**Typescript (.ts)**

En la consola tsc –w para modo debug

let

Variable let vive dentro del Scope

let mensaje = 'Hola';

    console.log(mensaje);

Const

    const MENSAJE = 'Hola';

~~MENSAJE = 'Hola2';~~

mensaje no puede modificar su valor porque es constante

pesan menos en memoria , declarar todo en mayúscula

Tipos de datos

Typescript infiere el tipo de dato de la variable let y no te deja modificarlo luego

let mensaje:string = ‘mensaje’;

:number

:boolean

:Date

:any

let spiderman{

        nombre ='Peter',

        edad = 30

    };

Infiere string para nombre y number para edad , no deja asignar otra cosa ni modificar la estructura de el objeto spiderman

Templates literales

Permite construir strings de forma mas fácil.

comillas antes del número 1 del teclado

puedo realizar operaciones dentro de las llavez de ${….} como en js

const nombre   = 'Fernando';

const apellido = 'Herrera';

const edad     = 33;

const salida = `${nombre}\n ${apellido} (Edad:${edad +100})`;

puedo llamar metodos

function getEdad(){

return 100 + 100 +300;

};

const salida = `${nombre}\n ${apellido} (Edad:${getEdad() +100})`;

Funciones(parámetros opcionales,obligatorios y por defecto)

Primero defino los obligatorios (importante!!!) , después los opcionales y por último los por defecto en la firma del método.

Obligatorios:

formato ----> :tipo

function activar(quien:string){

    };

    activar('Gordon');

Por defecto:

formato ----> :tipo = valor

 function activar(quien:string , objeto:string = 'batiseñal'){

    console.log(`${quien} activo la ${objeto}`)

    };

    activar('Gordon');

    activar('Gordon','sobrescribo el valor por defecto');

opcional :

formato ----> **?:**

function activar(quien  :string,

                    momento?:string,

                    objeto  :string = 'batiseñal'){

                        if(momento){

                            console.log(`${quien} activo la ${objeto} en la ${momento}`)

                        }else{

                            console.log(`${quien} activo la ${objeto}`)

                        }

    };

    activar('Gordon');

    activar('Gordon','sobrescribo el valor por defecto');

    activar('Gordon','tarde');

Funciones Flecha

Las funciones de flecha no modifican a lo que apunta el this.

normal

const miFuncion = function(a:string){

return a.toUpperCase();

};

Flecha mas de una linea

const miFuncionFlecha = (a:string )=>{

    return a;

}

Flecha en una linea

const miFuncionFlecha2 = (a:string )=> a.toUpperCase();

console.log(miFuncion('algo'));

console.log(miFuncionFlecha2('algo mas'));

sumar en una linea

const sumarF = (a:number,b:number)=>a+b;

console.log(sumarF(2,2));

deconstrucción de variables

en vez de llamar avenger.nombre , creo la variable nombre y extraigo el valor de avenger

const avenger = {

    nombre:'Steve',

    codigo:'Capitan america',

    poder:'escudo'

}

let {nombre ,codigo} = avenger;

    console.log(nombre);

    console.log(codigo);

deconstrucción de arreglos

llaves rectas para desetructurar , extraer arreglo es un método que desestructura su parámetros que recibe (el orden importa!!!)

let {nombre ,codigo} = avenger;

    console.log(nombre);

    console.log(codigo);

const avengers:string[]=['Thor','Ironman','Spiderman'];

const [loki,,spiderman]= avengers;

console.log(loki);

// console.log(ironman);

 console.log(spiderman);

const extraerArreglo = ([,iron,]:string[])=>{

    console.log(iron);

};

Promesas

Se ejecuta el log de ‘Inicio’ , luego empieza el timeout que es lo que hace la promesa , se ejecuta el log de ‘Fin’ y luego termina el timeout dentro de la promesa y muestra el resolve si esta bien o el reject si hay error .

El then es lo que se obtiene del resolve y el catch es lo que se obtiene del reject

Las promesas son código que se ejecuta sin trancar la ejecución del código restante.

console.log('Inicio');

const promesa1 = new Promise((resolve,reject)=>{

setTimeout(() => {

    //resolve('se termino el timeout');

    reject('error del timeout');

}, 1000);

});

promesa1

        .then(mensaje => console.log(mensaje))

        .catch(error  => console.warn(error))

console.log('Fin');

:Promise<T> especifica el tipo que resuelve la promesa cuando su ejecución es correcta , osea el resolve.

const retirarDinero =(montoRetirar:number):Promise<number> =>{

    let dineroActual = 1000;

    return new Promise((resolve,reject)=>{

        if(montoRetirar > dineroActual){

            reject('No hay suficientes fondos');

        }else{

            dineroActual -= montoRetirar;

            resolve(dineroActual);

        }

    });

};

retirarDinero(500)

    .then(montoActual => console.log(`Me queda ${montoActual}`))

    .catch(error => console.warn(error))

Interfaces

El objeto xmen que recibe mi función flecha debe tener por lo menos el atributo nombre de tipo string para poder ser pasado como argumento.

const enviarMision = ( xmen:{nombre:string} )=>{

console.log(`enviamos a ${xmen.nombre} a la mision`);

}

const wolverine ={

    nombre :'Logan',

    edad:60

}

enviarMision(wolverine);

interfaces

   interface Xmen{

        nombre:string;

        edad:number;

        poder?:string;

    }

const enviarMision = ( xmen:Xmen )=>{

console.log(`enviamos a ${xmen.nombre} a la mision`);

}

const wolverine :Xmen ={

    nombre :'Logan',

    edad:60

}

El tipo Xmen obliga a crear los objetos bien , el poder puede ser opcional para que wolverine no me de error porque no lo tiene.

Importar otras clases de otro archivo

Import {nombreclase} from ‘./clases/nombreclase.class’