Private
* Protected
* ReadOnly

Existe un mecanismo en la **POO**, en este caso usando **TS**, que permite hacer visibles o no, **métodos** y **atributos** de una clase. Este mecanismo se llama **modificadores**. Estos modificadores *pueden modificar la visibilidad, tanto de un atributo, cómo de un método*.

Existen **cuatro tipos de modificadores** dependiendo de qué tan visible se requiere que sea un atributo, estos modificadores son: public, private, protected y readonly

**Public**

Todos los **métodos** y **propiedades** de TS son públicos por defecto. Si no agregamos modificador de acceso por defecto es 'public'

**class** TodosSabenMisSecretos {

    secreto1='este atributo es público';

**public** secreto2='este otro método también es público'

}

*/\* el mundo exterior \*/*

**let** clasePublica = new TodosSabenMisSecretos();

console.log(clasePublica.secreto1); *//OKA lectura*

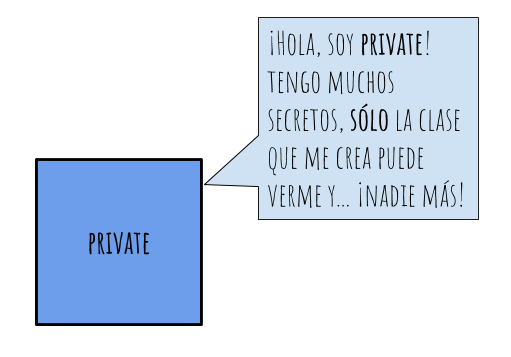
console.log(clasePublica.secreto2); *//OKA lectura*

clasePublica.secreto1 = 'nuevo secreto'; *//OKA escritura*

clasePublica.secreto2 = 'otro nuevo secreto'; *//OKA escritura*

**modificador private**

El modificador **private**, por otro lado, es lo opuesto a **public**. Este modificador restringe la visibilidad de los atributos de la clase *sólo a esta clase*. Dicho de otra manera, *nadie más que la clase tiene acceso a estos atributos*, tanto para *leer* como para *modificarlos*.

**class** SoloYoSeMisSecretos {

**private** secretoSoloLectura='Mi secreto';

}

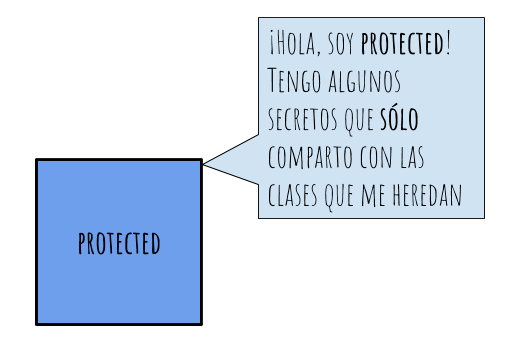
*/\* el mundo exterior \*/*

**let** objetoConSecretos = new SoloYoSeMisSecretos();

console.log(objetoConSecretos.secretoSoloLectura); *//ERROR LECTURA*

objetoConSecretos.secretoSoloLectura = 'otro secreto' *//ERROR ESCRITURA*

**modificador protected**

¿Se puede saber si una clase puede heredar de otra y tener acceso a todos los atributos de ésta? Es posible claro, si sus métodos y atributos no son privados. Pero… ¿qué pasaría si quisieras que una clase “hija” tenga acceso a los atributos de la clase “padre”, pero estos métodos no tengan acceso a el “mundo exterior”?

Para eso existe el modificador **protected**, que es lo que hace exactamente. **Protected** hace visibles atributos entre clases padre e hijas, pero los hace no-visibles al resto del mundo.

**class** ClasePadre {

**protected** atributoProtected:string;

}

**class** ClaseHija **extends** ClasePadre {

**constructor**() {

    super();

       this.atributoProtected='Este es un secreto entre ClasePadre'

    }

}

**let** hija = new ClaseHija();

console.log('cuál es el secreto?', hija.atributoProtected) *//esto nos tira un* ***error****, acá es el mundo exterior*

El código nos arroja un error ya que no podemos acceder al atributo **atributoProtected**, éste sólo es accesible entre las clases padre e hija.

**Modificador readonly**

Disponemos además de los modificadores 'private' y 'public' uno llamado 'readonly'. Mediante este modificador el valor de la propiedad solo puede ser cargado en el constructor o al momento de definirlo y luego no puede ser modificado ni desde un método de la clase o fuera de la clase.

El modificador **readonly**, permite sólo la lectura de un atributo ya que, en caso de querer modificarlo el código arrojaría un error.

**class** SóloYoModificoMisSecretos {

**readonly** secretoSoloLectura='este atributo es privado';

}

*/\* el mundo exterior \*/*

**let** objetoSoloLectura = new SóloYoModificoMisSecretos();

console.log(objetoSoloLectura.secretoSoloLectura); *//OK*

objetoSoloLectura.secretoSoloLectura = 'otro secreto' *//****ERROR***

¿Y qué pasa con los constructores abreviados?

**class** Persona {

**constructor**(nuevoNombre:string, nuevoApellido:string) { }

}

**class** Persona {

**constructor**(

**public** nombre:string,

**readonly** apellido:string,

**private** direccion:string

    ) {    }

}

**let** nuevaPersona = new Persona('Gustavo', 'Dohara', 'Mi dirección');

console.log(nuevaPersona.nombre); *// public*

console.log(nuevaPersona.apellido) *//read only*

console.log(nuevaPersona.direccion) *//ERROR esto es private*

**¿Qué es una Interface?**

Es una definición de contrato de código.

Una interface define la “forma” de la data, es una especie del molde o plantilla, con la que vamos a trabajar.

Para definir una Interface en TS debemos utilizar la palabra reservada “**interface**”

interface Book {

    id: number;

    title: string;

    author: string;

    coAuthor**?**: string;

    isLoan: (id: number) => void

}

La propiedad “coAuthor” tiene un signo de interrogación “?” que indica que esta propiedad es “opcional”.

A nivel de programación tradicional, lo típico que puedes hacer con una interfaz es implementarla en una clase.

Una vez definida tu interfaz, podrás implementarla en todas las clases que desees mediante la palabra "implements" *en la cabecera de la clase*. Para nuestro ejemplo, una vez definida la interfaz **sumergibleInterface**, todas las clases que vayan a tener objetos que se puedan sumergir, tendrán que implementarla.

  class relojSumergible **implements** sumergibleInterface {

    tiempoMaxBajoElAgua = 1;

    profundidadMaxima = 10;

    repelerAgua() {

      console.log('El agua me resbala');

    }

  }

La interfaz provoca que sea *necesario declarar todas las propiedades e implementar todos los métodos* a la hora de definir la clase. En resumen, es *como un contrato*.

Si no se cumple el contrato de la interfaz, entonces nuestro editor se quejará (si está preparado para mostrar los errores de código **TS**), o el compilador nos lo advertirá.

**Modo observador en TypeScript**

<https://www.discoduroderoer.es/modo-observador-en-typescript/>

El modo observador se usa en TS para compilar los cambios automáticamente cada vez que hagamos un cambio en un fichero, es bastante útil para compilar todos los archivos **ts** automáticamente.

Lo único que tenemos que hacer es ejecutar el siguiente comando a la hora de compilar usando el parámetro –watch

$ **tsc** index.ts --watch

O en el caso del ejemplo que estamos realizando de “generics”

PS F:\TScript\first\_app> **tsc** .\genericsFunction.ts **--watch**

La terminal enviara un mensaje luego de ejecutar el comando

Starting compilation in watch mode...

Esto hará que se quede pendiente de los cambios que haremos en el archivo.

**Conclusión**

Existen varias formas de definir funciones, he dejado muchas afuera, pero creo que estas son las formas más generales. Algunas que no han sido mecionadas:

* Type Guards
* Assertions
* Generators
* Function Overloads
* Modules
* Classes

CURSO A SEGUIR EN UDEMY en mi Cuenta

<https://www.udemy.com/course/typescript-in-a-nutshell/learn/lecture/29663590?start=0#overview>

Programa especializado: Desarrollo de aplicaciones móviles con Android

<https://www.coursera.org/specializations/programacion-android>

React Web App Testing With NodeJs, Cypress, and WebDriverIO

<https://www.udemy.com/course/react-web-app-testing-with-nodejs-cypress-and-webdriverio/?ranMID=39197&ranEAID=d2gvurItCFk&ranSiteID=d2gvurItCFk-2xAYRnADd3OxgPmI57WDTQ&utm_source=aff-campaign&LSNPUBID=d2gvurItCFk&utm_medium=udemyads>