**TYPESCRIPT**

Instalar compilador / transpilador tsc

Instalación global:

npm install -g typescript

Intalación local al proyecto:

npm install --save-dev typescript

Podemos iniciar un Proyecto sin angular-cli de esta forma:

Npm init 🡪 crea package.json

Tsc -init 🡪 crea tsconfig.json

If you do not want to install TypeScript globally, just add it to the dependency of your project, and create an npm script for it: "tsc": "tsc".

This will work, as npm scripts will look for the binary in the ./node\_modules/.bin folder, and add it to the PATH when running scripts. Then you can access tsc using npm run tsc. Then, you can pass options to tsc using this syntax: npm run tsc -- --all o npm run – tsc -- all (this will list all the available options for TypeScript).

Variables var, let y const

La diferencia está en el ámbito o scope que se estable con cada tipo.

var a = 5;

var b = 10;

if (a === 5) {

  let a = 4; // El alcance es dentro del bloque if

  var b = 1; // El alcance es dentro de la función

  console.log(a);  // 4

  console.log(b);  // 1

}

console.log(a); // 5

console.log(b); // 1

Con let se restringe el ámbito.

Las constantes tendrán el mismo ámbito que una variable let. Solo está permitido la asignación en su declaración. Pero sí podremos alterar un miembro de la constante:

const options = { frecuency: 10, all: null };

options.frecuency = 20; // Ok

options = { frecuency: 20, all: true } // Error

Operadores

Operador delete

Elimina propiedades de un objeto o elemento de un array.

Si se borra la posición de un array, no se redimensiona el array si no que el elemento de esa posición pasa a tipo undefined.

Devuelve true si se ha podido eliminar.

let x = 1;

let obj: Object = { x: 1 };

delete x; // false. Las variables no se pueden borrar

delete obj["x"] // true

Opereador … (spread)

Se utiliza para añadir elementos de un objeto o posiciones de un array.

let numbers = [1, 2];

let moreNumbers = [4, 5, 9, ...numbers, 10]; // 4,5,9,1,2,10

let keys = { a: 1, b: 2 };

let moreKeys = { ...keys, c: 3 };

También para hacer copias de estructuras. Realmente son copias superficiales pues los objetos que puedan contener esas estructuras no se copian. Para ello habría que hacer una copia profunda.

let originalArray = [1,2,3];

let copyArray = [...originalArray];

let originalObject = { a: 1, b: 2 };

let copyObject = { ...originalObject };

También sirve como parámetro de una función para indicar que admite un número infinito de argumentos, además de usarse como argumento en sí mismo:

function sum (x: number, ...more: number[]) { } // infinitos números

let numbers: number[] = [4,5,9,10];

sum(1, ...numbers); // pasamos el array expandido

también:

function sum (a: number, ...more: number ){ }

let numbers = [4,5,9,10];

sum(1,...numbers,20,300);

false en TS

0, "" (cadena vacía), null y undefined

Operador AND (&&)

let and = false && "str"; // false

let person: Person = null;

let name = person && person.name; // undefined

Como persona no es una instancia de Persona, no tiene acceso a sus atributos, por lo que persona.nombre fallará. De esta forma nos aseguramos que esto no ocurre porque si persona es null ( y null se evalúa como false), se asignará a nombre el valor del primer operando sin evaluar el segundo. Si persona no fuera null evaluaría persona.nombre y se lo asignaría a nombre.

Operador OR (||)

let or = false || 4; // 4

let or = 10; || "or"; // 10

Si el primer operando se evalúa como true, se asignará a la variable. Si se evalúa como false, se asignará el segundo operando sea cual sea su valor. **La variable quedará tipada como el tipo unión entre ambos operadores**. Es muy útil para inicializar variables que no sabes si ya lo están.

Operadores == y ===

Con === se compara también el tipo de las variables, con == se hace una transformación o cast interno.

let a = new String("cadena");

let b = new String("cadena");

a === b; // false

a == b; // false

La evaluación es falseporque está comparando objetos, que son objetos referencia. Y esta comparación se hace en base a la dirección de memoria. Si las direcciones son iguales, los objetos también lo son. En el ejemplo hemos instanciado dos veces la clase String por lo que tenemos dos objetos con direcciones distintas. Para que la igualdad se pueda evaluar como true: let a = new String("cadena");

let b:String = a;

a === b; // true

a == b; // true

Operador + de concatenación

Requiere que al menos uno de los operandos sea any o string.

let cadena1 = "Hola, ";

let cadena2 = "Mundo";

cadena1 + cadena2; // "Hola, Mundo"

let cadena1 = "Somos";

let numero = 2;

cadena1 + numero; // "Somos 2"

Podemos convertir un tipo number, any o enum a string concatenándole la cadena vacía.

let x: any = 10;

x+""; // "10"

let n:string = "10";

console.log(typeof +n); // pinta number

let n2 = 11 +"";

console.log(typeof n2); // pinta string

Operador in

Busca el valor del operando de la izquierda, que debe ser any, string o number, en el operando de la derecha, que debe ser any, object o array devolviendo true si lo encuentra y false en caso contrario. No busca valor de los atributos, sino el nombre de los mismos.

let animales = ["Perro", "Gato", "Conejo"];

"Perro" in animales; /\* False. Perro es un valor del íncide 0 del array \*/

1 in animales; /\* true. El array contiene 3 valores con lo cual existe el índice 1 \*/

let obj = {animal:"Perro"}

"Perro" in obj; /\* False. Perro es un valor del íncide “animal” \*/"animal" in obj; // True. Animal es un índice

Control de flujo

switch

let x = 1;

switch (x) {

case 0:

case 1:

case 2:

// Hacer algo si x vale 0, 1 ó 2

break;

default:

/\* Hacer algo en el caso de que no sea ninguno de los anteriores \*/

break;

}

While

let x = 0;

while (x < 10) {

x++; // x va incrementándose en 1 en cada iteración

}

for

for (declaración; condición; actualización) {

// Código

}

for-in

Va volcando el índice de nombres de la variable temporal indice. La ventaja es que no importa de qué tipo sea el índice: numérico, cadena de caracteres, etc. Existe un problema al recorrer arrays de esta forma. Debido a que recorre todas las propiedades de array, es posible que incluso itere por las intrínsceas (como length) lo que daría lugar a comportanmientos inesperados. Para ello debe debe usarte una precondición dentro del bucle que nos asegure de que exista:

let nombres: Array<string> = ["Carlos", "José", "Lama"];

for (let indice in nombres) {

if ( nombres.hasOwnProperty(indice) ){

console.log(nombres[indice]);

}

}

for-of

En la interaction obtenemos el valor no el índice como con for-in.

let names = ["Carlos", "José", "Lama"];

for (let name of names) {

console.log(name);

}

Si compilamos para ES6, es necesario que la estructura que estemos iterando implemente un método Symbol.iterator

Tipos

let variable:tipo;

Si no se indica tipo será tipo any, es decir, cualquier tipo, a no ser que se pueda obtener de forma automática por inferencia, es decir por los propios valores de asignación.

Tipos primitivos y objetos

Los tipos primitivos son los elementales que nos proporciona un lenguaje de programación y cuyos valores son guardados en la posición de memoria que se le asigna. A su vez tenemos los tipos objeto que no son más que clases que se han construido alrededor de estos primitivos para dotarlos de más funcionalidades.

string – String

boolean – Boolean

number – Number

undefined – Undefined

null – Null

void –Void

symbol –Symbol

Si declaramos una variable con un tipo primitivo, no la podemos inicializar con el tipo objeto aunque sí se puede hacer lo inverso.

let variable1: Boolean = new Boolean(true); // Correcto

let variable2: boolean = variable1; // Incorrecto boolean/Boolean

let variable3: boolean = true; // Correcto

let variable4: Boolean = variable3; // Correcto

Template string (cadenas plantilla)

èsto es una cadena plantilla`

Permite escribir texto en varias líneas sin tener que utilizar el operador de concatenación “+”.

let a = 50;

let b = 10;

function sum(strings : string[], ...values : number[]){

return values.reduce( (prev,actual ) => previo + actual )

}

let total = sum `La suma de a y b es ${a + b} y la multiplicación es ${a \* b}`

let total; // Resultado: 560

Esto también funcionaría:

function suma(strings : string[], x: number, y : number){

strings[0]// "La suma de a y b es"

strings[1]// "y la multiplicación es"

x; // a + b = 60

y; // a \* b = 500

}

suma `La suma de a y b es ${a + b} y la multiplicación es ${a \* b}

Otros tipos

Void

Sirve para determiner que una función no devuelve valor.

Null

Cualquier tipo de variable puede tener valor nulo (null).

Undefined

Para variables no inicializadas. También podemos asigar Undefined a cualquier tipo de variable, siempre que no hayamos activado strictNullChecks::

Never

Valores que nunca pueden ocurrir.

Enum

Es una forma más amigable de representar números.

enum Animals { Dog, Cat };

let dog = Animals.Dog;

Dog representaría el 0, Cat el 1 y así sucesivamente.

Function

En TS, como en JS, una función también es un objeto y se puede almacenar como tal.

let funcion: Function = function(){};

Array

let array1: Array<tipoDato>;

let array2: tipoDato[]; /\* Son iguales arrays del tipo tipoDato \*/

let array3:number[];

let array4:number [][];//array de arrays

let array5: Array<number> = new Array<number>(); //Inicializacion

let array6: number[] = []; // Inicializacion

Object

Symbols

Sólo existen a partir de la ES6 (ES2015). Son valores únicos e inmutables, es decir, una vez declarados e inicializados no se pueden modificar y no puede haber dos iguales. Para instanciarlos se usa la función Symbol. No es posible utilizar el operador new ya que daría un TypeError. El tipo es symbol

let aSymbol = Symbol();

let otherSymbol = \*\*new\*\* Symbol(); // Error

let aSymbol = Symbol("aSymbol");// El argumento sirve como identificador

Symbol("aSymbol") === Symbol("aSymbol"); // False

Una de las grandes utilidades de los symbol es que pueden ser usados como propiedades de objetos. Además existen los llamados “bien conocidos” que son symbol predefinidos que sirven para identificar funcionalidades específicas, como los iteradores.

Cálculo de tipos / inferencia de tipos

Mejor tipo común con Null

En el siguiente caso, x es un array de strings. Esto es porque string es compatible con Null

let x = ["uno", null, "dos"];

Pero si está activa la directiva strictNullCheck, entonces x un array de string | Null (string unión Null).

Mejor tipo común any[]

Cuando un array tiene un elemento any, entonces el array es de tipo any[]

Mejor tipo común {}

Cuando un array tiene un elemento {}, entonces el array es de tipo {}, salvo que haya alguno elemento any, en cuyo caso será del tipo any[].

Mejor tipo común entre tipos primitivos

Si introducimos un tipo primitivo y otro objeto relacionados (number/Number) el mejor tipo común será el del objeto.

let array = [3, new Number(1)] // Number[]

Mejor tipo común entre tipos referencia

Si rellenamos el array con instancias de clases o interfaces el mejor tipo común será aquél que sea compatible con todos los demás. En el caso de que no exista, el resultado será {}. Aunque las clases instanciadas sean subclases de otra común, el algoritmo debe elegir entre los tipos explícitamente introducidos.

class A { }

class B { }

let array = [new A(), new B()] // A[]

---

class B extends A { }

class C extends A { }

let array = [new B(), new C()]; // B[]

typeof

Este operador es muy útil pues nos informa del tipo de dato de una variable. El resultado que nos devuelve es del tipo string

let str = typeof 22; // "number"

let str = typeof new String("probando typeof"); // "object"

let x = 5;

let z: typeof x = 2; // z es number

keyof

Obtiene las propiedades de cualquier objeto en forma de unión de cadenas literales.

type obj = { a: 1, b: 2, c: 3 };

let obj2: keyof obj; // obj2 es del tipo "a" | "b" | "c"

obj2 = "a" // Correcto

obj2 = "Other" // Incorrecto

Esto se hace relevante en los tipos mapeados Se puede utilizar junto con typeof para obtener las propiedades de del tipo de una variable let obj : {a :number, b:number, c:string};

let obj2 : keyof typeof obj; // "a" | "b" | "c"

Tipo Alias (Alias type)

Este tipo es simplemente llamar de otra forma a un tipo ya definido. Se usa la palabra reservada type. Es útil para los tipos unión e intersección

type numberAndString = number | string;

let numstr :numberAndString // Es del tipo string | number

También es compatible con los genéricos: class Collection<T> {

private list: T[];

}

type List<T> = T[] | Collection<T>;

type ListNumber = List<number>;

let list: ListNumber = [];

let list2: ListNumber = new Collection<number>();

Tipos locales

function local() {

if (Math.random() > 0.5) {

interface Baz { }

}

let bar: Bar = new Bar(); /\*Error. Bar solo accessible dentro del if \*/

class Foo { };

}

let bar: Foo = new Foo(); /\* Error. Foo solo es accessible dentro de local() \*/

Tipos Tupla (Tuple Types)

Se utiliza en los arrays y sirve para determinar distintos tipos según la posición del elemento dentro del array.

let array: [number, string] = [5, "cadena"];

A la hora de introducir elementos en el array es obligatorio, al menos, el mismo número de ellos que de tipos declarados.

let array: [number, string] = [5]; // Error

Tipos unión (Union Types)

let x:string|number = “cadena”; // ok

x = 5; // OK

x = false; // error

class A {

x: string;

}

class B {

x: number;

}

let variable: A | B;

variable.x // Es del tipo string | number

Tipos intersección (Intersection Types)

Si antes hemos visto la unión de tipos, ahora vamos a ver la intersección. El nombre puede confundir ya que podríamos entender unión de tipos como la combinación de los tipos. Realmente tiene ese nombre porque puede albergar cualquier dato de los tipos de la unión. En la intersección la mecánica es distinta. Cuando tipamos con una intersección de tipos, estamos haciendo que de forma obligatoria el valor debe ser de todos los tipos especificados:

type A = { x: string } & { z: number }

let foo: A = {x:“hola”,z:1};

Tipos mapeados

Sirven para crear plantillas de tipos. Pueden ser de 4 formas:

{ [ P in K ] : T }

{ [ P in K ] ? : T }

{ readonly [ P in K ] : T }

{ readonly [ P in K ] ? : T }

**P** siempre debe estar presente y significa "property".

**K** es la lista sobre la que P puede tomar valores y debe ser compatible con string

**T** es el tipo genérico que se devuelve.

Se trata de un tipo con propiedades **P** que deben estar incluidas en el conjunto o lista **K** (los elementos de K deben ser tipos compatibles con string). A estas propiedades se les podrá asignar el tipo **T**.

Un ejemplo sería:

**type T1 = { [P in “x” | “y”] : number};**

let a:T1;

a = {x:3,y:2};// OK

let b:T1 = {x:4,y:null};//OK

console.log (a.x); // pinta 3

console.log (b.y); // pinta null

----

a = {x:4, y=”3”}//da error:

// error:

Type '{ x: number; y: string; }' is not assignable to type 'T1'.

Types of property 'y' are incompatible.

Type 'string' is not assignable to type 'number'.

----

a = {x:10}; //da error:

//error:

Type '{ x: number; }' is not assignable to type 'T1'.

Property 'y' is missing in type '{ x: number; }'.

Otro:

**type T2 = { [P in "x" | "y"]: P };** // { x: "x", y: "y" }

t2:T2={x:”x”, y:”y”]; // OK, tanto x solo pude tomar valor “x” o null. Con propiedad y, igual.

t2:T2={x:null, y:”y”]; //OK

Otro:

type Item = {a:string, b:number, c:boolean};

**type T3 = {[P in keyof Item]:Date};**

let a:T3 = {a:new Date('2000-11-16T00:00:00'), b:new Date(), c:new Date()};

console.log("a:a =" + a.a); // a:a =Thu Nov 16 2000 00:00:00 GMT+0100

console.log("a:b =" + a.b); // a:b =Fri Nov 10 2017 09:21:09 GMT+0100

Otro:

type Item = {a:string, b:number, c:boolean};

**type T4 = {[P in keyof Item]:Item[P];**

// T4 es un tipo con las propiedades de Item y cuyos tipos son iguales a Item:

let t4:T4={a:"12ab", b:4, c:true}; // OK

Otro:

type Item = {a:string, b:number, c:Boolean};

**type T5 ={[P in keyof Item]:Array<Item>};**

let t5:T5={a:["a", "z"], b:[1 , 2], c: [true]}; //OK

t5.c=[false,false]//Ok

t5.a = undefined; //OK

console.log(JSON.stringify(t5)); //{"b":[1,2],"c":[true]}

t5.a = ["a", "z"];

console.log(JSON.stringify(t5));// {"a":["a","z"],"b":[1,2],"c":[true]}

**type T6 ={ readonly [P in keyof Item]:Array<Item>};**

let t6:T5={a:["a", "z"], b:[1 , 2], c: [true]}; //OK

t6.c=[false,false]//Error es de solo lectura

Otro:

**type T7 = {[P in keyof Item]?:Array<Item[P]>};** //Parametros opcionales

let t7:T7={a:["a", "z"], b:[1 , 2]};//Ok

t7 = {a:["a", "b"], b:[1 , 2], c: [true]}; //OK

Tipos mapeados genericos ya definidos en TypeScript

Existen cuatro tipos genéricos ya definidos en TypeScript:

Partial

Convierte todas las propiedades de un tipo en opcionales.

Lo que StypeScript ha hecho es:

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

type Item = {a:string, b:number, c:boolean};

**type T8 = Partial<Item>;**

let t8:T8={a:"a", c:true}; //OK la propiedad b no se establce

Readonly

Convierte todas las propiedades de un tipo en solo de lectura.

Lo que StypeScript ha hecho es:

**type Readonly <T> = { readonly [ P in keyof T ] : T [P ] }**

type Item = {a:string, b:number, c:boolean};

**type T8 = ReadOnly<Item>;**

let t8:T8={a:"a", b:123, c:true}; //OK

t8.b=12;// Error

Pick

Pick crea un tipo a partir de las propiedades de otro tipo manteniendo el tipo de dichas propiedades. Más concretamente crea un subconjunto de otro tipo.

**type Pick<T, K extends keyof T> = {**

**[P in K]: T[P];**

**}**

type t9= Pick<Item, "a">;

let t9:T9 = {a:"a1", b:12}; // Error porque propiedad b no existe

Record

Record crea un tipo a partir de las propiedades especificadas pudiendo establecer el tipo de dichas propiedades.

Lo que hace internamente Typescript:

**type Record<K extends string, T> = { [P in K]: T;}**

typeT10 = Record<"b" | "c" | "d", number>; // {b: number, c: number, d:number}

let t10:T10 = {c:8, b:2, d:3}; // OK, y da igual el orden

Se puede combinar con el operador \_keyof \_para construir el tipo a partir de otro

type Item = { a: string, b: number, c: boolean };

type T11= Record<keyof Item, number>; // {a: number, b: number, c:number}

t11:T11= {a:1,b:2,c:3};//OK

Combinaciones entre tipos mapeados genericos

Vamos a crear un subtipo de Item tomando sólo dos propiedades a las que le vamos a cambiar el tipo y las vamos a hacer opcionales y de sólo lectura:

type ItemAll = Readonly<Partial<Record<keyof Pick<Item, "a">, Date>>>// {readonly a?: Date}

Guardas de tipos (Save Guard)

Guardas con operadores propios del lenguaje

Se trata de los operadores **typeof** e **instanceof**

function mitad (n:string | number | {num:number}){

if (typeof n === "number"){

return n/2;

} else {

if (typeof n === "string"){

return +n/2;

} else {

return n.num/2;

}

}

}

Console.log(mitad(12));// OK muestra 6

Console.log(mitad(“12”));// OK muestra 6

Console.log(mitad({num:12}));// OK muestra 6

Guardas propios

class Person { }

class Student extends Person {

studentId: string;

}

class Teacher extends Person {

teacherId: string;

}

function isStudent(p: Person): p is Student { // Aquí lo importante

return p instanceof Student;

}

function isTeacher(p: Person): p is Teacher { // Aquí lo importante

return p instanceof Teacher;

}

No se entiende.

Desconstrución (Destructuring)

Array destructuring

let [x,y,z] = [1, 2, 3];

De esta forma tenemos 3 variables, llamadas x, y, z, con el valor 1,2,3, respectivamente Podemos asignar arrays directamente:

let array = [1, 2, 3];

let [x,y,z] = array; // x=1, y=2,z=3

let {a,b] = array; //a = 1, b = 2

function numbers() {

return [1, 2, 3]

}

let [x, , z] = numbers();// Sólo obtenemos 1 y 3.

Deconstrucción de array con resto:

var [x, y, ...remaining] = [1, 2, 3, 4];

console.log(x, y, remaining); // 1, 2, [3,4]

var [x, , ...remaining] = [1, 2, 3, 4];

console.log(x, remaining); // 1, [3,4]

Object destructuring

var rect = { x: 0, y: 10, width: 15, height: 20 };

// Destructuring assignment

var {x:a, y, width, height} = rect; // se ha redefinido x por a

console.log(a, y, width, height); // 0,10,15,20

rect.x = 10;

({x:a, y, width, height} = rect); // assign to existing variables using outer parentheses

console.log(a, y, width, height); // 10,10,15,20

Object destructuring con resto

// Example function

function goto(point2D: {x: number, y: number}) {

// Imagine some code that might break

// if you pass in an object

// with more items than desired

}

// Some point you get from somewhere

const point3D = {x: 1, y: 2, z: 3};

/\*\* A nifty use of rest to remove extra properties \*/

const { z, ...point2D } = point3D;

goto(point2D);

Objetos

Formas de declarar e inicializar un objeto:

let obj:{};

let obj:Object;

let obj = {};

let obj = new Object();

let obj = {a:1,b:2}; // en este caso ya no es posible añadir nuevas claves.

type obj = {a: string , n: number };//lo más habitual es utilizar tipos alias

let a : obj = { a: "cadena", n : 1 } // Correcto

let b : obj = { a: "cadena", n : 1, x: 1} // Incorrecto. Tiene más propiedades.

let c : obj = { a: "cadena" } // Incorrecto. Tiene menos propiedades.

let d : obj = { a: "cadena", x : 1 } // Incorrecto. Tiene el mismo número de propiedades pero no se llaman igual.

let b = { a: "cadena", n: 1, x: 1 } as obj; // correcto, se utiliza una **confirmación de tipo**

let c = { a: "cadena" } as obj; // correcto, se utiliza una confirmación de tipo

class Persona{ name:string; age:number; }

let carlos3= new Persona();

carlos3.name = "h";

carlos3.age=18;

carlos3.apellido= "jj";// error apellido no está definido en Persona

carlos3[“apellido”] = “carlos”;//el objeto solo queda con el atributo apellido

type Person = {name: string , age: number }

let jc = { name: "Carlos", age : 26, nationality: "spanish" };

let carlos : Person = jc // Correcto

//hacemos que el **tipo objeto literal pueda tener infinitos atributos** de cualquier tipo:

type Person = {name: string , age: number,[more: string ] : any }

let carlos : Person = { name: "Carlos", age : 26, nationality: "spanish" };

type Person = {name: string , age?: number }//age es **atributo opcional**

let carlos2: Person = { name: "Carlos" } // Correcto.

type Person = {name:string, age:number};

let persona:Person {name:”pepe”, readonly age:1};// OK, aqui sí podemo asignar valor

persona.age = 19;// Error age es **propiedad readonly**

Podemos acceder a las propiedades mediante

Obj[“a”] o obj.a

Las claves solo pueden ser numéricas o cadena de caracteres.

Propiedades computadas (computed properties)

La expresión que determina la propiedad debe ir entre paréntesis:

let obj = { ["property"+"name"] : 1};

Es posible hacerlo de forma compleja: let name = "carlos";

let obj = { [name.toUpperCase()] : "carlos"};

También es posible utilizar esta utilidad en clases e interfaces:

class Person{

["name"] : string;

}

interface Person{

["age"] : number;

}

Propiedades symbol

Los símbolos pueden utilizarse en las propiedades de un objeto:

let metadata : symbol = Symbol("medatada");

let obj: {} = { [medatada] : { date: Date.now() }};

JSON.stringify(obj); // {}

Object.getOwnPropertyNames(obj) // []

for (let value in obj){

console.log(value) // No muestra metadata

}

La forma de obtener las propiedades que sean symbol es la siguiente: Object.getOwnPropertySymbols(obj); // [Symbol(medatada)]

Por supuesto podemos obtener su valor con el symbol: obj[metadata]; // {date: /\*...\*/}

Funciones

Las funciones en TypeScript añaden algunas cosas con respecto a las funciones JavaScript.

Llamaremos parámetros en una función TS a los que se encuentran en la firma de la función y argumentos a los valores que se establecen al invocar dicha función.

Parámetros opcionales e inicializados

Es posible establecer parámetros opcionales y con valores por defecto. Estos parámetros **opcionales** o **inicializados** no podrán preceder a otros parámetros estándar, es decir, que no sean ni opcionales ni inicializados:

function sum(x?: number, z: number) { } // Error

function sum(x = 20, z: number): { } // Error

Parámetros infinitos

function f (a:number, b:string, … moreparams){}

moreparams, por defecto sera un Array de any, pero podemos hacer que sea un Array de cualquier otro tipo.

function f(…parámetros:numbers[]){}// uso : f(1,2,3,4,5,6);

En el caso de arriba parámetros no puede ser opcional ni intentar inicializarlo.

function more(...moreP?: number[]): void { }; // Error

function more(...moreP: number[] = []): void { }; // Error

function adder(...moreP: number[]) {

let total = 0;

for (let num of moreP) {

total += num;

}

return total;

}

let sum = adder(3, 56, 89, 12, 56); // 216

Parámetros por valor y por referencia

Los parámetros de tipo primitivo como string, number, boolean son parámetros por valor.

Los parámetros de tipo objeto o array serán parámetros por referencia.

Funciones flecha

Si tenemos la función sin nombre:

let sumar = function (a:number, b:number) {

return a + b;

}

Podríamos crear la function lambda equivalente:

let sumar = (a:number, b:number) => a + b;

también

let sumar = (a:number, b:number) => {return a+b;}

// Si utilizamos {, obligatoriamente habría que utilizer ‘return´.

o

let sumar = (a:number, b:number):number => {return a+b;}

Tipado de variables function por inferencia:

let suma = (a:number, b :number) => a + b;

let suma = function (a :number, b: number) {return a + b; };

Al estar tipada, tendremos la ayuda del editor a la hora de utilizarla. Sin embargo, si usamos:

const sumar2: Function = (a: number, b: number) => a + b; // No queda tipada

No quedaría tipada, y no tendremos las ayudas en el editor.

También podríamos tipar de forma explícita:

const suma : (a :number, b: number) => number;// tipado explicito

Aquí la flecha se utiliza para especificar el tipo devuelto por la función.

Una función estándar también puede ser tipada:

let square: (x: number) => number = function (x: number): number { return x \* x }

Formas de tipar funciones

Tipado con función lambda (flecha)

let suma = (a:number, b :number) => a + b;

Tipado por inferencia

let suma = function (a :number, b: number) {return a + b; };

Tipado mediante objeto

const funcion: { (x: number): string; (x: string): number; } = sobrecargada; // Al definer el tipo no hay que incluir una function que incluye a las demás.

function sobrecargada(x: string): number;

function sobrecargada(x: number): string;

function sobrecargada(x: any): any { return 'sobrecargada'; }

Funciones con funciones como parámetros

A una función le podemos pasar otra función como parámetro:

class Foo {

public save (callback: (n: number) => any): void {

callback(2);

}

}

// …

const foo = new Foo();

const strCallback = (result: string): void => {

alert(result);

};

const numCallback = (result: number): void => {

console.log((result \* 2).toString());

};

foo.save(numCallback); // La consola muestra 4

// foo.save(strCallback); Error debido a que strCallback tiene un parametro string y no number

Funciones que devuelven funciones

const operaciones = {

suma: function (a: number): number {return a + a; },

multiplica: function (a: number): string {return ‘a \* a’; }

};

// funcionQueDevuelveFuncion no puede devolve la función multiplicación

function funcionQueDevuelveFuncion(): (x: number) => number {

return operaciones.suma;

}

// Para probar

let f = funcionQueDevuelveFuncion();

console.log (f(2));

Conservación del contexto (this) con funciones flecha

La ejecución del código siguiente fallará porque el contexto de la ejecución show() es window por lo que asigna this a window. Window no tiene la función showLetter().

class A {

show() {

this.showLetter();

}

showLetter() {

alert("A");

}

}

let a: A = new A();

let show = a.show;

show();

// La clase B es equivalente a la clase A:

class B {

show = function () {

this.showLetter();

};

showLetter() {

console.log('B');

}

}

En los dos casos de arriba, podemos hacer que este error aparezca en tiempo de compilación con el parámetro "noImplicitThis": true, en el fichero tsconfig.json. Si ocurre este caso veremos en la salida de la compilación un error: **'this' implicitly has type 'any' because it does not have a type annotation.**

Sí funciona si utilizamos funciones flecha:

class A {

show = () => {

this.showLetter();

}

showLetter() {

alert("A");

}

}

let a: A = new A();

let show = a.show;

show();

Otro ejemplo en el que podemos conservar this.

// const regalo = {

// vehiculo : ['coche', 'moto', 'bici'],

// color : ['rojo', 'verde'],

// get : function () {

// return function () {

// const v = this.vehiculo [Math.floor(Math.random() \* 100 % this.vehiculo.length)];

// const c = this.color[Math.floor(Math.random() \* 100 % this.color.length)];

// return ({vehiculo: v, color: v});

// };

// }

// };

const regalo = {

vehiculo : ['coche', 'moto', 'bici'],

color : ['rojo', 'verde'],

get : function () {

return () => {

const v = this.vehiculo [Math.floor(Math.random() \* 100 % this.vehiculo.length)];

const c = this.color[Math.floor(Math.random() \* 100 % this.color.length)];

return ({vehiculo: v, color: c});

};

}

};

// Uso

const funcRegalo = regalo.get();

const regalito = funcRegalo();

console.log(`regalo = ${JSON.stringify(regalito)}`);

En la definición con comentarios de la variable regalo, si tenemos "noImplicitThis": false o no se indica este parámetros, en tiempo de ejecución veremos:

ERROR

AppComponent\_Host.html:1

TypeError: Cannot read property 'vehiculo' of undefined

AppComponent\_Host.html:1

ERROR CONTEXT

AppComponent\_Host.html:1

DebugContext\_ {view: Object, nodeIndex: 1, nodeDef: Object, elDef: Object, elView: Object}

La que usa flecha para definir la función a devolver funciona correctamente.

Funciones anónimas autoejecutables

Son funciones que al mismo tiempo que se definen se ejecutan.

Deben definirse entre paréntesis y es necesario el punto y coma final.

(function () {console.log('Pruebas')}) ();

(() => {console.log('Prueba con flecha')}) ();

Funciones especializadas

No se entiende

asignación de funciones a funciones

let func1 = (y: number, x: number) => x \* y;

let func2 = (x: number) => x;

func1 = func2; // Correcto

// func2 = func1; // Incorrecto, error compilación

console.log(`func1(3, 4) = ${func1(3 , 4)}`); // pinta func1(3, 4) = 4

Clases

Declaración

class Person {}

let p : Person; // Tipamos la variable p como de tipo Persona

Expresión

Con ES6 es posible crear clases con expresiones de clase:

Person = class {};

Pero de este modo no es posible tipar una variable.

Podemos crear una clase mediante una expresión y asignarle un nombre:

Pero el nombre de la clase sólo estaría disponible dentro de la propia clase y no fuera.

let Person = class Person{

/\* Aquí sí podemos usar Person \*/

}

let carlos: Person; // Error

Esto permite crear clases anónimas de tal forma que podemos crearlas en el momento de usarlas, tal como ocurre con las funciones:

function foo ( classy : any ){}

foo (class Person {});

const f = function funcion () {};

const P = class Person {};

//…

console.log (`typeof P = ${typeof P}, 'typeof (new P()) = ${typeof (new P())} y typeof f = ${typeof f}`);

// muestra >>>typeof P = function, 'typeof (new P()) = object y typeof f = function

Instanciación

let persona = new Persona();

Los atributos / propiedades

Hay que puntualizar que las clases también aceptan propiedades opcionales.

class Persona {

nombre?: string;

apellido1: string;

apellido2: string /\* (…) \*/

}

Los atributos o propiedades pueden establecerse como públicas (por defecto), protegidas y privadas.

-públicas: accesibles por cualquiera

- protegidas: accesibles por la clase y clases herederas.

- privadas: sólo accesibles por la propia clase que las define.

Los parámetros de entrada de un constructor pueden ser miembros de la clase (propiedades) de forma automática sin declararlos de forma explícita.

class Persona {

constructor(private nombre: string, private apellido1: string, private apellido2?: string) { }

public mostrarNombre(): void {

alert(this.nombre + " " + this.apellido1 + " " + this.apellido2);

}

}

La diferencia ha sido la de escribir la palabra reservada private (con public y protected también funciona) en la propia definición del método constructor. De esta forma estamos declarando de forma automática esos atributos como miembros de la clase. Y, a la hora de introducirlos como argumentos del constructor, ya se estarían inicializando. El constructor también puede ser público, protegido o privado.

Contexto (this)

La palabra clave this tiene en Javascript un comportamiento diferente al de otros lenguajes pero por lo general, su valor hace referencia al propietario de la función que la está invocando o en su defecto, al objeto donde dicha función es un método.

La palabra clave del párrafo anterior es “propietario“.

Cuando no estamos dentro de una estructura definida, esto es un objeto con métodos, el propietario de una función es siempre el contexto global. En el caso de los navegadores web, tenemos que recordar que dicho objeto es window:

const myApp = {

name : 'Antonio',

lastName : 'Feo',

completeName : this.name + this.lastName, // this no está en function de un objeto, su context es window

getCompleteName: function (): string {return (this.name + this.lastName); }

};

//…

console.log(myApp.name);

console.log(myApp.completeName);

console.log(myApp.getCompleteName());

// Resultados:

Antonio

NaN

AntonioFeo

El this aplicado en funciones no incluidas en obejtos no funcionará, apunta a undefined.

const myFunc = function () {

const name = 'Pepe';

const lastName = 'Diaz';

const completeName = name + lastName;

let getCompleteName = function (){

console.log (name + this.lastName); //Error en ejecución, this es undefined

};

getCompleteName();

};

A dónde está apuntando this en este caso? Como la función no es ahora la propiedad de un objeto, this apunta de nuevo a undefined:

ERROR

AppComponent\_Host.html:1

TypeError: Cannot read property 'lastName' of undefined

AppComponent\_Host.html:1

ERROR CONTEXT

AppComponent\_Host.html:1

DebugContext\_ {view: Object, nodeIndex: 1, nodeDef: Object, elDef: Object, elView: Object}

AppComponent\_Host.html:1

TypeError: Cannot read property 'lastName' of undefined

this polimórfico

class HTMLElement {

id: string;

public setId(id: string ) {

this.id = id;

console.log(`this en HTMLElement = ${JSON.stringify(this)}`);

return this;

}

public getId(): string {

return this.id;

}

}

class Div extends HTMLElement {

width: number;

public setWidth( width: number ) {

this.width = width;

console.log(`this en Div = ${JSON.stringify(this)}`);

return this;

}

}

//

…

let div = new Div().setId("div22").setWidth(5);

console.log("div.getId() = " + div.getId());

// en conola muestra:

this en HTMLElement = {"id":"div22"}

this en Div = {"id":"div22","width":5}

div.getId() = div22

Lo que devuelve setId no es HTMLElement, sino cualquier clase que herede de ella y que esté invocando el método en ese instante.

Getter and Setter implícitos

class Persona {

private \_name: string; // el \_ está bastante generalizado

public get name(): string{

return this.\_name;

}

public set name(name: string) {

this.\_name = name;

}

}

//…

let p = new Persona();

p.name ="Pepe";

console.log (`JSON.stringify(p) = ${JSON.stringify(p)}`);

// En consola vemos

JSON.stringify(p) = {"\_name":"Pepe"}

Herencia mediante expresiones

La herencia clásica es explícita y estática, es decir, debemos especificar claramente de qué clase se hereda. Con la herencia mediante expresiones podemos darle un poco más de dinamismo ya que, aunque una vez definida la superclase no la podamos cambiar, sí que podemos elegirla según algún criterio antes del proceso de herencia.

class Persona {

name: string;

}

function PersonaClass(): typeof Persona {

return Persona;

}

class Student extends PersonaClass() {

public curso: string;

}

// …

let s = new Student();

Otro ejemplo:

interface Fuel {

fuelType(): void;

}

class Gasoil implements Fuel {

fuelType(): void {

console.log("Gasoil");

}

}

class Gasoline implements Fuel {

fuelType(): void {

console.log("Gasoline");

}

}

interface FuelConstructor {

new (): Fuel;

}

function fuelSelector(): FuelConstructor {

return Math.random() >= 0.5 ? Gasoil : Gasoline;

}

// …

class Car extends fuelSelector() { } //

let car = new Car();

Nos hemos creado otra interfaz (FuelConstructor ) cuyo único miembro es un constructor que devuelve la interfaz Fuel. La interfaz FuelConstructor sí puede ser usada como tipo en la devolución de la función fuelSelector().

Polimorfismo

Partimos de la familia:

Stundent Person Mammal LivingBeing

let be: LivingBeing = new Stundent(); // valido pues estudiante es un ser vivo

function example(ser: LivingBeing) { }

example(new Stundent()); // OK, correcto

Los type assertions se hacen encerrando el tipo al que queremos convertir entre <> seguido de la variable que queremos convertir o con la sentencia as

interface A { }

interface B extends A { }

class C implements B { }

function conversion(casting: A): void {

let c = <C>casting;

let c1 = casting as A;

}

conversion(new C());

En el caso en el que queramos usar un método del tipo de la variable a la que queremos convertir directamente sin asignarlo a otra variable tenemos que usar los paréntesis.

class C {

toDo(): void { }

}

function conversion(casting: A): void {

(casting as C).toDo();

}

conversion(new C());

Instanceof

No compila con tipos primitivos

variable Instanceof Tipo, devuelve boolean

let x = new Number(5);

x instanceof Number; // true

let y = 5;

y instanceof number; // No compila

let num: Number = 5

num instanceof Number // false, para true debe utilizarse constructor de la clase

class A { }

class B extends A { }

let a = new A();

let b = new B();

a instanceof A // true

a instanceof B // false

b instanceof A // true, b es A y algo más

No sirve para interfaces. Es algo que sí funciona en otros lenguajes pero en TS no.

Métodos estáticos

class Person {

/\* (...) \*/

public static mostrarNombre() {

alert("estático");

}

}

Son métodos de clase y no pueden llamar a métodos no estáticos o métodos de instancia.

Podemos invocar el método estático sin crear instancia. NombreClase.NombreMetodoEstatico();

Objetos dinámicos, añadir atributos en tiempo de ejecución

Podemos añadir atributos a un objeto en tiempo de ejecución así:

class Persona { }

let carlos = new Persona();

carlos["edad"] = 26;

let edad = carlos["edad"]; //

Para no perder el tipado en cuanto a los atributos, TS añade el tipado de sus atributos en la propia clase o a nivel de clase:

class Persona { [index: string]: number; }

let carlos = new Persona();

carlos["edad"] = 26;

let edad = carlos["edad"]; // number

Estas indexaciones que se añaden a la definición de la clase se llaman diccionarios.

Podemos tener como máximo dos indexaciones, una para string y otra para number.

class Diccionario {

[index: string]: string;

[index: number]: string;

}

class Diccionario {

[index: string]: string;

[index: number]: number; // Error

}

class Diccionario {

[index: string]: string | number;

}

class Diccionario {

[index: string]: string | boolean;

[index: number]: string;

public static prueba1: boolean = true;

public prueba2: boolean;

public prueba3: string;

}

// …

let di = new Diccionario();

di['prueba2'] = false;

di['prueba3'] = 'true';

di['pruebax'] = true;

di[18] = 'dieciocho';

console.log(`JSON.stringify(di) ${JSON.stringify(di)})`);

// En consola muestra:

JSON.stringify(di) {"18":"dieciocho","prueba2":false,"prueba3":"true","pruebax":true})

Hemos observado que con JSON.stringify(di) no muestra los atributos estáticos.

Asignaciones entre objectos, comparación de clases

El tipado estructural permite comparar clases o intercambiarlas aunque no tengan una relación de herencia entre ellas. Los miembros estáticos y el constructor se ignoran a la hora de la comparación.

class Person {

static EV: number;

name: string;

}

class Dog {

name: string;

}

let person: Person;

let dog: Dog;person = dog; // Correcto

dog = person; // Correcto

Si alguna de las clases tiene algún miembro privado o protegido la comparación difiere, a no ser que:

class Animal {

private nombre: string;

}

class Dog extends Animal { }

var animal: Animal;

var dog: Dog;

animal = dog; // Correcto

dog = animal; // Correcto

Es correcto porque el miembro privado de Dog (aunque parezca que no tenga porque recordemos que los miembros privados no se heredan) es el mismo que el de Animal ya que hereda de él. Si a Dog le añadimos un miembro privado con el mismo nombre que el de Animal, no compilaría pues se llama igual y no lo permite.

Si a Dog le añadimos un miembro privado de distinto nombre:

class Animal {

private name: string;

}

class Dog extends Animal {

private raza: string;

}

var animal: Animal;

var dog: Dog;

animal = dog; // Correcto

class Animal {

private name: string;

public setName(name: string) {

this.name = name;

}

}

class Dog extends Animal {

public raza: string;

public setRaza(raza: string) {

this.raza = raza;

}

}

let animal = new Animal();

let dog = new Dog();

animal.setName('blue');

dog.raza = 'bulldog';

dog.setName("lucas");

animal = dog; // Correcto

console.log(`animal = ${JSON.stringify(animal)}`); // animal = {"raza":"bulldog","name":"lucas"}

console.log(`dog = ${JSON.stringify(dog)}`); // dog = {"raza":"bulldog","name":"lucas"}

animal.raza = "yorkshire"; // Esto daría error: Property 'raza' does not exist on type 'Animal'

Clases implementando otras clases

class A {

name: String;

doSomething() { }

}

class B implements A {

name: String;

doSomething() { }

}

Para ello se hace como si de una interfaz se tratara, con la palabra reservada implements. La clase que implementa otra se comporta como una interfaz, es decir, debe implementar sus métodos y atributos.

Los estáticos y privados no se tienen en cuenta del mismo modo que en una interfaz no se pueden declarar.

Interfaces

Todos los métodos que componen la interfaz deberán ser implementados por la clase que la implemente.

interface A {}

class B implements A {}

interface A {

metodoA ():void;

}

class B implements A {

public metodoA():void{alert(‘hola’);}

}

No es posible heredar o extender de dos o más clases pero sí es posible implementar dos o más interfaces.

Las interfaces no pueden tener miembros privados ni estáticos.

Pueden definir miembros opcionales, si no lo son tendrán que definirse en todas las implementaciones y ser del mismo tipo.

Los atributos pueden ser opcionales por lo que no obligaría a la clase a implementarlos. Además, con strictNullChecks activado, el atributo opcional sería del tipo explícitamente escrito y undefined.

interface Desplazable {

velocidad?: number; // number | undefined con strictNullChecks

desplazar(): void;

}

Herencia entre interfaces

Una interfaz puede extender a otro interfaz.

interface Desplazable extends Motriz{}

Si una clase implementa la interfaz Desplazable, no sólo debe implementar todo lo que le obliga Desplazable, sino también todo lo de Motriz. Las reglas de sobreescritura son las mismas que para las clases.

Una clase que herede de otra clase que implemente una interfaz, estaría también implementándola de forma implícita.

class Person implements Desplazable, Inteligente {

//(...)

}

class Student extends Person {

//(...)

}

La clase Student estaría implementando las interfaces Desplazable y Motriz a través de Persona, que es clase padre.

Fusionado de interfaces

Puede declarar dos interfaces con el mismo nombre sin ningún problema. Éstas se fusionarán en una sola donde sus miembros será la combinación de todos. **Se puede tener atributos con nombre repetidos ya que en ese caso el atributo sería el tipo unión de ambos**. Los métodos también pueden repetirse por lo que se aplicarían las reglas de sobrecarga de funciones. Es muy útil para crear tipos aumentados, como por ejemplo los ya predefinidos por el lenguaje como Number o String.

Comprobar si una clase implementa una interfaz

En tiempo de ejecución no hay manera de saber si una clase implementa una interfaz porque en JS no existen interfaces. La única forma sería recorrer todos los métodos de la clase para ver si implementa la interfaz, y este método podría llevarnos a error.

Interfaces extendiendo clases

En TS podemos hacer que una interfaz herede de una clase. En este caso la interfaz hereda los miembros pero no su implementación porque una interfaz no puede contener implementaciones.

class A {

public name: string;

}

interface I extends A { }

¿Qué ocurriría si la clase contiene miembros? Como hemos visto en apartados anteriores, aunque los miembros privados no se heredan, realmente sí están presentes en las clases hijas aunque no accesibles. Por ello la interfaz lo contendría pero con una curiosa consecuencia:

class A {

private name: string;

}

interface I extends A { }

Sólo las clases que hereden de A podrán implementar la interfaz I y no otras pues serán las únicas que contengan el miembro privado original.

class A {

private name: string;

}

interface I extends A { }

class B extends A implements I { } // Correcto

class C implements I { } // Error. C No es subclase de A

Muestra un error indicando que C no está implementando todos los miembros de la interfaz I, concretamente el atributo nombre. Aunque declaremos un atributo nombre de tipo string y privado en la clase C, no conseguiríamos que compilara. El único que le sirve es el de la clase A y sólo se puede conseguir heredando de ella. El comportamiento con miembros protegidos sería exactamente el mismo.

Enumerados

Este tipo relaciona un literal con un número

enum Animal { Dog, Cat, Rabbit } // Dog – 0, Cat – 1, Rabbit – 2

enum Animal { Dog = 10, Cat, Rabbit } // Dog es el 10, Cat el 11…

enum Animal { Dog, Cat = 10, Rabbit } // Dog es el 0, Cat el 11 y Rabbit el 12

// Cuidado con hacer esto:

enum Animal { Dog = 10, Cat = 9, Rabbit } // Dog es el 10, Cat el 9 y Rabbit el 10

obtener la etiqueta a partir del número

enum Animal { Dog, Cat, Rabbit } // Dog – 0, Cat – 1, Rabbit – 2

let animal = Animal[1]; // devuelve un string con Cat

let animal = Animal[Cat]; //devuelve 1

let animal: string | number = Animal['Dog']; // Si no se indica el tipo union o any falla la siguiente

animal = Animal[animal];

Podemos establecer parámetros del tipo enum:

enum Animal { Dog, Cat, Rabbit }

function showAnimal(animal:Animal){

console.log(Animal[animal]);

}

showAnimal(Animal.Dog);

const enum – Enumerados constantes

const enum Animales {perro, gato, tortuga};

animal: Animales = Animales[1];// Error

Genéricos

Los genéricos parametrizan las clases y las interfaces.

class Cajon<T> {

private numItems;

private items: T[];

constructor () {

this.numItems = 0;

this.items = [];

}

public add(item: T) {

this.items[this.numItems++] = item;

}

public get(): T {

if (this.numItems > 0) {

return this.items[--this.numItems];

} else { return null;}

}

}

// … Aplicado a number

let numeros = new Cajon<number> ();

numeros.add (3);

numeros.add (1);

console.log(numeros.get()); // muestra 1

console.log(numeros.get()); // muestra 3

console.log(numeros.get()); // muestra null

Restricciones en los parámetros

class Concesionario <T extends Automovil>{}

class Cajon<T extends Object> // aplicado al ejemplo anterior

El parámetro T solo podrá aplicarse para una clase que herede de Automóvil.

Genéricos en interfaces

interface Contenedor <T> {

get(): T;

add(item: T): void;

}

class Cajon<T extends Object> implements Contenedor <T> {

private numItems;

private items: T[];

constructor () {

this.numItems = 0;

this.items = [];

}

public add(item: T) {

this.items[this.numItems++] = item;

}

public get(): T {

if (this.numItems > 0) {

return this.items[--this.numItems];

} else {

return null;

}

}

}

Restriciciones en la parametrizacion del interfaz

Las restricciones en los parámetros los podemos indicar en la interfaz.

interface Contenedor <T extends Object> {

get(): T;

add(item: T): void;

}

class Concesionario<T extends Automovil> implements IConcesionario<T>{

//(...)

}

Funciones genéricas

Las funciones y métodos pueden ser genéricos:

function arrayDeObjetos<T>(c: T, a?: T[]): T[] {

a = a || new Array<T>(); // si a no es 0 /false/ null / ausente, crea array

a.push(c);

return a;

}

interface Contenedor <T> {

get(): T;

add(item: T): void;

}

**function addExterno <T> (item: T, array: T[]) {**

**array.push(item);**

**}**

class Cajon<T extends Object> implements Contenedor <T> {

private numItems;

private items: T[];

constructor () {

this.numItems = 0;

this.items = [];

}

public add(item: T) {

this.items.push(item);

}

public add2(item: T) {

addExterno (item, this.items);

}

public get(): T {

if (this.items.length > 0) {

return this.items.pop();

} else {

return null;

}

}

}

Control de errores

En TS hablamos de errores y no de excepciones como en otros lenguajes.

Tipos de errores

Hay algunos ya predefinidos que resultan muy útiles. Cabe decir que todos heredan de Error. **EvalError**: Indica un error al ejecutar la función global eval().

**RangeError**: Indica que estamos tratando de acceder a un índice que no existe. Es aplicable a los arrays y a los objecs.

**ReferenceError**: Indica que estamos referenciando a una variable no declarada.

**SyntaxError**: Indica que hay un error de sintaxis.

**TypeError**: Indica que estamos tratando de acceder a miembros de algo que vale null o que no hemos pasado la variable con el tipo correcto como argumento.

Todos comparten una serie de atributos como el message, que describe el error de forma literal. Normalmente, los navegadores modernos describen estos errores de forma automática en su consola

Catch, throw y finally

function errorExample(): void {

let example = null;

example.edad = 10;

}

try { errorExample(); }

catch (e) {

console.error(e) /\* Mostramos el error \*/

throw e; // Si queremos lanzar nuestro error new RangeError("Error en errorExample");

console.info("No se muestra"); // esto no se muestra

}

finally {

/\* Se ejecuta siempre\*/

console.info("finally");

}

Control de bug

strickNullChecks a true

Esta directiva lo que hace es separarlos y no permitir que \_null \_y \_undefined \_sean asignables a todos los tipos, sólo a los que lo especifique de forma explícita y a ellos mismos.

Con esta directiva a true, este es el comportamiento del compilador:

const nombre: string = null; // devuelve: Type 'null' is not assignable to type 'string'.

// …

function countWords(str: string) {

return str.length;

}

countWords('Carlos'); // Ok

countWords(null); // Error. [ts] No se puede asignar un argumento de tipo 'null' al parámetro de tipo 'string'.

}

// …

function countWords(str: string | null) {

return str.length; //[ts] El objeto es posiblemente "null".

countWords('Carlos'); // Ok

countWords(null);

}

// …

function countWords(str: string | null) {

// OK, sin error

if (str) {

return str.length;

}

}

countWords('Carlos '); // OK

countWords(null); // OK

Otro ejemplo:

function countLines(text?: string[]): number {

let count = 0;

for (const line of text) { // Object (text) es posiblemente undefined

if (line.length !== 0) {

count = count + 1;

}

}

return count;

}

let a = countLines(['one', 'two', '', 'three']); // OK

let b = countLines(['hello', null, 'world']); // [ts]

No se puede asignar un argumento de tipo '(string | null)[]' al parámetro de tipo 'string[] | undefined'.

El tipo '(string | null)[]' no se puede asignar al tipo 'string[]'.

El tipo 'string | null' no se puede asignar al tipo 'string'.

El tipo 'null' no se puede asignar al tipo 'string'..

let c = countLines(); // OK

let e = countLines(undefined); //OK

// …………………………………………………..

function countLines(text?: string[]): number {

let count = 0;

if (!text) {return count; }

for (const line of text) {

if (line.length !== 0) {

count = count + 1;

}

}

return count;

}

let a = countLines(['one', 'two', '', 'three']); // OK

let b = countLines(['hello', null, 'world']); // [ts]

No se puede asignar un argumento de tipo '(string | null)[]' al parámetro de tipo 'string[] | undefined'.

El tipo '(string | null)[]' no se puede asignar al tipo 'string[]'.

El tipo 'string | null' no se puede asignar al tipo 'string'.

El tipo 'null' no se puede asignar al tipo 'string'..

let c = countLines(); // OK

let e = countLines(undefined); //OK

// ……………………………………………………..

function countLines(text?: (string | null) []): number {

let count = 0;

if (!text) {return count; }

for (const line of text) {

if (line.length !== 0) { // error [ts] El objeto es posiblemente "null"

count = count + 1;

}

}

return count;

}

let a = countLines(['one', 'two', '', 'three']);

let b = countLines(['hello', null, 'world']); // OK.

console.log(`countLines() = ${countLines()}`);

console.log(`countLines(undefined) = ${countLines(undefined)}`);

// ……………………………………………………..

function countLines(text?: (string | null) []): number {

let count = 0;

if (!text) {return count; }

for (const line of text) { // OK

if (line && line.length !== 0) {

count = count + 1;

}

}

return count;

}

let a = countLines(['one', 'two', '', 'three']);

let b = countLines(['hello', null, 'world']); // OK.

console.log(`countLines() = ${countLines()}`);

console.log(`countLines(undefined) = ${countLines(undefined)}`);

DOM

El objeto window es el que está posicionado en la parte superior de esta jerarquía. Éste contiene al objeto document que el que posee los métodos necesarios para la manipulación del DOM.



Módulos

Una nota sobre la terminología: es importante tener en cuenta que en TypeScript 1.5, la nomenclatura ha cambiado. Los "módulos internos" son ahora "espacios de nombres". Los "módulos externos" ahora son simplemente "módulos", para alinearse con la terminología de ECMAScript 2015 (es decir, que el módulo X {es equivalente al espacio de nombres preferido ahora X {).

La forma que tenemos en typescript para ordenas nuestras clases es mediante los módulos, que vienen a ser los paquetes de java. Se irán agrupando en estos módulos por funcionalidad de las clases.

module M {

// Código

}

Es similar a: namespace M {

// Código

}

Para que las clases definidas en estos módulos sean accesibles fuera del módulo , se tendrá que etiquetar compo export.

namespace M {

export class A { }

}

// Para acceder a ella basta con escribir el nombre del módulo seguido del nombre de la clase.

var a: M.A = new M.A();

También pueden contener variables ya inicializadas o no.

namespace M {

export class A { }

export var a = 1;

}

Se puede declarar una ruta completa para un módulo de la forma:

namespace M.A.B {

export class A { }

}

Y para acceder a la clase A habría que escribir: M.A.B.A

Partes de un módulo

Cuando hacemos referencia al módulo a través de su nombre original, en este caso M, estamos accediendo a la **parte de las interfaces, la no instanciada**. Pero si accedemos a través de una variable del tipo del módulo, estamos accediendo a la **parte instanciada que comprende las variables, clases, enumerados y funciones**. Realmente un módulo es un objeto de JS. Recordemos que las interfaces en TS no tienen equivalencia en JS, por lo que cuando declaramos una no genera código.

namespace M {

export class A { }

export interface B { }

export var a: number = 1;

}

let m: typeof M = M; /\* Ahora con m podemos acceder a las entidades que no sean interfaces \*/

m.a = 2 // Correcto

m.A // Correcto

m.B // Error. Es una interfaz.

let B: M.B; // Correcto.

LAS PRUEBAS CON NAMESPACES NO HAN FUNCIONADO. HAY QUE PROFUNDIZAR …

Iteradores

A value is considered iterable if it has a method whose key is the symbol [1] Symbol.iterator that returns a so-called iterator. The iterator is an object that returns values via its method next(). We say: it enumerates items, one per method call.

let arr = ['a', 'b', 'c'];

let iter = arr[Symbol.iterator]();

console.log(iter.next()) // pinta “{done:false, value: “a”}

Podemos construir nuestros propios iteradores. Podemos ver si algún tipo tiene ya iterador, podemos utilizar:

let nums = [1, 2, 3];

let iterator = "iterator";

let x = 26;

let obj : {} = {name : "Carlos", age: 26}

nums[Symbol.iterator]; // function

iterator[Symbol.iterator]; // function

x[Symbol.iterator]; // undefined

obj [Symbol.iterator]; // undefined

interface Number {

[Symbol.iterator]();

}

Number.prototype[Symbol.iterator] = function () {

const num: string = this.valueOf().toString();

var index = num.length;

return {

next: function () {

if (index >= 0) {

index -= 1;

return { value: num.charAt(index), done: false };

}

return { value: undefined, done: true };

}

}

}

//…

let num = 123;

for (n of num){ // NO FUNCIONA : Type must have a '[Symbol.iterator]()' method that returns an iterator.

// con ES5 devuelve el sisguiente ERROR: Type 'number' is not an array type or a string type.

console.log(n);

}

Otro ejemplo con función que devuelve Iterable:

function objectEntries(obj) {

let index = 0;

// In ES6, you can use strings or symbols as property keys,

// Reflect.ownKeys() retrieves both

let propKeys = Reflect.ownKeys(obj);

return {

[Symbol.iterator]() {

return this;

},

next() {

if (index < propKeys.length) {

let key = propKeys[index];

index++;

return { value: [key, obj[key]] };

} else {

return { done: true };

}

}

};

}

let obj = { first: 'Jane', last: 'Doe', segundo: 'dos' };

for (let [key,value] of objectEntries(obj)) {

console.log(`${key}: ${value}`);

}

// …

Este método funciona si establecemos: "noImplicitThis": false,

La salida es :

first: Jane

last: Doe

segundo: dos

Generadores

Los generadores son funciones que pueden ser pausadas y continuadas.

En typescript solo se utilizan para implementar iteradores.

function\* generator() {

yield "José";

yield "Carlos";

}

let gen = generator();

gen.next(); // {value: José, done: false}

gen.next(); // {value: Carlos, done: true}

function\* generator() {

yield "José";

yield "Carlos";

return "Lama";

yield "Ponce";

}

let gen = generator();

gen.next(); // {value: José, done: false}

gen.next(); // {value: Carlos, done: false}

gen.next(); // {value: Lama, done: true}

gen.next(); // {value: undefined, done: true}

El ejemplos siguiente solo funciona configurando tsconfog.json con la opción de compilación “target”: es6. Con es5 da este error: *Type 'IterableIterator<string>' is not an array type or a string type.*

function\* numIterator(num: number) {

let n = parseInt(num.toString()).toString();

let length = n.length;

for (let i = length - 1; i >= 0 ; --i){

yield n.charAt(i);

}

}

for ( let n of numIterator(2387) ){

console.log(n) // 7, 8, 3, 2

}

Para poder iterar sobre el propio número:

Promise, async – await

El objeto promise se introdujo con ES6, async + await en ES2016.

Las promesas son una alternativa a las funciones callbacks para la obtención de resultados de llamadas asíncronas.

function asyncFunc() {

return new Promise(

function (resolve, reject) {

···

resolve(result);

···

reject(error);

});

}

La llamada a la asynFunc sería:

asyncFunc()

.then(result => { ··· })

.catch(error => { ··· });

.then retorna siempre una promesa que permite encadenar llamadas asíncronas.

asyncFunc1()

.then(result1 => {

// Use result1

return asyncFunction2(); // (A)

})

.then(result2 => { // (B)

// Use result2

})

.catch(error => {

// Handle errors of asyncFunc1() and asyncFunc2()

});

catch maneja los posibles errores tanto de asyncFunc1 como de asyncFunc2. Es decir, los errores no manejados, se pasan hasta que son manejados.

**Promise is just an object that represents a task**. It might finish right now or in a little while. We could interact with „task” by passing a callback to its then function.

Una función asíncrona se declara con async, no es más que una función que devuelve una promesa. Lo gracioso del tema es que no tenemos que declarar el objeto Promise en ningún momento.

Estas funciones asíncronas, a su vez, pueden llamar a otras funciones que devuelven promesas. En lugar de .then, Promise.all y demás, podemos utilizar await. Esta instrucción detiene la ejecución de una función hasta que la promesa se ha resuelto, resolviendo así todo en una única línea.

No estamos obligados a utilizar awiat pero si no se hace obtendremos directamente una promesa en lugar de los valores esperados.

// Aquí no hay espera simultánea, se espera a que finalize getFool y luego getBar

let foo = await getFoo();

let bar = await getBar();

Una forma de obtener una espera simultánea, es decir, esperar la finalización de todas las promesas pendientes:

let [foo, bar] = await Promise.all([getFoo(), getBar()]);

Fundamentally, Promise.all will take an array of promises, and compose them all into a single promise, which resolves only when every child promise in the array has resolved itself.

Api del objeto promise:

**Promise.all(iterable)**

‘Promise.all’ devuelve una promesa que se resuelve cuando todas las promesas del argumento iterable han sido resueltas. Un ejemplo de esto es lo siguiente:

const pr1 = fetch('flowers.jpg');

const pr2 = fetch('clouds.jpg');

const pr3 = fetch('animals.jpg');

Promise.all([pr1, pr2, pr3])

.then(doSomething)

.catch(doError);

**Promise.race(iterable)**

Recibe un array de promesa como el anterior método, sin embargo, se resuelve o se rechaza tan pronto una de las promesas se resuelva o rechace.

**Promise.reject(reason)**

Devuelve un objeto ‘Promise’ que es rechazado con la razón dada.

**Promise.resolve(value)**

Devuelve un objeto ‘Promise’ que es resuelto con el valor dado.

El prototipo de ‘Promise’ contiene dos métodos más:

**Promise.prototype.catch(onReject)**

Que permite indicar una función a ejecutarse si la promesa pasa al estado rechazado.

**Promise.prototype.then(onFufilled, onRejected)**

Que permite indicar una función de completado y rechazado para según el estado en que pase la promesa.

function isDinnerTime() {

return new Promise(function(resolve, reject) {

setTimeout(function () {

const now = new Date();

if (now.getHours() >= 22) {

resolve('yes');

} else {

reject('no');

}

}, 5000);

});

}

isDinnerTime()

.then(data => console.log('success', data))

.catch(data => console.log('error', data));

const resolver = (msg, timeout) => new Promise((resolve) => {

console.log(msg);

setTimeout(resolve, timeout);

});

resolver('First', 500)

.then(() => resolver('Second', 500))

.then(() => resolver('Third', 1000))

.then(() => resolver('Fourth', 500));

Lo mismo pero con async + await

const resolver = (msg, timeout) => new Promise((resolve) => {

console.log(msg);

setTimeout(resolve, timeout);

});

async function run() {

await resolver('First', 1000);

await resolver('Second', 500);

await resolver('Third', 1000);

await resolver('Fourth', 500);

}

run();

//… NOTA

No se encuentra diferencia al quitar los await de la función async run().

async function main() {

await ping();

console.log("acaba de finalizar await ping()");

}

async function ping() {

for (var i = 0; i < 3; i++) {

await delay(300);

console.log('acaba de finalizar await delay(300) \*\*\* ping');

}

}

function delay(ms: number) {

return new Promise(resolve => setTimeout(resolve, ms));

}

main();

console.log("despues de llamar a main()");

// … SALIDA:

*despues de llamar a main()*

*acaba de finalizar await delay(300) \*\*\* ping*

*acaba de finalizar await delay(300) \*\*\* ping*

*acaba de finalizar await delay(300) \*\*\* ping*

*acaba de finalizar await ping()*

Uso de promise con .then + .catch V (async + await)

function isDinnerTime() {

return new Promise(function (resolve, reject) {

setTimeout(function () {

const now = new Date();

if (now.getHours() >= 16) {

resolve('yes');

} else {

reject('no');

}

}, 4000);

});

}

async function asyncIsDinnerTime() {

try {

var val = await isDinnerTime();

console.log(`OK - Resultado desde asyncDinnerTime : ${val}`);

}

catch (err) {

console.log(`ERROR - Resultado desde asyncDinnerTime : ${err}`);

}

}

isDinnerTime()

.then(data => console.log(`OK - Resultado desde isDinnerTime : ${data}`))

.catch(data => console.log(`ERROR - Resultado desde isDinnerTime : ${data}`));

console.log('Lanzado isDinnerTime()');

asyncIsDinnerTime();

console.log('Lanzado asyncIsDinnerTime()');

// … SALIDA:

Lanzado isDinnerTime()

Lanzado asyncIsDinnerTime()

OK - Resultado desde isDinnerTime : yes

OK - Resultado desde asyncDinnerTime : yes

function pinta (p ?: string) {

if (p) {

console.log(`${getHora()} pinta llamado desde ${p}`);

} else {

console.log(`${getHora()} pinta llamado sin parametro`);

}

}

function getHora (): string {

let dt = new Date();

return dt.getHours() + ':' + dt.getMinutes().toString() + ':'

+ dt.getSeconds().toString() + ':' + dt.getMilliseconds();

}

function f6 () {

return new Promise (function (resolve, reject) {

console.log (`${getHora()} f6`);

setTimeout (pinta, 1000);

resolve(getHora() + ' datos promesa f6');

});

}

var p1 = new Promise((resolve, reject) => {

console.log (`${getHora()} p1`);

setTimeout(resolve, 1000, getHora() + " datos promesa p1");

});

var p2 = new Promise((resolve, reject) => {

console.log (`${getHora()} p2`);

setTimeout(resolve, 2000, getHora() + " datos promesa p2");

});

var p3 = new Promise((resolve, reject) => {

setTimeout(resolve, 3000, getHora() + " datos promesa p3");

console.log (`${getHora()} p3`);

});

var p4 = new Promise((resolve, reject) => {

console.log (`${getHora()} p4`);

setTimeout(pinta, 1000);

let t = getHora();

resolve (t + " datos promesa p4");

});

var p5 = new Promise((resolve, reject) => {

//reject("reject");

pinta('p5');

resolve(getHora() + " datos promesa p5");

//setTimeout(resolve, 1000, "four");

});

Promise.all([p1, p2, p3, p4, p5, f6()]).then(values => {

console.log(values);

}, reason => {

console.log(reason)

});

//From console:

//"reject"

// Evenly, it's possible to use .catch

Promise.all([p1, p2, p3, p4, p5]).then(values => {

console.log(values);

}).catch(reason => {

console.log(reason)

});