APUNTES TYPESCRIPT HERRERA

Castear un tipo any

```
let Avenger:any = 1;

console.log(<string>Avenger.toUpperCase())

//también puedo usar as

console.log((Avenger as string).toLowerCase());
```

- Si un array es de tipo any no es necesario añadirle corchetes en el tipado porque acepta lo que venga

---

Tupla

- Cuando quiero que el primer valor siempre sea un string, el segundo un número (puedo añadir un tercero boolean)

```
const tupla: [string, number, boolean] = ["Tupla", 3, true]

tupla[1] = 3 // tupla[1] tiene que ser un número
```

Enum

```
enum AudioLevel{
    min,    //0
    mid,    //1
    max     //2
}

console.log(AudioLevel.max) // devuelve 2
```

- Puedo asignarle valores. Si no le asigno ningún valor será el valor del anterior + 1

```
enum AudioLevel{
    min = 1,
    med,        //2
    max = 20,
```

```
    super = 19,
    monster     //20
}
```

- Se recomienda usar siempre los valores del enum porque yo puedo asignarle cualquier número

```
const walkman: AudioLevel = AudioLevel.med;
```

Never

- Si una función devuelve never no debe acabar exitosamente

```
const funcionNever = (message: string):never{
    throw new Error(message)
}
```

Null y undefined

- Para hacer que los valores tipados acepten null o undefined cambiar strictNullCheck a false en el ts-config
- Por defecto undefined es asignable a una variable de tipo null

```
null !== undefined // true

let vacio:null = undefined; //no da error
```

Funciones

- Las funciones son objetos y js pasa los valores por referencia

```
const funcion1=(a:number, b:number)=> a+b

let miFuncion = funcion1 //myFuncion apunta a funcion1, NO ES UNA COPIA

miFuncion(1,2) // devuelve 3

//Yo puedo decir que
let miFuncion (x:number, y:number)=> number  //y no da error
```

Configurar consola error con archivo .ts

- En el tsconfig, si pongo SourceMap a true me muestra en consola en que linea del archivo .ts está el error
- Para remover los comentarios en el archivo .js de transpilación removeComments: true
- Para excluir archivos (o incluirlos)

```
{

    {
        //Logica tsconfig
    },
    "exclude":[
        "./path/*.js"
    ],
    "include":[
        "llllll.ts"
    ]

}
```

Clases

```
class Avenger {
    private name: string;
    public realName: string;
    static avgEdad: number;

}

const antman: Avenger = new Avenger()

//solo puedo acceder a antman.realName

//Para acceder a avgEdad solo lo puedo hacer desde la propia clase, no desde la instancia

Avenger.avgEdad = 30;

//si uso un constructor debo introducir las variables en la instancia

class Avenger {
    private name: string;
    public realName?: string;
    static avgEdad: number;
```

```
    constructor(name: string, realName?: string){
        this.name = name;
        this.realName = realName;
    }
}

const antman: Avenger = new Avenger('Antman', 'Captain')
```

Forma corta de asiganr propiedades

- Puedo crear y establecer en la misma linea sin usar el this.propiedad

```
class Avenger{
    static avgAge: number = 35

    constructor(private name: string, public realName?: string, avgAge: number =
55){ // no se puede poner static en el constructor
        Avenger.avgAge = avgAge
    }
}

console.log(Avenger.avgAge) // devuelve 55
```

Métodos públicos y privados

- Si no especifico nada el método por defecto es público
- Si le pongo el modificador de acceso private, no es accesible desde fuera de la clase (solo internamente)
- Con static puedo acceder sin hacer una instancia de la clase, sino directamente desde la clase como tal

```
class Avenger{
    static avgAge: number = 35
    static getAvgName(){
        return this.name  // me devuelve Avenger, el nombre de la clase  (las
clases también tienen un nombre)
    }

    constructor(private name: string, public realName?: string, avgAge: number =
55){ // no se puede poner static en el constructor
        Avenger.avgAge = avgAge
    }

    bio(){  //por defecto cuando no se pone nada es público
        return `${this.name} , ${this.realName}`
    }
}
```

Herencia, super y extends

```
class Avenger {
    constructor(
        public name: string,
        public realName:string
    ){
        console.log("Constructor Avenger llamado!")
    }

    protected getFullName(){  //con protected puedo acceder al método desde la propia clase y las subclases
        return `${this.name}, ${this.realName}`
    }
}

class Xmen extends Avenger {

    constructor(
        name: string,           //debo declarar las variables del padre para pasarlas al constructor super
        realName: string,
        public isMutant: boolean){  //si tiene un constructor tengo que llamar al constructor padre con super()
        super(name, realName); //debo pasarle name y realName

        //super debe ser llamado antes que cualquier otra cosa

    }

    getFullNameDesdeXmen(){
        console.log(super.getFullName())
    }
}

const Wolverine = new Xmen('Wolverine', 'Logan', true)

Wolverine.getFullNameDesdeXmen();
```

Gets y Sets

- Es lo mismo que en java

```
class Xmen extends Avenger {

    constructor(
        name: string,           //debo declarar las variables del padre para
```

```
pasarlas al constructor super
        realName: string,
        public isMutant: boolean){  //si tiene un constructor tengo que llamar al
constructor padre con super()
        super(name, realName); //debo pasarle name y realName

        //super debe ser llamado antes que cualquier otra cosa

    }

    get fullName(){     //los getters tienen que devolver un valor
        return `${this.name}- ${this.realName}`
    }

    //Se pueden disparar funciones asíncronas en los getters pero debe devolver un
valor síncrono

    set fullName(name: string){  //set solo puede recibir un argumento

       if(name.length< 3){
        throw new Error("El nombre es demasiado corto")  //puedo ponerle lógica al
setter
       }
       this.name = name
    }


    getFullNameDesdeXmen(){
        console.log(super.getFullName())
    }
}

const Wolverine = new Xmen('Wolverine', 'Logan', true)

console.log(Wolverine.fullName)

//los getters no se ejecutan ( no se invocan con parentesis)
Wolverine.fullName = "Caguetes" // al pasarle un valor con = el setter lod etecta
```

Clases Abstractas

- Las clases abstractas sirven para crear otras clases, no sirven para crear instancias

```
abstract class Mutante {
    constructor(
        public name: string,
        public realName: string
    ){}


}
```

```
//NO se puede crear una nueva instancia de una clase abstracta
//ERROR NO PERMITIDO
const wolverine = new Mutant('Wolverine', 'Logan')//ERROR
//ERROR NO PERMITIDO



class Xmen extends Mutante {
    name: string;
    realName: string;
    constructor(public team: string){
        super(name, realName)
    }
}

class Villan extends Mutante{

}

const wolverine = new Xmen('Wolverine', 'Logan', "SuperTeam')
```

- Puedo crear otras clases que extiendan de Mutante
- Si uso el tipo Mutante voy a tener disponible .name y .realName

```
const magneto = new Villan("Magneto", "Magnus")

function printName = (character: Mutante)=>{
    console.log(character.name)
}

printName(magneto)
```

- Si Villan no fuera una extensión de Mutante daría error ~-----

Constructores Privados

- Se puede usar para controlar cómo las instancias son ejecutadas
- Sirve sobretodo para manejar SingleTones (una única instancia de la clase)

```
class Apocalipsis{

    static instance: Apocalipsis;
    private constructor(public name: string){ // si le coloco private al
constructor

    }
```

```
    static callApocalipsis(): Apocalipsis{ // me va a devolver algo de tipo
Apocalipsis
        if(!Apocalipsis.instance){
            Apocalipsis.instance = new Apocalipsis("Soy Apocalipsis")
        }  // si no existe la instancia la creo
    }
}

const apocalipsis = Apocalipsis.callApocalipsis()
const apocalipsis2 = Apocalipsis.callApocalipsis() //es la misma instancia que
apocalipsis

// NO puedo crear instancias con new Apocalipsis()
```

02 TYPESCRIPT HERRERA

Interfaces

- Interfaces y tipos son muy parecidos.
- Las interfaces son más facilmente extensibles
    - Muy usadas con peticiones http
- Los tipos son más usados con patrones como redux para definir que tipo de acciones son permitidas a un objeto

Estructuras complejas

```
interface Client{
    name: string;
    age?: number;
    address?:{        //No se recomienda tener más de un nivel en una interfaz
        id: number;
        zip: string
    }
}
////////////////////////////////////////////////////////

interface Client{
    name: string;
    age?: number;
    address: Address  // de esta manera está la interfaz está solo a un nivel sin
identación, lo recomendable
}

interface Address {
    id: number;
    zip: string
}
```

```
const client: Client ={
    name: "Manuel",
    age: 25,
    address:{
        id: 125,
        zip: 'KYD'
    }
}
```

Métodos en las interfaces

- En las interfaces los métodos se definen distinto que en los tipos
- Si en los tipos lo definirias getFullName(id: string)=> void
- En una interfaz se utilizan los dos puntos

```
interface definirMetodo{
    name: string;
    getFullName(id: string): void
}
```

- Cualquiera que implemente la interfaz debe tener un name y un metodo getFullName

Interfaces en las clases

```
interface Xmen{
    name: string;
    realName: string;
    mutantPower(id: number): string
}

interface Human{
    age: number
}

class Mutant implements Xmen, Human {
    public age: number;
    public name: string;
    realName: string;

    mutantPower(id: number){
        return `${this.name}`
    }
}
```

Interfaces para las funciones

```
interface addTwoNumbers{
    (a: number, b: number): number  //función que recibe dos números y devuelve un
número
}



let addNumbersFunction: addTwoNumbers = (x:number = 1, y: number = 2)=>{
    return x + y
}
```

Namespaces

- Normalmente se usan más en la parte del desarrollo del propio framework, no tanto en el desarrollo front o back
- El NameSpace sirve de agrupador para poder usar su contenido en cualquier otro lado

```
namespace Validations{

    //para poder llamar a los métodos del NameSpace debo usar export

    export const validateText = (text: string): boolean=>{
       return (text.length > 3)? true: false
    }
    export const validateDate=(fecha: Date): boolean =>{
        return ( isNaN(fecha.valueOf())) ? false : true //si no es un número
regreso un false
    }
}

console.log(Validations.validateText("Miguel")) //devuelve true
```

- La idea de estos NamSpaces es tener en un lugar agrupado toda una lógica
- Se puede ver como una clase, solo qeu el NameSpace puede agrupar infinidad de clases dentro suyo
- Se usan bastante en creación de frameworks y librerías

03 TYPESCRIPT HERRERA

Genéricos

- En lugar de usar any uso un genérico para obtener la autoayuda y las alertas de typescript
-

```
//En las funciones de flecha el genérico se define antes del paréntesis después del igual
//En este caso devuelve el genérico también

export const genericFunctionArrow=<T>(argument : T): T=>{
    console.log(argument)
}

//declarando con function los genericos se declaran así ( en este caso devuelve un genérico también )

export function printObject2<T>(argument:T): T{
    console.log(argument)
}

genericFunctionArrow<number>((123.322233).toFixed(2)); //123,32
```

Ejemplo de función genérica en acción

```
export interface Hero{
    name: string;
    realName: string
}

export interface Villain {
    name: string;
    dangerLevel: number
}

//creo un personaje que puede ser Heroe y Villano
const deadpool = {
    name: "Deadpool",
    realname: "Mr.White",
    dangerLevel: 42
}

//como devuelve un genérico del mismo tipo, al poner . despues del paréntesis me sale la autoayuda de Hero
genericFunctionArrow<Hero>(deadpool).realName; //"Mr.White"     dangerLevel no está disponible al ser de tipo Hero
```

Agrupar imports

- Puedo meter todas las interfaces en /interfaces/index.ts para luego poder importarlas todas en la misma importación de "./interfaces"
- interfaces/index.ts

```
export {Hero} from './Hero'
export {Villain} from './Villain'
```

Ejemplo aplicado de genéricos

- Usaré la API POKEMON
- Instalo axios

> npm i axios

```
import axios from 'axios'

//async transforma mi función para que retorne una promesa
export const getPokemon = async (pokemonId: number) =>{

    const resp = await axios.get(`https:pokeapi.co/api/v2/pokemon/${pokemonId}`)
    console.log(resp)
    return 1 ;
}


getPokemon(4)
    .then( resp => console.log(resp)
    .catch(error => console.log(error))    //atrapo el error con el catch
    .finally(()=> console.log("Fin de getPokemon"))  // finally no recibe ningún
argumento
```

Mapear respuesta http

```
import axios from 'axios'

                                                    //Especifico que la promesa
resuelve en un número
export const getPokemon = async (pokemonId: number): Promise<number> =>{

    const resp = await axios.get(`https:pokeapi.co/api/v2/pokemon/${pokemonId}`)
    console.log(resp.data) // en data tengo toda la info de pokemon
    return 1
     }
```

- Pero lo que yo quiero no es devolver un número si no devolver un objeto de tipo Pokemon
- Para ello creo la interfaz Pokemon

```
export interface Pokemon{
    name: string;
    picture: string
}
```

- Puedo especificarle que lo que luce en la respuesta va aser de tipo Pokemon

```
export const getPokemon = async (pokemonId: number): Promise<Pokemon> =>{

    const resp = await axios.get<Pokemon>
(`https:pokeapi.co/api/v2/pokemon/${pokemonId}`)
    console.log(resp.data.name) // ahora tengo name y picture, pero en el .picture
no tengo nada (undefined)
                                //la interfaz sirve para decir como luce un objeto
pero no necesariamente va a tener esas propiedades
                                //el name funciona porque ya viene en la respuesta
    return resp.data

}
```

- Para mapear con una interfaz voy a la API y busco un resultado, por ejemplo

> https:pokeapi.co/api/v2/pokemon/4

- Selecciono la opción view as RAW json (o uso alguna aplicación para verlo en modo json)
- Copio todo el json
- En quickType.io coloco el nombre de la interfaz (Pokemon) en el primer campo de la columna izquierda
    - Abajo pego todo el json
    - En el gizmo de top-right me aseguro de tener seleccionado Typescript y Interfaces only
    - Se generan las interfaces necesarias
- También se puede usar una extensión de VSCode! (Quicktype.io extension)

```
getPokemon(4)
    then(pokemon=> console.log(pokemon.sprites)) //Ahora tengo toda la info de la
respuesta disponible gracias a la interfaz
```

Quicktype.io extension

- Sigo necesitando la respuesta en json copiada con ctrl + c
- La extensión es Paste JSON as Code
- Voy al command Palette (ctrl + shift + P) y busco Paste json as code
- Me pregunta el top-level, le pongo Pokemon y Enter
- Ya tengo la interfaz completa de Pokemon

---

04 TYPESCRIPT HERRERA

Decoradores

- Los decoradores pueden ponerse en linea o en linea múltiple
- No son más que una función que se utiliza para añadir o expandir la funcionalidad de un objeto

```
@aksj @poaa
sale();

@aksja
@aksj
sale()
```

- Es raro necesitar crear un propio decorador
- Pueden recibir parámetros (Factory Decorators)

```
@Component({
    selector: 'app-product',
    templateUrl: './products.html',
    styleUrls: ['./products.css']
})
```

- En Nest se pusieron muy de moda, casi todo se hace con decoradores y clases
- Si le quitas los decoradores te queda una clase y una función

```
@Controller('cats')
export class CatsController{

   @Get()
   findAll(): string {
        return 'This action return all cats'
    }
}
```

---

decoradores de clases

- Los decoradores de clases son más sencillos que los de métodos y propiedades que tienen más parámetros
- Para usar los decoradores en tsconfig experimentalDecorators: true

```
//creo mi decorador
```

```
function printToConsole(constructor: Function){
    console.log(constructor)
}

@printToConsole
export class Pokemon{
public publicApi: string = 'https://pokeapi.co'

    constructor(public name: string)
}
```

- Este decorador imprime toda la clase. Se ejecuta en el momento que se define la clase
- Se ejecuta aunque yo no tenga ninguna instancia de la clase

---

Factory Decorators

- Si quiero recibir argumentos no puedo usar la fórmula anterior para hacer un decorador
- Para ser un Factory Decorator debe retornar una función

```
const printToConsoleCondicional = (print: boolean = false): Function =>{
    if(print){
        return printToConsole; //retorno la función, no la ejecuto. Paso la
referencia
    }else{
        return ()=> console.log("No imprime")
    }
}

@printToConsoleCondicional(true)
export class Pokemon{
public publicApi: string = 'https://pokeapi.co'

    constructor(public name: string)
}
```

Ejemplo de un decorador - Bloquear prototipo

- Voy a crear un decorador que me sirva para bloquear el prototipo de una clase y no se pueda expandir

```
const bloquearPrototipo = function(constructor: Function){
    //seal es una propiedad que previene la modificacion de los atributos y
adicionar nuevas propiedades
    Object.seal(constructor)
    Object.seal(constructor.prototype)
}

@bloquearPrototipo   //los decoradores son funciones que se ejecutan en tiempo de
```

```
transpilación
@printToConsoleCondicional(true)
export class Pokemon{

public publicApi: string = 'https://pokeapi.co'

    constructor(public name: string)
}

const charmander = new Pokemon('Charmander')

//NO ME DEJA, ERROR
(Pokemon.prototype as any).customName = 'Pikachu' //Intento expandir su prototipo
añadiendo una nueva propiedad
                                                    //coloco as any para que no
importe el tipo pero he bloqueado añadir propiedades
```

- Sin @bloquearPrototipo todas las clases Pokemon tendrían customName = 'Pikachu'

Decoradores de métodos

-

```
@printToConsoleCondicional(true)
export class Pokemon{
public publicApi: string = 'https://pokeapi.co'

    constructor(public name: string)
        //No hay pokemons negativos ni con el id 5000
    savePokemonToDB(id: number){
        console.log(`Pokemon Saved in to DB ${id}`)
    }
}
```

- Puedo hacer un decorador para hacer la validación que el id esté en un rango de 1 a 800
- Se suele apuntar a hacer FactoryDecorators porque no se sabe si en un futuro vamos a necesitar más parámetros

```
function checkValidPokemonId(){

    //como decorador de método se va a disparar con varios argumentos.
    //target depende de lo que coloque, propertykey el nombre del método que estoy
decorando, y el descriptor
    return function(target:any, propertyKey:string, descriptor:
PropertyDescriptor){
        console.log({target, propertyKey, descriptor})
    }
}
```

- Se lo coloco al método savePokemonToDB
- Puedo ver en consola que
    - propertyKey: savePokemonToDB -> apunta a lo que esta decorando
    - descriptor: permite ponerlo solo lectura o escritura. Puede servir si se quiere hacer solo readonly
    - target: constructor: class Pokemon -> tengo acceso a la clase
- Lo que quiero es controlar que el id este entre 1 y 800

```
function checkValidPokemonId(){

    return function(target:any, propertyKey:string, descriptor:
PropertyDescriptor){
        //al ejecutar esta función no se ejcuta el console.log del método
savePokemonToDB
        //descriptor.value = ()=> console.log("Hola mundo")

        //Si coloco una función en el descriptor.value, se va a ejcutar con los
argumentos que haya en el método (en este caso el id)

        const OriginalMethod = descriptor.value // necesito declararlo para poder
ejecutarlo en el else. No pasa por referencia

        descriptor.value= (id: number) =>{
            if(id < 1 || id > 800){
                return console.error("El id del Pokemon debe estar entre 1 y 800)
            }else{
                OriginalMethod(id)
            }
        }
    }
}
```

- Ahora solo salva en la DB si el id de Pokemon está entre 1 y 800

Decoradores de propiedades

- para los decoradores se suele usar function para poder usar this
- Voy a crear un decorador para bloquear el publicApi de la clase Pokemon y sea solo readonly
- A pesar de ponerle private a publicApi puedo acceder desde cualquier instancia y cambiarlo (aunque Typescript se queje)
- Pongamos que lo quiero public pero no quiero que nadie lo pueda cambiar, que sea solo lectura
- El descriptor solo lo tengo disponible si decoro un método
- De hecho la lògica al decorar una propiedad es crear el objeto de descriptor (un PropertyDescriptor)

```
function readonly(isWritable: boolean = true): Function{
    return function(target: any, propertyKey: string){
```

```
      const propertyDescriptor: PropertyDescriptor = {
         get(){
             console.log(this, "getter") //cuando intento acceder a la propiedad
publicApi se dispara "getter", QUE RARO!
             return "Miguel" //de esta manera le caigo a la propiedad publicApi y
ahora vale "Miguel"
         },
         set(this, value){ //accedo al this, que es la instancia de la clase y al
value que es a lo que estoy intentando establecer
             //si esta en modo lectura no debería poder accederse al set
             Object.defineProperty(this, propertyKey, {     //el tercer argumento
es un descriptor!
                 value: value, // el value por defecto va a ser el value que estoy
recibiendo en el set.
                             //solo la primera vez que se ejecute esto se va a
poder escribir
                 writable: !isWritable, //si quiero que sea solo lectura le pasaré
un false
                 enumerable: false // lo pongo en false para que no se pueda ver!
             })
         }
      }
      return descriptor;
     }
}
```

- Rara vez te vas a sentar a crear decoradores. Vas a usar los previamente creados, es lo más habitual

---

CURSO TYPESCRIPT 1

- Para transformar un archivo a módulo y que Typescript no confunda las variables dentro del scope de otros archivos coloco export default {}
- Si dentro coloco un método podré usarlo con la notación por punto

```
export default{
    metodo
}

//otro archivo

import basics from './basics.ts'

basics.metodo()
```

?

- Para hacer un parámetro opcional en un objeto debo declarar los tipos primero para poder usar **?**

```
let objeto:{
    name: string;
    lastName?: string } = {
    name: "María"
}
```

- Puedo usar ? en expresiones para que en caso de ser undefined o null no de error

> Objeto.propiedad?.toLowerCase()

typeof

- Si quiero usar el autocompletado cuando tengo más de una opción en el tipado puedo usar **typeof**

```
function funcionPrimaria(id: string | numero){
    if(typeof id === "string"){
        id.toLowerCase()
    }
}
```

- Si tengo un array no puedo usar typeof

```
function funcionArray(people: string | string[]){
    if(Array.isArray(people)){
        return "is an Array"
    }else{
        return "is a string"
    }
}
```

Tipos

- Puedo crear tipos con **type**

```
type Id = string | number

function funcionPrimaria(id:Id){}

//Puedo usar template strings

type Protocol = "http" | "https"
type TLD = "com" | "io" | "org" | "es"

type Url = `${Protocol}://${string}.${TLD}`
```

Interfaces

- Extendemos las interfaces con **extends**

```
interface Teacher {
    name: string;
    lastname: string
}

interface MathTeacher extends Teacher{
    matter: string
}

const teacherInfo: Teacher ={
    name: "Jhon",
    lastName: "Carpenter"
}
```

Extender tipos

- Para extender tipos uso **&**. Para objetos se recomienda usar interfaces, para el resto tipos
- Para extender se recomienda usar interfaces
-

```
type User = {
    name: string
}

type Student = User &{
    goodStudent: boolean
}
```

as

- Para decirle a Typescript "yo se más que tu, se de que tipo es" uso **as**

```
const userInput = document.getelementById("user")

//Si dejo que Typescript infiera por mi el tipo será un HTML element genérico por
lo que
//no podré acceder a value que es del tipo HTMLInputElement

//Para ello uso as

if(userInput)
const castUserInput = userInput as HTMLInputElement
```

```
castuserInput.value;
//Puedo hacerlo directamente en la declaración. Uso también null para que si lo
fuera no me de error

const userInput = document.getelementById("user") as HTMLInputElement | null

//Esto me obliga a usar if como en el caso anterior, para comprobar que no es null
y poder acceder
//a .value
```

```
let name= "Mire"

if(name === "Strange") // esto no da error porque es de tipo string

//Si en lugar de usar let uso const si da error, porque entonces name es de tipo
literal "Mire"

//Puedo asignar tipos literales ( no solo string, number...)

type Direction = "Left" | "Right

function chooseDirection(direction:Direction){

}

chooseDirection("Left"); //Solo me deja Left o Right
```

- A veces a Typescript se le va la olla y no marca los errores. Haciendo un pequeño cambio y guardando aparecen
- En este caso pasa algo parecido

```
type MethodRequest = "GET" | "POST"

function peticionWeb(url: string, method: MethodRequest){}

const req = {
    url: "urlRandom",
    method: "GET"
}

peticionWeb(req.url,req.method)// req.method da error

//Puedo usar as para que lo tome como valor literal y no tipo string

peticionWeb(req.url,req.method as "GET")

//Puedo tambien usar as const en el objeto para que lo tome como valor literal
```

```
const req = {
    url: "urlRandom",
    method: "GET"
} as const

//Si quiero que solo pase con method y no con url

const req = {
    url: "urlRandom",
    method: "GET" as const
}
```

!

- Para decirle a Typescript que el valor seguro no va a ser null puedo usar **!**

```
function doSomething(something: string | null){
    parseInt(something!)
}
```

- De esta manera, con la exclamación no me da error el parseInt ya que le aseguro que no es null

Funciones

- En este nuevo caso le paso una función que lleva como parámetro un string y devuelve un boolean

```
function whatever(fn: (arg: string)=> boolean){

}

//Puedo usar rest, para ello debo declararlo como un string porque Typescript lo detecta como tal

function whatever2(...args: string[]){}

//Si la función no devuelve nada uso void

function whatever3(parameter: string): void {}

//Puedo usar desestructuracion

interface User {
    id: string;
    name: string
}

function userSearch({id, name}: User){
```

```
    console.log(id, name)
}
```

Never

- El tipo never se usa en el caso de que no puedan haber más parámetros

```
type UserName = "Carlitos" | "Snoopy" | "Mafalda"

function invalidName(name: never){
    throw new Error("")
}

function cualquiera(name: UserName){
    switch(name){
        case "Carlitos":
            return "Es Carlos";
        break;
        case "Snoopy":
            return "is Snoopy!";
        break;
        case "Mafalda":
            return "Oh, Mafalda!"
        default:
            invalidName(name)      //En este caso no va a entrar nunca.
            break;                  //Pero si añado un nuevo nombre a UserName me
dará error
    }
}
```

Clases

```
class User {
    name: string; // Si name no tiene un valor por defecto y no tiene un
constructor dará error

    constructor(name: string){
        this.name = name
    }

}

class Student {
    constructor(name: string){
        super(name) //Llamo al constructor padre
    }
}

//Para implementar una interfaz
```

```
interface Animal {
    name: string;
    move(): void
}

class Dog implements Animal{
    name: string;
    move(){

    }
    constructor(name: string){
        this.name = name
    }
}

//public es por defecto. La propiedad es accesible desde cualquier sitio.
//La propia clase, subclases, etc

class Teacher{

    public courses = [""]
    protected UploadCourses() {} //protected es solo utilizable por las subclases
    private id = 1; //Con private no se puede acceder desde fuera
                    //Solo es accesible dentro de la clase Teacher
}

new Teacher().courses;

new Teacher().id; // esto da error. id es solo accesible para uso dentro del
código de Teacher

//Puedo saber si es una instancia de Teacher con insteanceof

const Manu = new Teacher();

if(Manu insteanceof Teacher){
    console.log("yes")
}
```

02 Typescript

Tipado avanzado de objetos

```
interface User {
    username: string;
    readonly birthDate: string; //La fecha de nacimiento del usuario no va a
cambiar. Por eso uso readonly
}
```

```
const user: User ={
    username: "Rie",
    birthdate: "1997"
}

user.birthDate = //Me da error, no puedo cambiar birthDate

// Se que todas las propiedades de Teacher van a ser de tipo número

interface Teacher {
    [key:string]: number
}

const teacher:Teacher ={
    propiedad: 1,
    propiedad2: 2,
    propiedad3: "Esto es un string" //Esto da error
}
```

Satisfies

```
type CourseId = string | number;

interface Course {
    id: CourseId;
    title: string;
    description: string;
    students: number
}

interface User{
    id: string;
    name: string;
    courses: Course[] | string[] //Puede tener un arreglo de tipo Course o de
strings
}

const myUser: User ={
    id: "ssdsd",
    name:"Migue",
    courses:[{
        id:"1212",
        title: "Titulo",
        description:"Description",
        students: 2
    },
    {
        id:"3232",
        title: "Titulo2",
        description:"Description2",
```

```
        students: 2
    }]
}

myUser.courses.forEach(course=> console.log(course.title))
//course.title me da error porque course sigue la interfaz, puede ser un array de
Courses o de strings

//Yo tengo claro que es de tipo Course, uso satisfies. Le quito la implementación
de la interface a
//la declaración y se la añado con el satisfies al final


const myUser ={
    id: "ssdsd",
    name:"Migue",
    courses:[{
        id:"1212",
        title: "Titulo",
        description:"Description",
        students: 2
    },
    {
        id:"3232",
        title: "Titulo2",
        description:"Description2",
        students: 2
    }]
} satisfies User

//Ahora detecta el tipo (lo infiere Typescript) y ya tengo autocompletado de
course. y no me da error
//Si pongo un array de strings Typescript me avisa que no puedo usar .id, etc
```

Genericos en funciones

```
//Esto va a funcionar con cualquier tipo de valor pero tiene un coste

function filterValues(search: any, values: any[]){
    values.filter(value => value === search)
}

const foundValues = filterValues("Migue", ["Migue", "Perico", "Clara"]) // como es
de tipo any no reconoce
                                                                        // que es
una array de strings

//hay una forma  que es

const foundValues = filterValues("Migue", ["Migue", "Perico", "Clara"]) as
```

```
string[]

//pero hay otra mejor que es usar un genérico para expresar que puede ser de un
tipo u otro

function filterValues<T>(search: T, values: T[]):T[]{  //devuelve un array de
genérico, en el ejemplo string

    values.filter(value => value === search)
}

const foundValues = filterValues<string>("Migue", ["Migue", "Perico", "Clara"])

//Ahora search va a equivaler a string y values también
```

Genericos en tipos

- Quiero generar varios tipos que van a compartir ciertas propiedades (un id, createdAt,updatedAt)

```
 type DBRecord<T> ={
    id: number;
    createdAt: Date;
    updatedAt: Date;
    data: T
 }

type User =DBRecord<{name: string; username: string}>

//Ahora a data le está asignando el tipo que yo le he pasado, name y usernames
strings

type Course = DBRecord<{title: string; description: string}>

//Ahora tiene otro tipo de data
```

Genéricos con constrains

- Los constrains sirven para limitar los valores que le puedo pasar a un genérico

```


type UserData<T>={
    username: T
}

type MyUserData = UserData<string> //username sería de tipo string

//Pongamos que quiero limitar el genérico a string o a un objeto que tenga un name
y fullname de tipo string
```

```
type UserData<T extends string | {name: string; fullname: string}>={
    username: T
}

//Pongamos que quiero crear un tipo que reciba un genérico y saque el tipo de su
propiedad id

type IdOf<T>= T["id"] // esto da error porque en el genérico me pueden pasar
cualquier cosa
                    // us string que no tenga la propiedad id, un numero que no
tenga la propiedad id, etc

//Solución

type IdOf<T extends {id: string | number}> = T["id"] //saco del genérico la
propiedad id

type MyId = IdOf<{id:string}>
type MyId2 = IdOf<{id:number}>

//Si ahora hago una función con el parámetro MyId2 va a detectar que es un numero
y en el autocompletado
//aparecen los métodos de los números

function myFunction(id:MyId2){
    id.toFixed()//salen todos los métodos de los números
}
```

keyof

- Funciona con tipos y con interfaces

```
interface Directions{
    left: number;
    right: number
}

function moveToDirection(direction: any){
    console.log(`moving to ${direction}`)
}

//si al parámetro direction le pongo la interface Directions me va a obligar que
le pase
//un objeto con left y right y yo no quiero eso, quiero solo que me obligue a una
de las dos

function moveToDirection(direction: keyof Directions){
    console.log(`moving to ${direction}`)
}

//Me habría servido también con type en lugar de interface
```

typeof

```
let name = "Migue" //TypeScript infiere en que es de tipo string

//Si quiero que este tipo sea del mismo que name uso typeof

type UserName = typeof name;

//Un ejemplo más práctico

type GetAxis = ()=> {x: number, y : number}//GetAxis va a ser una función que
devuelva una x y number


const getAxis:GetAxis = ()=>{
    return {
        x:1,
        y:1
    }
}

//los utilityTypes son tipos que ya vienen definidos en Typescript
type Axis = ReturnType<GetAxis>; //de esta manera va adetectar que el tipo es un
objeto
                                 //que devuelve x e y number

//ReturnType obtiene el tipo que retorna esa función

//Si ahora no existiera GetAxis y le pasar a Axis ReturnType<getAxis> me daría
error
//Porque espera un tipo y uno una función, pero puedo usar typeof getAxis

//(Si no he declarado el tipo GetAxis)

type Axis = ReturnType<typeof GetAxis>

//Si quiero extraer el tipo de este Array

const fruits = ['orange', 'apple', 'melon']

type Fruits = typeof fruits; //Typescript detecta que es un array de strings

//Si le coloco el as const deberá ser un array que contenga orange, apple y melon

const fruits = ['orange', 'apple', 'melon'] as const;

//Si lo hago con un objeto

const username ={
    name: "Pepe",
    username: "Pepito"
```

```
}


type User = typeof username; //de esta manera detecta que el type User contiene un
name y username strings

//Lo mismo, si quiero que contenga los tipos literales uso as const al final del
objeto

//Puedo usar typeof en una función

function removeUser(user: typeof username)

removeUser({name:"Juan", username:"Juanito"});
```

Indexed Type Access

```
interface Course {
    id: string | number;
    title: string;
    description: string;
    students: number
}

//Si quiero sacar un tipo a través de una de las propiedades de esta interface

type CourseId = Course ["id"] // de esta manera CourseId siempre va a hacer
referencia al tipo
                            // del id de la interfaz Course

const users = [
    {
        id: "paso"
        name: "Pepe",
        username: "Pepito"
    },
     {
        id: 4,
        name: "Terelu",
        username: "Chomsky"
    }
]

//El tipo id puede ser string o number
type User = typeof users; //esto lo detecta como un array, pero yo quiero el
objeto del usuario en si

type User = typeof users[number] // De esta manera extraigo el tipo de las
propiedades de los objetos del array de users

type UserId = typeof users[number]["id"]; //extraigo el tipo del id ( me dice que
```

```
puede ser string o number)

//Entonces puedo extraer el tipo de las propiedades de un array de objetos  con
typeof y [number]
```

---

03 Typescript

Conditional Types

```
interface Animal {
    move(): void;

}

interface Dog extends Animal{
    bark(): void;
}

//Quiero comprobar si Dog extiende o no de Animal y que mi tipo sea distinto en
función a eso
//uso un ternario

type Example = Dog extends Animal ? string : number;

//Esto es útil cuando se usan genéricos

function getUserId<T>(): T extends string ? {id: string, case: string}: number{

}


const userId = getUserId<string>() // devuelve un objeto con id y case de tipo
string
//cualquier cosa que no sea un string va a devolver de tipo number
```

Creando un tipo Flatten

```
//Quiero que esto , cuando lo utilice infiera en el tipo, si le paso un array de
string que saque el tipo string

type Flatten<T> = T extends Array<infer Item> ? Item : T //infiere sobre los items
del array ( es un ternario )

type FlattenArray= Flatten<string[]> //me saca el tipo string del array
```

```
type FlattenArray2= Flatten<[string, number]> // El array puede ser de string o
number

//Si le paso algo que no es un array me devuelve el tipo original

type FlattenNumber = Flatten<number> // devuelve el tipo number

//si me pasan un array devuelve el tipo del array
```

Infer en Conditional Types

- Quiero tener un tipo que pasándole el tipo de una función extraiga lo que devuelve esa función

```
//Quiero crear un type con las misma funcionalidad como el ReturnType predefinido
en Typescript
//lo voy a limitar para que solo se pueda usar con una función
//puede recibir cualquier número de argumentos
//cualquier función va aser válida para pasar como genérico aqui
//comparo el genérico con una función y le digo que infiera el return
// si no devuelve el Return devuelve de tipo never porque nunca debería de llegar
ahi
//lo que me pase tiene que ser una función

type GetReturnType<T extends (...args: any[])=> any> = T extends (...args:any[])
=> infer Return ? Return : never
```

- Entonces:
  - El tipo es el de una función de cualquier tipo con cualquier numero de argumentos de cualquier tipo
  - Le digo que en esta función con cualquier argumento que tenga, infiera en el valor de return
  - Me devuelve never si no se cumple porque siempre se va a cumplir porque siempre debo pasarle una función

```
 function getDate(){
    return new Date()
 }

//Recibe un genérico que es el tipo de una función, por ello uso typeof para traer
el tipo
 type NewDate = GetReturnType<typeof getDate> // devuelve un tipo Date
```

Distributive Conditional Types

```
type ToArray<T>= T extends any ? T[]: never // que devuelva un array del tipo que
sea el genérico
```

```
type StringToArray = ToArray<string> // devuelve un array de strings

//Si le paso varios tipos va a inferir en los varios tipos
type StringOrNumberToArray = ToArray<string | number> //devuelve un array de
string o un array de numbers
```

- En una condición si se le pasa una unión de tipos lo que devuelva la condición se va a aplicar en cada uno de ellos

Desactivar Distributive Conditional Types

```
type StringOrNumberToArray = ToArray<string | number> // No puedo mezclar tipos en
el mismo array

//Puedo desactivar el comportamiento por defecto metiendo el genérico y el tipo
entre llaves

type ToArray<T>= [T] extends [any] ? T[]: never

type StringAndNumberToArray = ToArray<string | number> //Ahora no es un array de
strings o un array de numbers.
 //Es un array que puede contener strings y numbers en el mismo array

const myArray: StringAndNumberToArray = ["strings", 123, "hi"]
```

- Meto entre llaves el genérico y el tipo y se desactiva el comportamiento por defecto el distributive conditional types

Mapped Types

```


//puede ser un tipo y una interfaz

type User={
  username: string;
  age?: number;
  readonly birthday:string; //readonly porque no va acambiar
}

//Ahora quiero tener un tipo que va a recibir como genérico un objeto y quiero que
devuelva sus propiedades convertdias a boolean
//MakeBoolean<User>=>{username:boolean, age?:boolean, birthday:boolean}

//Para eso tenemos que iterar, y para eso sirve Mapped Types

type MakeBoolean<T> ={
  [key in keyof T]:boolean //por cada propiedad del objeto voy a devolver un
boolean
```

```
}

type UserBoolean = MakeBoolean<User> //Ahora username es de tipo boolean, age es
boolean o undefined y birthdate es boolean
```

- Hay tipos predefinidos en Typescript que hacen estos mapeos, como Partial o Required

Tipos con dos genéricos

```
//Quiero un tipo con dos genéricos, un genérico y un key
//MyPickUser<T, K> La T va a ser el tipo, y la K la propiedad de T con la que me
quiero quedar
//MyPick<User, "username">=> {username: string} // solo se quedaría con esa key
que le he pasado como genérico

type MyPick<T,K extends keyof T>={
  [property in K]: T[property] // del genérico quedate con la property. Puedes
poner property P o lo que quieras
  //property va a equivaler a cada una de las propiedades que me pasen en K
  //de mi tipo solo me voy a quedar con esa propiedad
}

type OnlyUsername = MyPick<User, "username"> // ahora es un tipo que solo tiene
username
```

Utility Types

- Los utility types son tipos que ya vienen definidos en Typescript
- Nos proporcionan utilidad para distintos escenarios
-

Awaited

- Es un tipo que recibe como genérico un tipo de promesa (por ejemplo que devuelve un string)
- Lo que va a hacer Awaited es extraer lo que resuelve esa promesa, en este caso un string

```
type TaskResult = Promise<string> // de tipo una promesa que resuleve en un
string. Le puedo pasar any

type TaskPromiseResult = Awaited<TaskResult> //resuelve en un string
```

Partial

```
interface Todo {
  name: string;
  title: string;
  date: string;
}

//Quiero actualizar uno de los campos de Todo
function updateTodoFields(todo: Todo, fields: Partial<Todo>){{  //Partial recibe un genérico, ene ste caso Todo
    fields.toLowerCase()
}

const myUser={
  name: "Pere",
  title:"casum",
  date:"12-09-2003"
}

updateTodoFields(myUser, "name") // pone el nombre todo en minúsculas
```

- Partial convierte las propiedades en opcionales

Required

- Convierte las propiedades en obligatorias

```
interface Student {
  name: string;
  username?: string
}


//Quiero hacer algo que actualice name y username (las dos obligadas) pero le he dicho en la interface que username es opcional
function updateStudentInfo(student: Required<Student>)


updateStudentInfo({
  name: "Pere",
  username: "Pit" //si no le paso las dos me marca error
})

//el codigo del tipo Required (predefinido en Typescript) es como los mapped que vimos antes

type Required<T>={
  [P in keyof K]-?: T[P] //con el menos esta quitando el modificador del opcional y devuelve todas las propiedades
}

//en el del Partial no está el menos, le añade el modificador del opcional ?
```

```
type Partial<T>={
  [P in keyof T]?: T[P]
}
```

- En lugar de crear una nueva interfaz RequiredStudent puedo usar la misma

ReadOnly

```
interface Pet {
  name: string;
  size: number
}

type ReadonlyPet = Readonly<Pet> // le añado readonly a todas las porpiedades de Pet

//El tipo predefinido Readonly le está añadiendo un modificador a todas sus propiedades iterando

type Readonly = {
 readonly [ P in keyof T]: T[P]
}
```

Record

- Sirve para designar un objeto

```
interface PositionInfo{
  amount: number;
  delay: number
}

type Position = "Left" | "Right"

//Quiero determinar un objeto cuyas keys se determinen en base a Position, que sea left y right
//Los valores van a ser determinados por PositionInfo

//{left:{amount: 1, delay:2}, right:{ amount:3, delay:2}}


type PositionComplete = Record<Position, PositionInfo> // Le paso primero lo que va a determinar las keys, y segundo los valores

const positions: PositionComplete = {
  left:{
    amount:1,
```

```
    delay:2
  },
  right:{
    amount:3,
    delay:2
  }
}
```

- Ahora el tipo PositionComplete obliga a tener las keys Left y Right, cada una con los valores amount y delay

Pick

- Pick nos sirve para pasarle como primer genérico un tipo o una interfaz que sea un objeto, y como segundo genérico con que propiedades me quiero quedar y se va a quedar solo con esa

```
interface User {
  id: number;
  email: string;
  username: string
}

function registerUser(user:User){
  //lógica
}

//en el user que le paso a registerUser todavía no tengo el id
//podrías pensar en crear otra interfaz y quitarle el id, pero hay otra manera


function registerUser(user: Pick<User, "email" | "username">){ // me quiero quedar
solo con email y username
  //lógica
}

//En el código de Typescript el tipo Pick se describe asi

type Pick<T, K extends keyof T>={
  [P in K]: T;
}
```

Omit

- En el caso anterior sería más útil quitar el id y ya está
- Para eso sirve Omit

```
function registerUser(user: Omit<User, "id">){}
```

Exclude

- Exclude sirve para dada una unión de tipos eliminar los que yo quiera

```
type UserId = string | number;

function uppercaseId( id: Exclude<UserId, number> ){ // me quedo solo con el tipo string

}

//Si quiero excluir varios los separo con pipe |
```

NonNullable

- Sirve para dada una unión de tipos quitar los que sean null o undefined

```
type PlayerPhone = string | undefined

//si yo se que esta función solo la voy a llamar cuando este phone de tipo PlayerPhone exista puedo usar el NonNullable
function updatePhone(phone: NonNullable<PlayerPhone>){ //ahora apunta a un string si o si
  phone.toUpperCase()
}
```

Parameters

- Extrae como una tupla los parámetros de las funciones

```
function feedDog(dogId: number, food: string){

}

//Quiero crear un tipo que extraiga los parámetros que le he dado a la función feedDog

type FeedDogParams = Parameters<typeof feedDog>

// si ahora quiero crear una constante de tipo FeedDogParams tiene que ser una tupla con un número y un string

const params: FeedDogParams = [1, "pienso"]
```

Manipulación de strings

```
const teachername = "Migue"

const teacherUpper:Uppercase<typeof teacherName> = "MIGUE" //Debo poner migue en
mayúscula para que no de error, pq es un literal
                                                          //al llevar const
                                                          //si hubiera usado let
tendría que ser cualquier string en mayúscula

//Lowercase para obligarme a ponerlo en minúscula

//También está Capitalize para poner la primera mayúscula

//Uncapitalize solo me va a dejar ponerlo si la primera no es mayúscula
```

Exports

```
type CourseId = string | number;

export interface Course { //le añado directamente export para exportar
  name: string;
  title: string;
  students: number;
  id: CourseId
}
```

BONUS- REACT CON TYPESCRIPT

Props

```
interface Props {
  value: number
}

export const Counter: React.FunctionComponent<Props> = ({value})=>{ // esto me
obliga a pasarle un value de tipo number
  return (
    <>

      <h1> Hi! I'm a counter </h1>

    </>
  )
}
```

```
//sería lo mismo hacerlo así

export function SuperCounter({value}:Props){
  return(
    <div>
      <h1>I'm a counter! </h1>
    </div>
  )
}

<Counter value = 0> // estoy obligado a pasarle un value
```

Children

```
export const ParentComponent = ()=>{
  return(
    <p> hi!</p>
  )
}

//Yo no puedo ponerle children al componente si no lo especifico antes
<ParentComponent>
  <SuperCounter />
</ParentComponent>
//estas lineas de arriba darían error

interface ParentComponentProps{
  children: ReactNode;
}

export const ParentComponent: React.FunctionComponent<ParentComponentProps> = ()=>
{
  return(
    <p> hi!</p>
  )
}

//si ahora pongo el componente ParentComponent sin children me marca error
//Puedo usar el condicional ? en la interfaz para hacerlo opcional

interface ParentComponentProps{
  children?: ReactNode;
}
```

useState

```
//puedo usar un genérico en el useState

const [state, setState]= useState<T>()

//si le pongo number, Counter va aser de tipo number

const [state, setState]= useState<number>()
```

useRef

```
const ref = useRef<string>() // solo me va a dejar pasarle strings

//muchas veces se declara null para usarlo cuando se lo ibamos a pasar a un
elemento

const ref = useRef(null)

//para solucionarlo le paso el elemento HTMLHeadElement al ser un h1
//Si fuera un div sería HTMLDivElement

const ref = useRef<HTMLHeadingElement>(null)
```