Apuntes de Typescript

migueluisV

Abril 2024

Índice

Índice de Figuras

Índice de Tablas

Este documento se hizo con **Overleaf** y los ejemplos fueron desarrollados y probados principalmente en el Playground de Typescript.

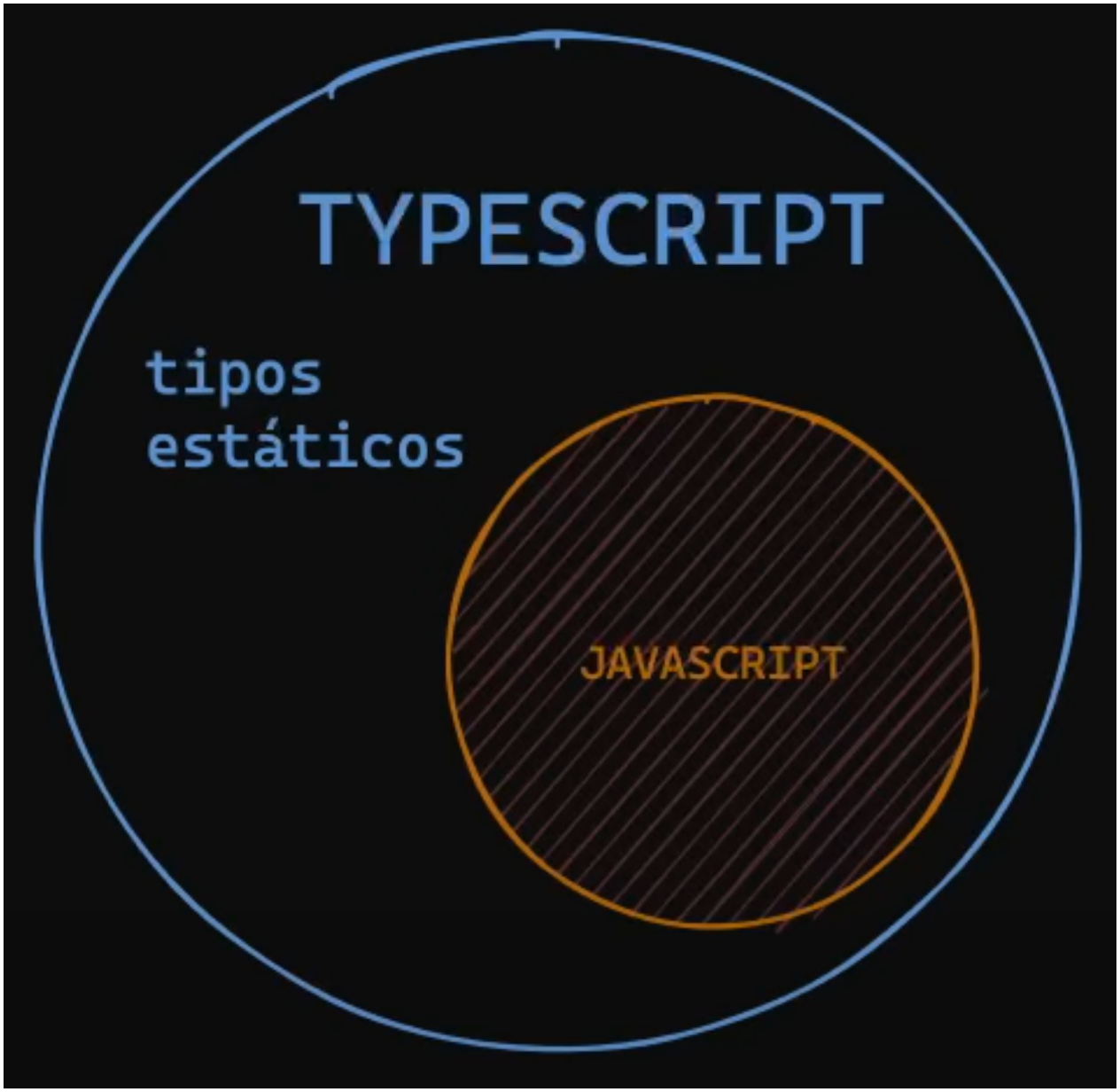
La información de este documento se obtuvo de un mini curso de Youtube: parte 1 y parte 2 y la plantilla de Lstlisting usada para Javascript fue tomada y modificada de este repositorio de Github.

1 Introducción

1.1 ¿Qué es Typescript?

Typescript (TS) es un súper set de Javascript (JS), es lo mismo pero con una sintaxis para tipos de datos. Se puede ver como una extensión de JS (Figura 1).

Figure 1: Visualización de TS sobre JS



TS no es un lenguaje de programación aparte como tal. TS no funciona en tiempo de ejecución, lo que llega al buscador como tal es JS. Esto es lo que el buscador recibiría de TS (1):

Table 1: Código JS generado de TS

Buscador (JS)	**Servidor (TS)**
"use strict"; const nombre = 'Miguel'	const nombre = 'Miguel'

1.2 ¿Porqué aprenderlo?

- Por popularidad.
- Los tipos de datos.
- Mayor control sobre los resultados de funciones según los parámetros pasados.

El siguiente código muestra la clara diferencia con respecto a los tipos de datos entre TS y JS:

Table 2: Aplicación de TS a código básico de JS

Lenguaje	**Código**	**Resultado**

JS	function suma(a, b) {
	return a + b
	}
		suma(4, 'hola')
		'4hola'

TS	function suma(a, b) {
	return a + b
	}
		Error

El código de JS te permite hacer la suma de dos tipos (siempre y cuando se pueda) pero no sabes a ciencia exacta qué tipos de datos puede llegar a ingresar el usuario, haciendo vulnerable el código a cosas que no se pueden predecir. Con TS en cambio, le dices que tipo esperas en una función y qué tipo regresará, controlando los las entradas y salidas. El código de JS en TS anterior muestra un error porque no se le indica a TS el tipo de los parámetros de la función **suma**.

2 ¿Qué es la inferencia?

La inferencia de TS se refiere a que el propio TS lee el código que estamos desarrollando e infiere (supone) el tipo de dato de las variables, funciones u objetos que estamos creando, si tenemos estos dos ejemplos:

```
let mascota = 'mia';

const persona = {
    name = 'juan',
    age = 30
}
```

TS infiere que la variable *mascota* es de tipo string por el valor contenido en esa variable, infiere que los atributos *persona.name* y *persona.age* son de tipo string y number, se le puede decir explícitamente el tipo, pero TS tiene esta cualidad de que infiere los tipos de las cosas. La manera de asignar un tipo a una variable, objeto o función es la siguiente

```
let booleano: boolean = true;
let cadena: string = 'hola';
let numero: number = 5;
let nulo: null = null;
let indefinido: undefined;
let cualquiera: any;
let desconocido: unknown;
```

Los tipos de datos básicos de JS son las cadenas (string), números (number), booleanos (boolean), null y undefined. Se utiliza el nombre de la variable, dos puntos y el tipo esperado, lo mismo para las propiedades de objetos, el valor de retorno de una función y las variables. Si se utiliza el método **typeof** con las variables tipo *any* y *unknown* lo que se regresará es el valor undefined.

Se recomienda ESCRIBIR LA MENOR CANTIDAD DE TIPOS DE DATOS en las variables, objetos o funciones, de eso se tiene que encargar TS. Se recomienda principalmente en variables u objetos que utilicen los tipos sencillos que se mencionaron anteriormente, para cosas más pesadas, como objetos de dependencias internas y externas, conexiones a bases de datos y otras cosas, es preferible especificar el tipo.

2.1 Inferencia con las funciones y sus parámetros

Por lo general, TS si logra inferir el tipo de los parámetros de una función, incluso el tipo del retorno de la función, en cualquier caso se recomienda revisar qué es lo que dice el IDE, editor de texto o TS con respecto al tipo inferido, si es any, asignar el tipo que realmente se espera, esto con el fin de seguir la regla de escribir la menor cantidad de tipos posibles.

En caso de mandar un objeto como parámetro a una función sin haberlo declarado antes también se recibe este objeto como any en todas sus propiedades, por lo que es recomendable tiparlo para evitar conflictos. Se puede tipar de dos maneras:

```
// opcion 2
function saludar({name, age}: {name: string, age: number}) {
    console.log(`Hola ${name} y tu edad es ${age}.`);
}

// opcion 2
function saludar(persona: {name: string, age: number}) {
    const { name, age } = persona
    console.log(`Hola ${name}, y tu edad es ${age}.`);
}
```

Con la segunda manera tendríamos que hacer *destructuring* del objeto para acceder a sus propiedades.

2.2 Inferencia en funciones como parámetro

Como sabemos, las funciones en JS se pueden usar como parámetro dentro de otra función, y el tipado tiene que ver con esta forma de trabajar los parámetros. Pongamos el siguiente ejemplo:

```
const sayHiFromFunction = (fn) => {
    return fn('Miguel')
}

sayHiFromFunction((name) => {
    console.log(`Hola ${name}$`)
})
```

TS tendría el problema de no poder identificar el tipo del parámetro **fn** y **name**, sabe que es una función parámetro pero no sabe qué tipo es, por lo que le asignará un tipo **any**, y como se ha dicho anteriormente, debemos evitar usar este tipo lo más que se pueda.

Hay una especia de tipo llamada Function, la cual le indica a JS que una función es una función como tal de manera muy general, es como asignarle el tipo Function a una función; utilizar este tipo para tipar una función es incorrecto, ya que si, una función será un tipo función, pero para los fines de lo que regresa o no nuestra función es incorrecto, las funciones siempre se deben de tipar según el tipo que vayan a retornar (si no retornan nada, se tipan como **void**)

Solucionando el ejemplo anterior, quedaría de la siguiente manera:

```
const sayHiFromFunction = (fn: (name: string) => void) => {
    return fn('Miguel')
}

const sayHi = (name: string) => {
    console.log(`Hola ${name}$`)
}

sayHiFromFunction(sayHi)
```

La función sayHi (que antes estaba insertada dentro de los parámetros de sayHiFromFunction) no regresa nada, pero tiene un parámetro string, por lo que sayHiFromFunction se tipa como que tiene una función parámetro que a la vez tiene un parámetro (*fn: (name: string)*) y no retorna nada (*fn: (name: string) => void*).

2.3 Inferencia en funciones anónimas

Por lo general, TS si logra inferir correctamente el tipo de una función anónima, pondremos de ejemplo lo siguiente:

```
const avengers = ['spiderman', 'hulk', 'avengers']

avengers.forEach(function (avenger) {
```

```
4     console.log(avenger.toUpperCase())
5 })
```

El arreglo **avengers** es de tipo string por la inferencia de TS, al usar un método de arreglos (forEach) el cual solicita una función que realice una acción, TS infiere que la función a utilizar también es string, esto por el tipo del arreglo.

3 Asignando tipos de datos

3.1 Funciones regulares

Para tipar una función regular (que utiliza la palabra reservada function), se hace de la siguiente manera:

```
1 function sumar(a: number, b: number) : number {
2     return a + b;
3 }
```

3.2 Funciones arrow

Para tipar una función arrow, se hace de la siguiente manera:

```
1 const sumar = (a: number, b: number): number => {
2     return a + b
3 }
```

3.3 Funciones anónimas

Recordemos que una función anónima es aquella que son aquellas que no han sido declaradas con un nombre, generalmente las encontramos en las funciones utilizadas por arreglos (map o filter por ejemplo). Se tipan de la siguiente manera:

```
1 const arreglo = ['a', 'b', 'c']

3 arreglo.forEach((item: string) => {
4     console.log(item)
5 })
```

3.4 Tipos especiales de TS

El tipo **any** se utiliza para, literalmente, decirle a TS que la variable u objeto puede ser de cualquier tipo, por lo cual, el lenguaje deja de recomendarnos métodos especiales para los tipos existente. Dicho en otras palabras, con any se ignora completamente el tipado que ofrece TS. Se recomienda usar lo menos posible este tipo.

El tipo **unknown** se utiliza cuando uno no sabe el tipo de dato que esperar de una variable, asignando este tipo a una variable TS no recomienda métodos especiales para tipos porque no sabe el tipo de la variable u objeto.

El tipo **never** se utiliza cuando se tiene una función donde se tiene una certeza del 100% de que esta no devolverá ningún tipo de variable. Generalmente se utiliza en funciones que lanzan un mensaje de error como la siguiente:

```
function throwError(message: string): never {
    throw new Error(message);
}
```

Otro ejemplo un poco más claro sobre la aparición del tipo never:

```
function fn(x: string | number) {
    if(typeof x == 'string') {
        // type string.
        // do something.
    }
    else if(typeof x == 'number') {
        // type number.
        // do something.
    }
    else {
        x // type never.
    }
}
```

4 Tipos propios

4.1 Types alias

Podemos crear alguna especie de tipo de dato para asignárselo a objetos y que estos tengan un tipo de dato en lugar de que sean simplemente objetos sin tipo; podría decirse que es como crear una clase, la clase es una clase como tal y tiene propiedades y métodos dentro de ella que son de cierto tipo, luego podemos crear objetos de la clase y estos objetos son del tipo de la clase. Visto en C# se ve así:

```
// se crea la clase.
class Persona
{
    // metodos y atributos de cierto tipo.
    public string Nombre;
    public int Edad;
}

// se crea el objeto.
Persona persona = new Persona();
```

Con los type alias sería así:

```
// se crea el alias.
type Persona = {
    Nombre: string
    Edad: number
}

let persona: Persona = {
    Nombre: 'thor',
    Edad: 1500
}
```

Ahora el objeto tiene un tipo definido.

4.2 Union types

Podemos crear un tipo el cual solamente pueda recibir ciertos valores o cierto tipo de dato, para el primer caso, es como si tuviéramos un if o switch donde, según el valor recibido, se le asigne otro valor a una variable, para el segundo caso, podemos decirle TS que la variable que estamos declarando puede ser un número o una cadena, una cadena o un booleano, un número o booleano, etc.

Veamos ambos casos

```
type heroLevel = 'low' | 'medium' | 'high'

var a : number | string

var hero1 : heroLevel = 'extreme' // error.
var hero2 : heroLevel = 'low' // correcto.
a = true // error.
a = 2 // correcto.
```

Incluso se puede optar por un tipo o un valor concreto:

```
var b : number | 'hola'
```

4.3 Templates union types

Podemos crear un tipo que esté constituido de otros tipos según un patrón o las necesidades que tengamos, por ejemplo, hablando de los Ids, podemos crear un formato de Id donde aparezcan 3 cadenas separadas por un guión:

```
type PersonaId = `${string}-${string}-${string}`
```

Este tipo ahora se puede usar dentro de objetos u otros tipos:

```
type Persona = {
    Id: PersonaId
    Nombre: string
    Edad: number
}
```

4.4 Propiedades opcionales

Como vimos anteriormente, podemos crear types alias, los cuales tienen propiedades de cierto tipo, podemos hacer que algunas de estas propiedades no sean obligatorias al momento de aplicar estos alias a objetos mediante el símbolo ?:

```
type Persona = {
    Id?: PersonaId
    Nombre: string
    Edad: number
}
```

Ahora, cuando se cree un objeto tipo Persona, no será obligatorio rellenar la propiedad Id.

4.5 Intersection types

Otra cosa muy interesante que se puede hacer es poder combinar tipos diferentes en uno solo, armando así un nuevo tipo constituido de otros, veamos este ejemplo: tenemos un tipo HeroBasicInfo, el cual contiene dos atributos obligatorios que son el nombre y edad de un héroe, también tenemos HeroProperties, el cual tiene otros atributos no obligatorios del héroe, como sería su status y escala de poder:

```
type heroLevel = 'low' | 'medium' | 'high'

type HeroBasicInfo = {
    nombre: string,
    edad: number
}

type HeroProperties = {
    isActive?: boolean,
    powerScale?: heroLevel
}
```

HeroLevel es un complemento visto en uno de los ejemplos anteriores. Si queremos combinar estos dos tipos se hace de la siguiente manera:

```
type Hero = HeroBasicInfo & HeroProperties;
```

```
let hero3 = {
    name: 'spiderman',
    age: 30
}
```

De esta manera, tenemos los atributos de ambos tipos en uno solo, así el tipo puede recibir menor cantidad de atributos y aligerar la carga de trabajo.

4.6 Types indexing

En caso de que tengamos un tipo con un objeto como atributo, al momento de querer acceder a este objeto y querer asignar valores en una variable o constante, podemos utilizar el type indexing para lograr este cometido:

```
type HeroProperties = {
    isActive: boolean,
    address: {
        planet: string,
        city: string
    }
}

const addressHero: HeroProperties['address'] = {
    planet: 'Tierra',
    city: 'Madrid'
}
```

Vemos que se crea el tipo HeroProperties el cual tiene un objeto llamado 'address' dentro, al crear una variable tipo HeroProperties, podemos acceder al objeto dentro del tipo solamente utilizando los corchetes con el nombre del objeto que deseamos utilizar, como se ve en la constante addressHero.

4.7 Types de valores y funciones

Si declaramos un objeto con ciertos atributos de x tipo, podemos crear un tipo que sea igual a este objeto:

```
const address = {
    planet: 'Tierra',
    city: 'Madrid'
}

type Address = typeof address
```

Esto se puede transportar a lo que regresa un función:

```
function createAddress() {
```

```
    return {
        planet: 'Tierra',
        city: 'Madrid'
    }
}

type Address = ReturnType<typeof createAddress>
```

ReturnType recupera el tipo de alguna función que le pases, en este caso recupera el tipo del objeto que regresa la función createAddress, el cual es un objeto con dos atributos tipo string.

5 Colecciones de datos

5.1 Arreglos

La forma típica para declarar un arreglo en JS es la siguiente:

```
const languages = [];
```

Sin embargo, TS tendrá problemas al hacer esto, ya que esta forma de declaración le está diciendo a TS que este arreglo siempre contendrá valores tipo *never* (visto en los tipos especiales de TS), por lo que, si intentamos hacer un push con cualquier tipo de valor, tendremos problemas.

La manera correcta de declarar un arreglo en TS es remarcando el tipo de dato del arreglo:

```
// opcion 1.
const languages: string[] = []

// opcion 2.
const languages: Array<string> = []
```

¿Cuál de las dos escoger?, depende de los gustos.

¿Qué pasa si quiere almacenar dos o más tipos en mi arreglo?, esta es la manera correcta e incorrecta de hacerlo:

```
// incorrecto.
const languages: Arrary<string> | Array<number> = []
const languages: string[] | number[] = []

// correcto.
const languages: (string | number)[] = []
```

Nota: utilizando el Playground de TS, la forma *const languages: Arrary<string> | Array<number> = []* parece ser que también es correcta.

5.2 Matrices y Tuplas

Una matriz es una colección de datos de n dimensiones mayores a lo que vendría siendo un arreglo convencional, una tupla es una colección de datos que se puede encontrar en diferentes lenguajes de programación, suele ser una agrupación de datos que, por su naturaleza, vienen todos juntos, esta agrupación suele ser de un mismo número de valores y tipo.

Las tuplas y matrices también pueden ser creadas en TS según el tipo de declaración de TS, veamos el siguiente ejemplo: tenemos el juego del gato, donde tenemos una matriz 3x3 y ponemos los típicos valores 'x' ó 'o', tendría este aspecto:

```
['x', 'o', 'o']
['o', 'x', 'o']
['o', 'o', 'x']
```

Para lograr esto, lo primero que se nos podría ocurrir sería crear un arreglo de arreglos:

```
const gameBoard: string[][] = {
    ['x', 'o', 'o'],
    ['o', 'x', 'o'],
    ['o', 'o', 'x']
}
```

El gran detalle con esto es que podemos seguir asignando valores a este arreglo de arreglos y podemos asignarle algo diferente a los valores esperados (asignar el valor '234effdf3e' en lugar de 'x' o 'o'). Si hacemos la corrección queda así:

```
type Cells = 'x' | 'o' | ''
type GameBoard = [
    [Cells, Cells, Cells],
    [Cells, Cells, Cells],
    [Cells, Cells, Cells]
]

const gameBoard: GameBoard = [
    ['x', 'o', 'o'],
    ['o', 'x', 'o'],
    ['o', 'o', 'x']
]
```

Con esto, primero definimos el tipo donde se aceptan solo los valores que deseamos (*Cells*), luego definimos otro tipo que solo contenga arreglos de tres elementos tipo Cells (*GameBoard*), finalmente declaramos una constante que es del tipo GameBoard y con eso satisface todas las necesidades correspondientes a una matriz 3x3 con los valores requeridos solamente.

Un problema con las tuplas es que son mutables (error del lenguaje directamente), si utilizamos el siguiente código:

```
type RGB = [number, number, number]

const rgb: RGB = [0,0,0]

rgb.push(4)
```

Ahora la constante *rgb* tendrá cuatro espacios o variables para almacenar datos, este error es del lenguaje y no ha habido solución, por lo que, si no quieres que se acceda a la información de la tupla, la puedes volver *readonly*:

```
type RGB = readonly [number, number, number]
```

5.3 Enums

Los *enums* son un tipo de colección de datos donde se almacena una lista de valores conocidos, esta lista es finita y suele ser corta, de alrededor de 20 itemos por enum.

Los enums no existen en JS, por lo que al momento de utilizarlos en TS toman otro aspecto que veremos más adelante, por lo pronto, esta es la sintaxis para declarar un enum:

```
enum ERROR_TYPES {
    NOT_FOUND,
    UNAUTHORIZED,
    FORBIDDEN
}

function mostrarMensaje(tipoDeError: ERROR_TYPES) {
    if(tipoDeError == ERROR_TYPES.NOT_FOUND)
        console.log('No se encuentra el recurso')
    else if(tipoDeError == ERROR_TYPES.UNAUTHORIZ)
        console.log('No tienes el permiso para acceder')
    else if(tipoDeError == ERROR_TYPES.FORBIDDEN)
        console.log('Error')
}
```

Vemos que se suelen utilizar con listas de cosas que ya conocemos, como una lista de tipos de errores, sexos, días de la semana, meses, etc. Se declara el enum con sus valores y se puede utilizar en código. Recordemos que los enums no existen en JS, por lo que si vemos en el compilador online de TS:

```
"use strict";
var ERROR_TYPES;
(function (ERROR_TYPES) {
    ERROR_TYPES[ERROR_TYPES["NOT_FOUND"] = 0] = "NOT_FOUND";
    ERROR_TYPES[ERROR_TYPES["UNAUTHORIZED"] = 1] = "UNAUTHORIZED";
    ERROR_TYPES[ERROR_TYPES["FORBIDDEN"] = 2] = "FORBIDDEN";
})(ERROR_TYPES || (ERROR_TYPES = {}));
function mostrarMensaje(tipoDeError) {
    if (tipoDeError == ERROR_TYPES.NOT_FOUND)
        console.log('No se encuentra el recurso');
    else if (tipoDeError == ERROR_TYPES.UNAUTHORIZED)
        console.log('No tienes el permiso para acceder');
    else if (tipoDeError == ERROR_TYPES.FORBIDDEN)
        console.log('Error');
}
```

Esto genera muchísimo código extra, que en general declara un objeto o función que sirve para realizar la funcionalidad del enum en JS. Es mucho código, si agregamos la palabra reservada *const* al inicio de nuestro enum pasa otra cosa:

```
"use strict";
function mostrarMensaje(tipoDeError) {
    if (tipoDeError == 0 /* ERROR_TYPES.NOT_FOUND */)
        console.log('No se encuentra el recurso');
    else if (tipoDeError == 1 /* ERROR_TYPES.UNAUTHORIZED */)
        console.log('No tienes el permiso para acceder');
```

```
    else if (tipoDeError == 2 /* ERROR_TYPES.FORBIDDEN */)
        console.log('Error');
}
```

Nos hemos desecho del código extra y se sustituye el rastro del enum con unos valores numéricos **comenzando desde el 0** (para este caso del *if*). Pero aquí podrás pensar que tal vez no quieras utilizar estos índices, que prefieres utilizar cadenas de texto, hay una solución para eso:

Código TS

```
const enum ERROR_TYPES {
    NOT_FOUND = 'not found',
    UNAUTHORIZED = 'unauthorized',
    FORBIDDEN = 'forbidden'
}

function mostrarMensaje(tipoDeError: ERROR_TYPES) {
    if(tipoDeError == ERROR_TYPES.NOT_FOUND)
        console.log('No se encuentra el recurso')
    else if(tipoDeError == ERROR_TYPES.UNAUTHORIZED)
        console.log('No tienes el permiso para acceder')
    else if(tipoDeError == ERROR_TYPES.FORBIDDEN)
        console.log('Error')
}
```

Código JS compilado

```
"use strict";
function mostrarMensaje(tipoDeError) {
    /* ERROR_TYPES.NOT_FOUND */
    if (tipoDeError == "not found")
        console.log('No se encuentra el recurso');
    /* ERROR_TYPES.UNAUTHORIZED */
    else if (tipoDeError == "unauthorized")
        console.log('No tienes el permiso para acceder');
    /* ERROR_TYPES.FORBIDDEN */
    else if (tipoDeError == "forbidden")
        console.log('Error');
}
```

Se sustituyen los valores numéricos por cadenas de texto, si quitamos el *const* en este segundo caso no habría inconveniente alguno.

¿Cuál utilizar?, ¿Con o sin *const*?

Esta pregunta se responde viendo para qué es el proyecto, si tu proyecto es interno y no habrá un tercero que lo consumirá como un servicio o librería, puedes utilizar los enums con *const*, si fuera caso contrario se recomienda omitir este último para que, quien te consuma, vea el código generado del enum.

6 Aserciones

6.1 Con elementos HTML

Recordemos que JS se utiliza en conjunto con HTML para desarrollo web, TS no se queda atrás y podemos utilizar la inferencia con los elementos HTML de nuestro proyecto, pero hay que saber utilizarlo para que el código TS funcione correctamente con el de JS (compilado y en ejecución). Tenemos el siguiente ejemplo muy sencillo

```
1 const canvas = document.getElementById('canvas')

3 const ctx = canvas.getContext('2d')
```

Podemos recuperar los atributos e información de un elemento HTML mediante el objeto y función *document.getElementById('canvas')*, algo que se ha visto en JS también, con TS podemos inferir que la constante *const* es de tipo canvas o tipo HTML, existe un tipo genérico para los elementos HTML llamado **HTMLElement**, la desventaja de este tipo es que es general, no especifica qué tipo de elemento es el que queremos recuperar, para nuestra fortuna, existen tipos más específicos para nuestros elementos:

```
1 const canvas = document.getElementById('canvas')
2     as HTMLCanvasEelement

4 const ctx = canvas.getContext('2d')
```

Con esto TS ya sabe que la constante *canvas* es un elemento Canvas de HTML, sin embargo, esta constante puede recibir un valor cualquiera o puede recibir *null*, ya que el código no sabe si realmente recibirá un objeto canvas o no, por lo que tenemos que meter alguna condicional para asegurar que recibiremos un valor:

```
1 const canvas = document.getElementById('canvas')
2     as HTMLCanvasElement

4 if (canvas != null) {
5     const ctx = canvas.getContext('2d')
6 }
```

El código anterior nos está atando ahora a que lo que recibamos es un elemento Canvas, pero qué pasa si recibimos algún elemento diferente (p, span, img, etc.), debemos poder asegurar también que el elemento recibido es el esperado:

```
1 const canvas = document.getElementById('canvas')

3 if (canvas != null && canvas instanceof HTMLCanvasElement) {
4     const ctx = canvas.getContext('2d')
5 }
```

Esta seria la solución definitiva a nivel TS y JS para nuestro problema, veamos qué aspecto tiene en código JS:

```
"use strict";

const canvas = document.getElementById('canvas');

if (canvas != null && canvas instanceof HTMLCanvasElement) {
    const ctx = canvas.getContext('2d');
}
```

Como se puede apreciar, el código es igual en TS y en JS, la diferencia está en que TS logra inferir el tipo de elemento recibido en la constante canvas y se puede trabajar de manera más segura. ¿Qué aspecto tendría el código JS si si hubiéramos utilizado el penúltimo código TS?

```
"use strict";

const canvas = document.getElementById('canvas');

if (canvas != null) {
    const ctx = canvas.getContext('2d');
}
```

Vemos que la instrucción *as HTMLCanvasElement* al final de nuestra constante desaparece, por lo que en el futuro podríamos romper el código.

6.2 Con Fetching

Otra de las cosas que se suele hacer con JS es el consumir datos de una API, sabemos que se debe declarar una constante con la URL de la API que queremos que consumir y utilizar algún método para consumirla, al obtener una respuesta no negativa tenemos acceso al objeto de respuesta y con hacer lo que se requiera, con TS también podemos realizar una aserción para controlar el los datos y tipos de la respuesta de la API. Tenemos el siguiente código:

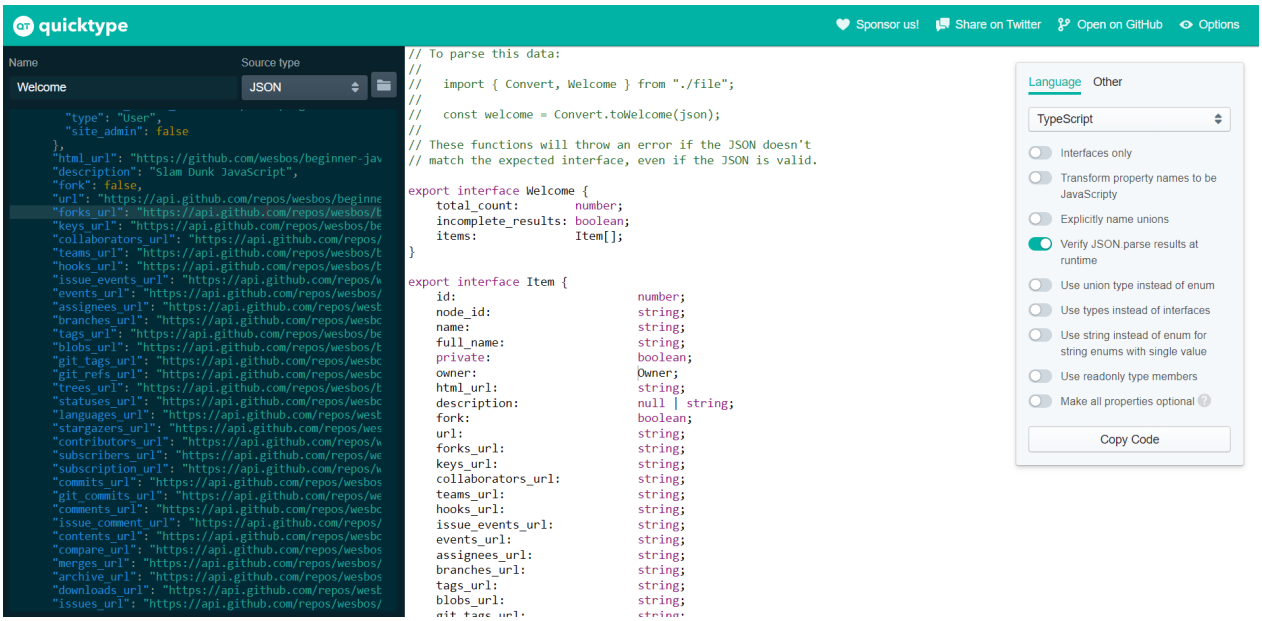
```
const API_URL =
    "https://api.github.com/search/repositories?q=javascript"

// response es del tipo 'Response'
const response = await fetch(API_URL)

if (!response.ok) {
    throw new Error('Request failed')
}

// data es del tipo 'any', se debe indicar que tipo es.
const data = await response.json()
```

```
const repos = data.items.map(repo => {
    console.log(repo)
})
```

Como tal, TS no tiene un tipo específico para las respuestas de una API, tendríamos que crear nosotros un tipo con todos los tipos de los atributos del objeto JSON que nos retorne la respuesta de la API y asignárselo a la constante o variable.

Crear el tipo manualmente puede ser muy tardado y tedioso, existe el sitio web quicktype que recibe un objeto JSON como respuesta de una API y te genera el tipo personalizado con todos los tipos de los atributos del objeto, es un método más rápido para trabajar esta parte. Si acedemos a la URL de la api del código de arriba, nos regresará un JSON bastante extenso, lo copiamos y pegamos en *quicktype* y nos generará los tipos automáticamente. Tome en cuenta lo siguiente:

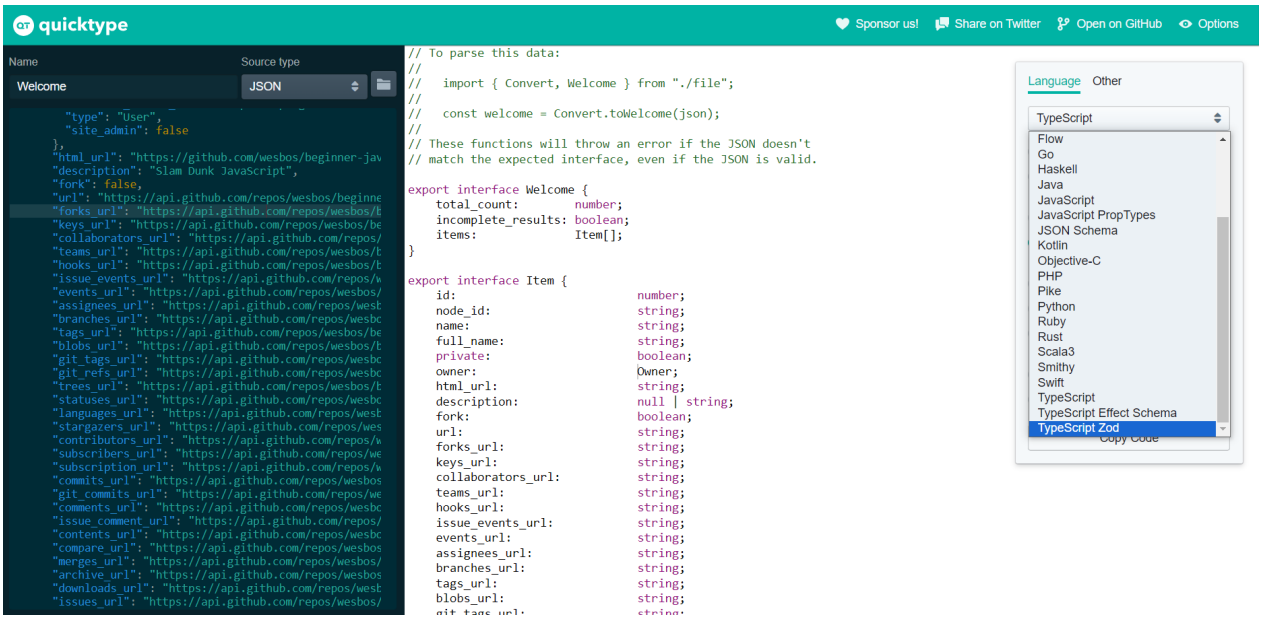
Figure 2: Área de trabajo de *quicktype*



Nota: no olvidar cambiar el nombre del tipo que se va a generar en quicktype en la parte superior de la sección izquierda donde se pega la respuesta.

La figura anterior muestra el área de trabajo, vemos que en el lado derecho hay una ventana flotante con varias opciones, el lenguaje seleccionado es TS pero se pueden seleccionar más, podemos marcar y desmarcar otras opciones según nuestros requerimientos. Hay otras versiones que podemos seleccionar de TS, en especial TypeScript Zod, donde se incluye una librería para validación de tipos de datos, esta librería debe instalarse y tomará un poco más de tiempo que termine su ejecución, pero añade el factor de validación a nivel TS y JS.

Figure 3: Selección del lenguaje a TypeScript Zod



Si pegamos el código generado con el lenguaje TS sin Zod en el Playground de TS, tenemos un código JS como este:

```
// To parse this data:
//
//   import { Convert, Welcome } from "./file";
//
//   const welcome = Convert.toWelcome(json);
//
// These functions will throw an error if the JSON doesn't
// match the expected interface, even if the JSON is valid.
export var DefaultBranch;
(function (DefaultBranch) {
    DefaultBranch["Dev"] = "dev";
    DefaultBranch["Main"] = "main";
    DefaultBranch["Master"] = "master";
})(DefaultBranch || (DefaultBranch = {}));
export var Language;
(function (Language) {
    Language["CSS"] = "CSS";
    Language["HTML"] = "HTML";
    Language["JavaScript"] = "JavaScript";
    Language["TypeScript"] = "TypeScript";
})(Language || (Language = {}));
export var Type;
(function (Type) {
    Type["Organization"] = "Organization";
    Type["User"] = "User";
})(Type || (Type = {}));
```

```
27 export var Visibility;
28 (function (Visibility) {
29     Visibility["Public"] = "public";
30 })(Visibility || (Visibility = {}));
31 // Converts JSON strings to/from your types
32 // and asserts the results of JSON.parse at runtime
33 export class Convert {
34     static toWelcome(json) {
35         return cast(JSON.parse(json), r("Welcome"));
36     }
37     static welcomeToJson(value) {
38         return JSON.stringify
39             (uncast(value, r("Welcome")), null, 2);
40     }
41 }
42 function invalidValue(typ, val, key, parent = '') {
43     const prettyTyp = prettyTypeName(typ);
44     const parentText = parent ? ` on ${parent}` : '';
45     const keyText = key ? ` for key "${key}"` : '';
46     throw Error(`Invalid value${keyText}${parentText}.
47         Expected ${prettyTyp} but got ${JSON.stringify(val)}`);
48 }
49 function prettyTypeName(typ) {
50     if (Array.isArray(typ)) {
51         if (typ.length === 2 && typ[0] === undefined) {
52             return `an optional ${prettyTypeName(typ[1])}`;
53         }
54         else {
55             return `one of [${typ.map(a =>
56             {
57                 return prettyTypeName(a);
58             }).join(", ")}]`;
59         }
60     }
61     else if (typeof typ === "object" &&
62             typ.literal !== undefined) {
63         return typ.literal;
64     }
65     else {
66         return typeof typ;
67     }
68 }
69 function jsonToJSProps(typ) {
70     if (typ.jsonToJS === undefined) {
71         const map = {};
72         typ.props.forEach((p) => map[p.json] = {
73             key: p.js, typ: p.typ
74         });
75         typ.jsonToJS = map;
```

```
76      }
77      return typ.jsonToJS;
78  }
79  function jsToJSONProps(typ) {
80      if (typ.jsToJSON === undefined) {
81          const map = {};
82          typ.props.forEach((p) => map[p.js] = {
83              key: p.json, typ: p.typ
84          });
85          typ.jsToJSON = map;
86      }
87      return typ.jsToJSON;
88  }
89  function transform(val, typ, getProps, key = '', parent = '') {
90      function transformPrimitive(typ, val) {
91          if (typeof typ === typeof val)
92              return val;
93          return invalidValue(typ, val, key, parent);
94      }
95      function transformUnion(typs, val) {
96          // val must validate against one typ in typs
97          const l = typs.length;
98          for (let i = 0; i < l; i++) {
99              const typ = typs[i];
100             try {
101                 return transform(val, typ, getProps);
102             }
103             catch (_) { }
104         }
105         return invalidValue(typs, val, key, parent);
106     }
107     function transformEnum(cases, val) {
108         if (cases.indexOf(val) !== -1)
109             return val;
110         return invalidValue(cases.map(a => {
111             return l(a);
112         }), val, key, parent);
113     }
114     function transformArray(typ, val) {
115         // val must be an array with no invalid elements
116         if (!Array.isArray(val))
117             return invalidValue(l("array"), val, key, parent);
118         return val.map(el => transform(el, typ, getProps));
119     }
120     function transformDate(val) {
121         if (val === null) {
122             return null;
123         }
124         const d = new Date(val);
```

```
125         if (isNaN(d.valueOf())) {
126             return invalidValue(l("Date"), val, key, parent);
127         }
128         return d;
129     }
130     function transformObject(props, additional, val) {
131         if (val === null ||
132             typeof val !== "object" ||
133             Array.isArray(val)) {
134             return invalidValue
135                 (l(ref || "object"), val, key, parent);
136         }
137         const result = {};
138         Object.getOwnPropertyNames(props).forEach(key => {
139             const prop = props[key];
140             const v = Object.prototype.hasOwnProperty
141             .call(val, key)? val[key] : undefined;
142             result[prop.key] = transform(v, prop.typ,
143             getProps, key, ref);
144         });
145         Object.getOwnPropertyNames(val).forEach(key => {
146             if (!Object.prototype.hasOwnProperty
147             .call(props, key)) {
148                 result[key] = transform(val[key], additional,
149                 getProps, key, ref);
150             }
151         });
152         return result;
153     }
154     if (typ === "any")
155         return val;
156     if (typ === null) {
157         if (val === null)
158             return val;
159         return invalidValue(typ, val, key, parent);
160     }
161     if (typ === false)
162         return invalidValue(typ, val, key, parent);
163     let ref = undefined;
164     while (typeof typ === "object" && typ.ref !== undefined) {
165         ref = typ.ref;
166         typ = typeMap[typ.ref];
167     }
168     if (Array.isArray(typ))
169         return transformEnum(typ, val);
170     if (typeof typ === "object") {
171         return typ.hasOwnProperty("unionMembers") ?
172         transformUnion(typ.unionMembers, val)
173         :
```

```
174         typ.hasOwnProperty("arrayItems") ? transformArray
175             (typ.arrayItems, val)
176         :
177         yp.hasOwnProperty("props") ? transformObject
178             (getProps(typ), typ.additional, val)
179         :
180         invalidValue(typ, val, key, parent);
181     }
182     // Numbers can be parsed by Date but shouldn't be.
183     if (typ === Date && typeof val !== "number")
184         return transformDate(val);
185     return transformPrimitive(typ, val);
186 }
187 function cast(val, typ) {
188     return transform(val, typ, jsonToJSProps);
189 }
190 function uncast(val, typ) {
191     return transform(val, typ, jsToJSONProps);
192 }
193 function l(typ) {
194     return { literal: typ };
195 }
196 function a(typ) {
197     return { arrayItems: typ };
198 }
199 function u(...typs) {
200     return { unionMembers: typs };
201 }
202 function o(props, additional) {
203     return { props, additional };
204 }
205 function m(additional) {
206     return { props: [], additional };
207 }
208 function r(name) {
209     return { ref: name };
210 }
211 const typeMap = {
212     "Welcome": o([
213         { json: "total_count", js: "total_count", typ: 0 },
214         { json: "incomplete_results", js: "incomplete_results",
215             typ: true },
216         { json: "items", js: "items", typ: a(r("Item")) },
217     ], false),
218     "Item": o([
219         { json: "id", js: "id", typ: 0 },
220         { json: "node_id", js: "node_id", typ: "" },
221         { json: "name", js: "name", typ: "" },
222         { json: "full_name", js: "full_name", typ: "" },
```

```
223         { json: "private", js: "private", typ: true },
224         { json: "owner", js: "owner", typ: r("Owner") },
225         { json: "html_url", js: "html_url", typ: "" },
226         { json: "description", js: "description",
227             typ: u(null, "") },
228         { json: "fork", js: "fork", typ: true },
229         { json: "url", js: "url", typ: "" },
230         { json: "forks_url", js: "forks_url", typ: "" },
231         { json: "keys_url", js: "keys_url", typ: "" },
232         { json: "collaborators_url", js: "collaborators_url",
233             typ: "" },
234         { json: "teams_url", js: "teams_url", typ: "" },
235         { json: "hooks_url", js: "hooks_url", typ: "" },
236         { json: "issue_events_url", js: "issue_events_url",
237             typ: "" },
238         { json: "events_url", js: "events_url", typ: "" },
239         { json: "assignees_url", js: "assignees_url", typ: "" },
240         { json: "branches_url", js: "branches_url", typ: "" },
241         { json: "tags_url", js: "tags_url", typ: "" },
242         { json: "blobs_url", js: "blobs_url", typ: "" },
243         { json: "git_tags_url", js: "git_tags_url", typ: "" },
244         { json: "git_refs_url", js: "git_refs_url", typ: "" },
245         { json: "trees_url", js: "trees_url", typ: "" },
246         { json: "statuses_url", js: "statuses_url", typ: "" },
247         { json: "languages_url", js: "languages_url", typ: "" },
248         { json: "stargazers_url", js: "stargazers_url", typ: "" },
249         { json: "contributors_url", js: "contributors_url",
250             typ: "" },
251         { json: "subscribers_url", js: "subscribers_url",
252             typ: "" },
253         { json: "subscription_url", js: "subscription_url",
254             typ: "" },
255         { json: "commits_url", js: "commits_url", typ: "" },
256         { json: "git_commits_url", js: "git_commits_url",
257             typ: "" },
258         { json: "comments_url", js: "comments_url", typ: "" },
259         { json: "issue_comment_url", js: "issue_comment_url",
260             typ: "" },
261         { json: "contents_url", js: "contents_url", typ: "" },
262         { json: "compare_url", js: "compare_url", typ: "" },
263         { json: "merges_url", js: "merges_url", typ: "" },
264         { json: "archive_url", js: "archive_url", typ: "" },
265         { json: "downloads_url", js: "downloads_url", typ: "" },
266         { json: "issues_url", js: "issues_url", typ: "" },
267         { json: "pulls_url", js: "pulls_url", typ: "" },
268         { json: "milestones_url", js: "milestones_url", typ: "" },
269         { json: "notifications_url", js: "notifications_url",
270             typ: "" },
271         { json: "labels_url", js: "labels_url", typ: "" },
```

```
272         { json: "releases_url", js: "releases_url", typ: "" },
273         { json: "deployments_url", js: "deployments_url",
274             typ: "" },
275         { json: "created_at", js: "created_at", typ: Date },
276         { json: "updated_at", js: "updated_at", typ: Date },
277         { json: "pushed_at", js: "pushed_at", typ: Date },
278         { json: "git_url", js: "git_url", typ: "" },
279         { json: "ssh_url", js: "ssh_url", typ: "" },
280         { json: "clone_url", js: "clone_url", typ: "" },
281         { json: "svn_url", js: "svn_url", typ: "" },
282         { json: "homepage", js: "homepage", typ: u(null, "") },
283         { json: "size", js: "size", typ: 0 },
284         { json: "stargazers_count", js: "stargazers_count",
285             typ: 0 },
286         { json: "watchers_count", js: "watchers_count", typ: 0 },
287         { json: "language", js: "language", typ: u(r("Language"),
288             null) },
289         { json: "has_issues", js: "has_issues", typ: true },
290         { json: "has_projects", js: "has_projects", typ: true },
291         { json: "has_downloads", js: "has_downloads", typ: true },
292         { json: "has_wiki", js: "has_wiki", typ: true },
293         { json: "has_pages", js: "has_pages", typ: true },
294         { json: "has_discussions", js: "has_discussions",
295             typ: true },
296         { json: "forks_count", js: "forks_count", typ: 0 },
297         { json: "mirror_url", js: "mirror_url", typ: null },
298         { json: "archived", js: "archived", typ: true },
299         { json: "disabled", js: "disabled", typ: true },
300         { json: "open_issues_count", js: "open_issues_count",
301             typ: 0 },
302         { json: "license", js: "license", typ: u(r("License"),
303             null) },
304         { json: "allow_forking", js: "allow_forking", typ: true },
305         { json: "is_template", js: "is_template", typ: true },
306         { json: "web_commit_signoff_required", js:
307             "web_commit_signoff_required", typ: true },
308         { json: "topics", js: "topics", typ: a("") },
309         { json: "visibility", js: "visibility",
310             typ: r("Visibility") },
311         { json: "forks", js: "forks", typ: 0 },
312         { json: "open_issues", js: "open_issues", typ: 0 },
313         { json: "watchers", js: "watchers", typ: 0 },
314         { json: "default_branch", js: "default_branch",
315             typ: r("DefaultBranch") },
316         { json: "score", js: "score", typ: 0 },
317     ], false),
318     "License": o([
319         { json: "key", js: "key", typ: "" },
320         { json: "name", js: "name", typ: "" },
```

```
321         { json: "spdx_id", js: "spdx_id", typ: "" },
322         { json: "url", js: "url", typ: u(null, "") },
323         { json: "node_id", js: "node_id", typ: "" },
324     ], false),
325     "Owner": o([
326         { json: "login", js: "login", typ: "" },
327         { json: "id", js: "id", typ: 0 },
328         { json: "node_id", js: "node_id", typ: "" },
329         { json: "avatar_url", js: "avatar_url", typ: "" },
330         { json: "gravatar_id", js: "gravatar_id", typ: "" },
331         { json: "url", js: "url", typ: "" },
332         { json: "html_url", js: "html_url", typ: "" },
333         { json: "followers_url", js: "followers_url", typ: "" },
334         { json: "following_url", js: "following_url", typ: "" },
335         { json: "gists_url", js: "gists_url", typ: "" },
336         { json: "starred_url", js: "starred_url", typ: "" },
337         { json: "subscriptions_url", js: "subscriptions_url",
338             typ: "" },
339         { json: "organizations_url", js: "organizations_url",
340             typ: "" },
341         { json: "repos_url", js: "repos_url", typ: "" },
342         { json: "events_url", js: "events_url", typ: "" },
343         { json: "received_events_url", js: "received_events_url",
344             typ: "" },
345         { json: "type", js: "type", typ: r("Type") },
346         { json: "site_admin", js: "site_admin", typ: true },
347     ], false),
348     "DefaultBranch": [
349         "dev",
350         "main",
351         "master",
352     ],
353     "Language": [
354         "CSS",
355         "HTML",
356         "JavaScript",
357         "TypeScript",
358     ],
359     "Type": [
360         "Organization",
361         "User",
362     ],
363     "Visibility": [
364         "public",
365     ],
366 };

368 const API_URL =
369     "https://api.github.com/search/repositories?q=javascript";
```

```
// response es del tipo 'Response'
const response = await fetch(API_URL);

if (!response.ok) {
    throw new Error('Request failed');
}

// data es del tipo 'GithibAPIResponse'
const data = await response.json() as GithibAPIResponse;

const repos = data.items.map(repo => {
    console.log(repo);
});
```

En el Playground, el código de JS funciona y el de TS está tipado.

7 Interfaces

Las interfaces son un objeto que se suele utilizar mucho en el área de POO, consisten en una especie de clases donde es obligatorio utilizar todos los atributos y métodos que residen dentro de la interfase.

Si hacemos memoria, esto se parece mucho a los type alias vistos con anterioridad, y es que son súper parecidos pero tienen características que los pueden diferenciar y uno puede optar por utilizar type alias, interfaces o una combinación de ambas.

Esta es la forma de declarar una interface:

```
// con types alias
// type Hero = {
//     id: string
//     name: string
//     age: number
// }

// con interfaces
interface Hero {
    id: string
    name: string
    age: number
}

const hero: Hero = {
    id: '1',
    name: 'spiderman',
    age: 30
}
```

Podemos anidar una interface dentro de otra y extender el funcionamiento de una interface con otra (algún tipo de herencia):

```
interface Productos {
    id: number
    nombre: string
    precio: number
    quantity: number
}

// anidacion de interfaces.
interface CarritoDeCompras {
    totalPrice: number
    productos: Productos[]
}

// expande el funcionamiento de una interface con otra.
interface Tenis extends Productos {
    talla: string
}

const carritoCompras: CarritoDeCompras = {
    totalPrice: 1000,
    productos: [
        {
            id: 1,
            nombre: 'camisa',
            precio: 1000,
            quantity: 1,
        }
    ]
}
```

Para el primer caso, podemos crear un objeto de la interface CarritoDeCompra y en él podemos almacenar la información referente a un carrito de compra y una arreglo con los productos que se comprarán. Para el segundo caso, podemos crear un objeto de la interface Tenis que cuenta con todas las propiedades de Produtos y le expande una nueva propidad llamada *talla*.

Poniendo este último ejemplo en el Playground de TS, este es el código JS que se compila:

```
"use strict";

const carritoCompras = {
    totalPrice: 1000,
    productos: [
        {
            id: 1,
            nombre: 'camisa',
            precio: 1000,
```

```
            quantity: 1,
        }
    ]
};
```

Todas las interfaces desaparecen, lo mismo que se ha visto con el resto de ejemplos que se han desarrollado, claro, las interfaces no realizan comprobación o validación de datos.

7.1 Tipar funciones dentro de interfaces

Al igual que con la inferencia de funciones o la asignación de sus tipos, hay dos formas de asignar un tipo a las funciones dentro de una interface que depende de los gustos de cada quien para tomarla:

```
// opcion 1.
interface OperacionesCarrito {
    add: (product: Producto) => void
    remove: (id: number) => void
    clear: () => void
}

// opcion 2.
interface OperacionesCarrito {
    add(product: Producto): void
    remove(id: number): void
    clear(): void
}
```

Recordemos que los parámetros de las funciones y el retorno de la misma se debe tipar, en los casos anteriores se puso de ejemplo funciones con y sin parámetros, la palabra *void* se sustituye por algún tipo disponible.

Una característica curiosa de las interfaces es que podemos duplicar su declaración variando la cantidad y nombres de las propiedades y métodos que tengamos en cada declaración. En el ejemplo anterior TS lanzaría un error donde diría directamente que tenemos declarados más de una vez la interface *OperacionesCarrito*, si separamos de la siguiente manera esta declaración el error desaparece.

```
interface OperacionesCarrito {
    add: (product: Producto) => void
    remove: (id: number) => void
}

interface OperacionesCarrito {
    clear: () => void
}
```

Esto ocasiona que queramos declarar en varias partes del código la misma interface para agregar funcionalidades según alguna condicional o requerimiento, se sugiere abstenerse de esta práctica.

7.2 Interfaces vs Type aliases

Como se pudo leer en la descripción de estos dos conceptos, ambos comparten muchas similitudes, la gran pregunta es, ¿cuál es mejor o por cuál inclinarse más?

La siguiente figura muestra algunas características fundamentales extraídos de este link en conjunto con algunas conclusiones sacadas del estudio de estos dos conceptos:

Figure 4: Diferencias entre Interfaces y Types aliases

Las interfaces no pueden crear tipos u objetos que estén altamente relacionados con los tipos primitivos (strings, numbers, bool, etc.), los types alias si pueden mediante las template

union types, además de que estos alias pueden seleccionar entre varios tipos cuál resultarán siendo.

Para temas de clases, objetos y POO en general, podríamos seleccionar las interfaces, fuera de eso, nos podemos quedar con los alias sin ningún problema.

8 Narrowing

El concepto **narrowing** se refiere a que aseguremos que una constante o variable sea del tipo esperado para utilizar los métodos de su tipo, el narrowing podría verse como la solución a un problema que puede llegar a ocurrir con los unions types, tenemos el siguiente ejemplo:

```
function mostrarLongitud(objeto: number | string) {
    return objeto.length // da error.
}
```

Si utilizamos el ejemplo anterior así como viene, al retornar la longitud del parámetro *objeto* nos da un error porque este parámetro puede ser un número o una cadena, el método *length* solo funciona con cadenas, para poder solucionar esto, podemos aplicar una condicional:

```
function mostrarLongitud(objeto: number | string) {
    if(typeof objeto == 'string') {
        return objeto.length
    }

    return objeto.toString().length()
}
```

Esta solución es para tipos básicos, probemos ahora con una interface (se puede sustituir por un tipo más robusto):

```
interface Mario {
    company: 'nintendo',
    name: string,
    saltar: () => void
}

interface Sonic {
    company: 'sega',
    name: string,
    correr: () => void
}

type Personaje = Mario | Sonic

function jugar(personaje: Personaje) {
    console.log(personaje.correr()) // da error.
```

```
}
```

Con estas estructuras más robustas resolveremos el problema de los tipos asignado a otro tipo. Se tienen dos interfaces y a un tipo nuevo se le asigna una de las interfaces o la otra, ambas comparten atributos (*company* y *name*) pero su atributo característico es el de *saltar* o *correr*, la función *jugar* recibe un parámetro tipo Personaje y es aquí donde el problema comienza, TS no puede saber qué método va a llegar, por lo que nos da un error, para solucionarlo se hace una comprobación antes de mandarlo a llamar:

```
interface Mario {
    company: 'nintendo',
    name: string,
    saltar: () => void
}

interface Sonic {
    company: 'sega',
    name: string,
    correr: () => void
}

type Personaje = Mario | Sonic

function jugar(personaje: Personaje) {
    if(personaje.company == 'nintendo') {
        personaje.saltar()
        return
    }

    personaje.correr()
}
```

Problema resuelto.

8.1 Type guard

Si queremos resolverlo sin la existencia de este atributo *company* hay otro método, que es mediante una función bajo la técnica **type guard**: esta técnica nos pide realizar una función para asegurar que un parámetro es de un tipo en específico según un atributo de este tipo que lo diferencie de otros similares, la sintaxis de esta técnica para este ejemplo es la siguiente:

```
function checkIsSonic(personaje: Personaje): personaje is Sonic {
    return (presonaje as Sonic).correr != undefined
}
```

Este método por defecto toma el parámetro recibido como un objeto tipo Sonic, si el resultado del método *correr* del parámetro es *undefined*, entonces el parámetro no es del tipo Sonic, de esta manera no necesitamos poner alguna condicional en nuestro código principal.

El narrowing y el type guard utilizan otro concepto normalmente llamado **discriminación de tipos**, el cual consiste en tomar tomar preferencia de un tipo sobre otro con baso a un atributo diferenciador, como podemos ver, estas técnicas siguen ese concepto para validación de datos y tipos. El código completo de este ejemplo con type guard es:

```
interface Mario {
    company: 'nintendo',
    name: string,
    saltar: () => void
}

interface Sonic {
    company: 'sega',
    name: string,
    correr: () => void
}

type Personaje = Mario | Sonic

function checkIsSonic(personaje: Personaje): personaje is Sonic {
    return (presonaje as Sonic).correr != undefined
}

function jugar(personaje: Personaje) {
    if(checkIsSonic(personaje)) {
        personaje.correr()
        return
    }

    personaje.saltar()
}
```

Nota: SE RECOMIENDA NO UTILIZAR TAN SEGUIDO EL TYPE GUARD, ya que conlleva realizar más comprobaciones y código.

9 Encapsulamiento de atributos de Clases

JS puede manejar la creación de clases al igual que el encapsulamiento de sus atributos según el enfoque esperado (privado o público), se puede agregar un # al inicio de un atributo para indicar que este es privado, al mismo tiempo que hay una convención que es agregar un _ al inicio de un atributo para indicar que este no debe ser modificado ni accedido directamente, pero igual podemos trabajarlo como un atributo público. TS si agrega los modificadores de acceso para lograr el encapsulamiento exitosamente, estos modificadores son **private**, **public** y **protected**, si tienes conocimiento sobre POO sabrás para que sirven y cual es el predefinido si no se define un modificador a un atributo o método.

```
class Avenger {
```

```
    private name: string
    private powerScore: number
    private wonBattles: number

    constructor(name: string, powerScore: number) {
        this.name = name
        this.powerScore = powerScore
    }

    get fullName() {
        return `${this.name}, de poder ${this.powerScore}`
    }
    set power(newPower: number) {
        if(newPower <= 100) {
            this.powerScore = newPower
        }
        else {
            throw new Error('Power score cannot be more than 100')
        }
    }
}

const avenger = new Avenger('spidey', 80)
avenger.name = 'Hulk' // da error.
console.log(avenger.fullName()) // no da error.
```

Cuando decimos que podemos agregar un # o _ al inicio del nombre de un atributo o método en JS es porque de esta manera lo volvemos privado o aplicamos una convención para tratarlo como miembro privado, TS es el que tiene explícitamente un modificador que vuelve un miembro privado SOLO EN LA COMPILACIÓN, de preferencia utilizar lo sugerido en JS para asegurar el alcance de los miembros en tiempo de ejecución. Aquí está el mismo código anterior pero con lo mencionado anteriormente:

```
class Avenger {
    #name: string
    #powerScore: number
    #wonBattles: number

    constructor(name: string, powerScore: number) {
        this.#name = name
        this.#powerScore = powerScore
    }

    get fullName() {
        return `${this.#name}, de poder ${this.#powerScore}`
    }
    set power(newPower: number) {
        if(newPower <= 100) {
            this.#powerScore = newPower
```

```
        }
        else {
            throw new Error('Power score cannot be more than 100')
        }
    }
}
```

10 Usando interfaces con clases

Al igual que en otros lenguajes de programación para POO, podemos hacer que una clase implemente una clase:

```
interface IAvenger {
    name: string
    powerScore: number
    wonBattles: number
}

class Avenger implements IAvenger {
    name: string
    powerScore: number
    wonBattles: number

    constructor(name: string, powerScore: number,
        wonBattles: number) {
        this.name = name
        this.powerScore = powerScore
        this.wonBattles = wonBattles
    }

    get fullName() {
        return `${this.name}, de poder ${this.powerScore}`
    }
    set power(newPower: number) {
        if(newPower <= 100) {
            this.powerScore = newPower
        }
        else {
            throw new Error('Power score cannot be more than 100')
        }
    }
}
```

11 Convención *types.d.ts*

Una convención es una forma estándar de llevar a cabo una tarea u organizar la arquitectura de un proyecto. Una convención vista en el vídeo es que metan toda las declaraciones de interfaces en un solo archivo llamado **types.d.ts**, dentro de este archivo no se puede meter otro código que no sea la declaración de una clase.

Utilizando esta convención en el ejemplo anterior quedaría de la siguiente manera:

types.d.js

```
// opcion 1.
export interface Avenger {
    name: string
    powerScore: number
    wonBattles: number
}

// opcion 2.
interface Avenger {
    name: string
    powerScore: number
    wonBattles: number
}

export default Avenger
```

main.ts

```
import {type Avenger} from './types.d'

class Avenger implements IAvenger {
    name: string
    powerScore: number
    wonBattles: number

    constructor(name: string, powerScore: number,
        wonBattles: number) {
        this.name = name
        this.powerScore = powerScore
        this.wonBattles = wonBattles
    }

    get fullName() {
        return `${this.name}, de poder ${this.powerScore}`
    }
    set power(newPower: number) {
        if(newPower <= 100) {
            this.powerScore = newPower
        }
        else {
```

```
23                 throw new Error('Power score cannot be more than 100')
24             }
25         }
26 }
```